**What is Angular :**

1. The Angular is a development platform for building a Single Page Application for mobile and desktop.
2. It uses Typescript & HTML to build Apps. The Angular itself is written using the Typescript.
3. It comes with features like [component](https://www.tektutorialshub.com/angular/angular-components/), [Directives](https://www.tektutorialshub.com/angular/angular-directives/), [Forms](https://www.tektutorialshub.com/angular/angular-forms-fundamentals/), [Pipes](https://www.tektutorialshub.com/angular/angular-pipes/), [HTTP Services](https://www.tektutorialshub.com/angular/angular-httpclient/), [Dependency Injection](https://www.tektutorialshub.com/angular/angular-dependency-injection/), etc.
4. Angular is a UI framework for building mobile and desktop web applications.
5. Using Angular you can build amazing client-side applications using HTML, CSS, and Typescript.

Angular 10 is released on 24-06-2020.

#### **Version History : Angular 1 2,4,5,6,7,8,9,10,11**

**Angular 2** is now known as **Angular**is the new front end framework and is the successor to the most popular AngularJs. The Angular is an open-source and helps us build dynamic & single-page applications (SPAs).

Angular has many improvements over AngularJS. It has lots of innovations, which makes it easy to learn and develop enterprise-scale applications. You can build extendable, Maintainable, Testable and Standardized Applications using Angular.

**Features of Angular :** Some of the features are listed below :

* **Two-Way [Data Binding](https://www.tektutorialshub.com/angular/angular-data-binding/)**This is the coolest feature of the Angular. Data binding is automatic and fast. changes made in the View is automatically updated in the component class and vice versa
* **Powerful [Routing](https://www.tektutorialshub.com/angular/angular-routing-navigation/) Support**  
  The Angular Powerful routing engine loads the page asynchronously on the same page enabling us to create a Single Page Applications.
* **Expressive HTML**  
  Angular enables us to use programming constructs like if conditions, for loops, etc to render and control how the HTML pages.
* **Modular by Design**Angular follows the modular design. You can create [Angular modules](https://www.tektutorialshub.com/angular/angular-modules/) to better organize and manage our codebase
* **Built-in Back End Support**  
  Angular has built-in support to communicate with the back-end servers and execute any business logic or retrieve data
* **Active Community**  
  Angular is Supported by google and has a very good active community of supporters. This makes a lot of difference as your queries are quickly resolved.

Angular has changed massively from the AngularJS. Angular completely redesigned from scratch. There are many concepts of angularJS that have changed in Angular.

**Key differences between AngularJs & Angular :**

#### **Support for ES6 :**Angular is completely written in Typescript and meets the ECMAScript 6 specification. This means that it has support for ES6 Modules, Class frameworks, etc.

#### **Components are new controllers:** In AngularJS we had Controllers. In Angular Controllers are replaced with [Angular Components](https://www.tektutorialshub.com/angular/angular-components/).In Angular, we are using Components. The simple component is written using Typescript.

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

@Component({

selector: 'app',

template: '<h1>{{message}} </h1>'

})

export class AppComponent {

message: string=’Hello Angular’;

}

The components can have child components and parent components.

**Directives :** The AngularJS had a lot of directives. Some of the most used directives are ng-repeat & ng-if

The Angular also has [directives,](https://www.tektutorialshub.com/angular/angular-directives/) but with a different syntax. It has a \* before the directive name indicating it as a structural directive.

<ul>

    <li \*ngFor =#customer of customers>

        {{customer.name}}

    </li>

</ul>

<div \*ngIf=”vm.isVIP”>

    <h3> VIP Customer </h3>

</div>

The style directives like ng-style, ng-src & ng-href are all gone. These are now replaced by property binding HTML elements to the class properties

We can create a Custom Directives.

@Directive({

   selector: '[MyDirective]'

})

**class** MyDirective { }

**Data Bindings :**The powerful angular data bindings stay the same,  with minor syntax changes.

##### **Interpolation**

*//AngularJS*

<h3> {{vm.customer.Name}}</h3>

*//Angular*

<h3> {{customer.Name}}</h3>

**One way Binding :** The Angular can bind to any property of the HTML element.

*//AngularJS*

<h3> ng-bind=vm.customer.name></h3>

*//Angular*

<h3 [innerText]=”customer.name” ></h3>

##### **Event Binding :** The AngularJS uses the ngClick directive to bind to the event. In Angular ngClick Directive is removed. You can bind directly to the DOM events.

*//AngularJS*

<button ng-click=”vm.save()”>Save<button>

*//Angular*

<button (click)=”save()”>Save<button>

##### **Two- way binding**

*//AngularJS*

<input ng-model=”vm.customer.name”>

*//Angular*

<input [(ng-model)]=”customer.name”>

**$scopes are out :**Angular is not using $scope anymore to glue view and controller.

AngularJS used to run a dirty checking on the scope objects to see if any changes occurred. Then it triggers the watchers. And then it used to re-running the dirty checking again.

The Angular is using zone.js to detect changes. Zone.js apply patches on all the global asynchronous operations like click event, timer events, HTTP requests, etc. It then intimates the Angular, whenever the changes occur in Angular Application. The Angular then runs the change detection for the entire application

**Filters are renamed to Pipes :**In AngularJS, we used Filters and as shown below

<td>{{vn.customer.name | uppercase}}</td>

Angular uses the same syntax but names them as [pipes](https://www.tektutorialshub.com/angular/angular-pipes/)

td>{{customer.name | uppercase}}</td>

**Platform-specific Bootstrap :** In AngularJS we used the ng-app directive in our HTML, then the Angular would bootstrap and attach itself the ng-app

<body ng-app=’app’> </html>

The bootstrapping in Angular is done through code. The bootstrapping of Angular is not simple as that of AngularJS. The sample code below shows how Angular application bootstraps the AppModule using **platformBrowserDynamic** Module

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

import {AppModule } from './app.module'; platformBrowserDynamic().bootstrapModule(AppModule);

The Bootstrap is also Platform-specific in Angular. You can have different bootstrapper for mobile & Web application.

**Services :** The AngularJS had Services, Factories, Providers, Constants and values, which used to create reusable code. These are then injected into Controllers so that it can use it.

**Mobile Support :** AngularJS was not built with mobile support in mind. Angular designed with mobile development in mind.

**Components :**

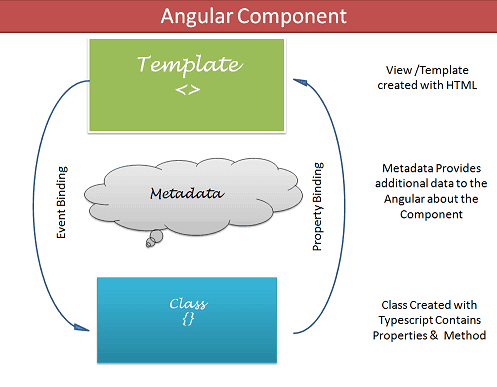
1. The Component is the main building block of an Angular Application.
2. A Component contains the definition of the View and the data that defines how the View looks and behaves.
3. The Angular Components are plain javascript classes and defined using @component Decorator. This Decorator provides the component with the View to display & Metadata about the class

The Component passes the data to the view using a process called Data Binding. This is done by Binding the DOM Elements to component properties. Binding can be used to display component class property values to the user, change element styles, respond to a user event, etc.

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

The Components consists of three main building blocks

* **Template**
* **Class**
* **MetaData**



**Template (View) :** The template defines the layout of the View and defines what is rendered on the page.  Without the template,  there is nothing for Angular to render to the DOM.

The Templates are created with HTML. You can add [Angular directives](https://www.tektutorialshub.com/angular/angular-directives/) and bindings on the template.

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

* Defining the Template Inline
* Provide an external Template

**Class :** The class is the code associated with Template (View). The Class is created with the Typescript. Class Contains the Properties & Methods.

The Properties of a class can be bind to the view using Data Binding.The Component classes in Angular are prefixed with the name “Component”, To easily identify them.

export **class** AppComponent{

    title : **string** ="app"

}

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

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

**@Component decorator :** A class becomes a Component when Component Decorator is used. A Decorator is always prefixed with @. The Decorator must be positioned immediately before the class definition. We can also build our own decorators.

**Important Component metadata properties**

**Selector :**Selector specifies the simple CSS selector, where our view representing the component is placed by the Angular.The angular places the view (template) inside the selector app-root

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

**Directives :**

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

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

The creation of the Angular component requires you to follow these steps

1. Create the Component file
2. Import the required external Classes/Functions
3. Create the Component class and export it
4. Add @Component decorator
5. Add metadata to @Component decorator
6. Create the Template
7. Create the CSS Styles
8. Register the Component in Angular Module

**app.component.ts**

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

@Component({

**selector: 'app-root',**

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

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

})

export class AppComponent {

title = 'app';

}

**Angular Module :**

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

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

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

**@NgModule**({

declarations: [

**AppComponent**

],

imports: [

BrowserModule

],

providers: [],

**bootstrap: [AppComponent]**

})

export class **AppModule** { }

We use @NgModule class decorator to define a Module and provide metadata about the Modules.

We add all the components, pipes and directives that are part of this module to the declarations array. We add all the other modules that are used by this module to imports array. We include the services in the providers’ array.

The Component that angular should load, when the app.module is loaded is assigned to the bootstrap property.

We want appComponent to be loaded when Angular starts, thus we assign it to bootstrap property

bootstrap: [AppComponent]

**Creating the inline Template & StyleUrls :** In the above example, we have used the external template & Styles.

We can also specify the Template, Styles inline using the template and styles property of @Component metadata as shown below.

In the case of a Multi-line template, you can use BackTicks ( ` ) to enclose the template string.

**The component selector :** The Angular renders the components view in the DOM inside the CSS selector, that we defined in the Component decorator

@Component({

selector: 'app-root',

The selector <app-root></app-root> is in the index.html (under src folder)

 <body>

    <app-root></app-root>

  </body>

When we build Angular Components, we are actually building new HTML elements. We specify the name of the HTML element in the selector property of the component metadata.  And then we use it in our HTML.

The Angular, when instantiating the component, searches for the selector in the HTML file and renders the Template associated with the component.

# Data Binding in Angular :

1. Data binding is a technique, where the data stays in sync between the component and the view.
2. Whenever the user updates the data in the view, Angular updates the component. When the component gets new data, the Angular updates the view.
3. We use techniques like [Interpolation](https://www.tektutorialshub.com/angular/interpolation-in-angular/), [Property Binding](https://www.tektutorialshub.com/angular/property-binding-in-angular/), [Event Binding](https://www.tektutorialshub.com/angular/event-binding-in-angular/) & [Two Way Binding](https://www.tektutorialshub.com/angular/ngmodel-two-way-data-binding-in-angular/) to bind data.
4. use the **[ngModel](https://www.tektutorialshub.com/angular/ngmodel-two-way-data-binding-in-angular/) directive** to achieve the two-way binding in [Angular Forms](https://www.tektutorialshub.com/angular/angular-forms-fundamentals/).

Data Binding in Angular : The data binding in Angular can be broadly classified into two groups. :-

* One way binding
* 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 :** **[Interpolation](https://www.tektutorialshub.com/angular/interpolation-in-angular/)** allows us to include expressions as part of any string literal, which we use in our HTML. The angular evaluates the expressions into a string and replaces it in the original string and updates the view. You can use interpolation wherever you use a string literal in the view

The Angular uses the **{{ }} (double curly braces)** in the template to denote the interpolation. The syntax is as shown below :

**{{ 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”**

}

Run the app and you will see Welcome, Sachin Tendulkar in the output. The Angular replaces both {{firstName}} & {{lastName}} with the values of firstName & lastName variable from the component.

Also, whenever the values of firstName & lastName change, Angular updates the view. But not the other way around.

**Property binding :**

1. The [Property binding](https://www.tektutorialshub.com/angular/property-binding-in-angular/) allows us to bind HTML element property to a property in the component.
2. Whenever the value of the component changes, the Angular updates the element property in the View.
3. You can set the properties such as class, href, src, textContent, etc using property binding.
4. You can also use it to set the properties of custom components or directives (properties decorated with @Input).

The Property Binding uses the following Syntax :

**[binding-target]=”binding-source”**

The binding-target (or target property) is enclosed in a square bracket []. It should match the name of the property of the enclosing element.

Binding-source is enclosed in quotes and we assign it to the binding-target. The Binding source must be a template expression. It can be property in the component, method in component, a template reference variable or an expression containing all of them.

Whenever the value of Binding-source changes, the view is updated by the Angular.

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

<button [disabled]="**isDisabled**">I am disabled</button>

export **class** AppComponent {

  title="Angular Property Binding Example"

*//Example 1*

  isDisabled= **true**;

 }

The title property of the component class is bound to the innerText property of the h1 tag. Disabled Property of the button is bound to the isDisabled Property of the component

Whenever we modify the title or isDisabled in the component, the Angular automatically updates the view.

The property binding has special syntaxes for setting the class & styles. Also, both interpolation & property binding does not set the attributes of the HTML elements. Hence we have an attribute binding to such situations.

##### **Class Binding :** You can set the class in the following ways. Click on the links to find out more

* ClassName Property binding
* Set the Class attribute with class binding
* ngClass directive

##### **Style Binding** :You can set the class in the following ways. Click on the links to find out more

* [Style Binding](https://www.tektutorialshub.com/angular/angular-style-binding/)
* [ngStyle directive](https://www.tektutorialshub.com/angular/angular-ngstyle-directive/)

##### **Attribute binding :**Sometimes there is no HTML element property to bind to. The examples are [aria](https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA) (accessibility) Attributes & [SVG](https://developer.mozilla.org/en-US/docs/Web/SVG). In such cases, you can make use of attribute binding

The attribute syntax starts with attr followed by a dot and then the name of the attribute as shown below :

<button [attr.aria-label]="closeLabel" (onclick)="closeMe()">X</button>

**Read more :**

1. **[ngClass Directive](https://www.tektutorialshub.com/angular/angular-ngclass-directive/)**
2. **[Style Binding](https://www.tektutorialshub.com/angular/angular-style-binding/)**
3. **[ngStyle directive](https://www.tektutorialshub.com/angular/angular-ngstyle-directive/)**

status:**string**='error';

<button [style.color]="status=='error' ? 'red': 'blue'">Button 1</button>

getColor() {

**return** 'yellow';

}

<button [style.color]="getColor()">Button 2</button>

<p [style.color]="getColor()"

   [style.font-size.px]="'20'"

   [style.background-color]="status=='error' ? 'red': 'blue'">

   paragraph with multiple styles

</p>

**The style property name can be written in either dash-case (font-size), as shown in above example, or camelCase (fontSize)**

**From View to Component :**

**Event Binding :**Event binding allows us to bind events such as keystroke, clicks, hover, touch, etc to a method in component. It is one way from view to component.

**For Example,** when the user changes a input in a text box, we can update the model in the component, run some validations, etc. When the user submits the button, we can then save the model to the backend server.

Angular uses the following syntax for event binding

<target-event)="TemplateStatement"

Angular event binding syntax consists of a target event name within parentheses on the left of an equal sign, and a quoted template statement on the right.

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

The above example, binds the click event of a button to a onSave() method in the component class. Whenever user clicks on the button, the Angular invokes the onSave() method.

**Two Way binding :** Two-way binding means that changes made to our model in the component are propagated to the view and that any changes made in the view are immediately updated in the underlying component

Two-way binding is useful in data entry forms. Whenever a user makes changes to a form field, we would like to update our model. Similarly, when we update the model with new data, we would like to update the view as well

The two-way binding uses the special syntax known as a banana in a box [()]

**<someElement [(someProperty)]="value"></someElement>.**

The above syntax sets up both property binding & event binding. But to make use of it, the property must have the change event with the name <propertyName>Change

But, angular has a special directive ngModel, which sets up the two-way binding

**ngModel :**The Angular uses the ngModel directive to achieve the two-way binding on HTML Form elements. It binds to a form element like input, select, selectarea. etc.

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

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

Then you can use it using the two-way binding syntax as shown below :

<input type="text" name="value" [(ngModel)]="value">

When you bind to a ngModel directive, behind the scene it sets up property binding & event binding. It binds to the value property of the element using property binding. It then uses the ngModelChange event to sets up the event binding to listen to the changes to the value.

### IMG_256

The two way data binding nothing but both property binding & event binding applied together. Property Binding is one way from view to component. The event binding is one way from component to view. If we combine both we will get the Two-way binding.

h2>Example 1</h2>

<input type="text" [value]="name" (input)="name=$event.target.value">

<p> You entered {{name}}</p>

<button (click)="clearName()">Clear</button>

name=""

clearName() {

**this**.name="";

}

We bind the name property to the input element ([value]="name"). We also use the event binding (input)="name=$event.target.value". It updates the name property whenever the input changes. The Angular interpolation updates the {{name}}, so we know the value of name property.

**ngModel Example**

Import FormsModule

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

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

**Template**

<h2>Example 2</h2>

<input type="text" name="value" [(ngModel)]="value">

<p> You entered {{value}}</p>

<button (click)="clearValue()">Clear</button>

Component

value="";

clearValue() {

this.value="";

}

The ngModel data property sets the element’s value property and the ngModelChange event property listens for changes to the element’s value.

Run the project and see that as you modify the name, the component class model is automatically updated.

**Custom Two-way binding :**

As we mentioned earlier the **[( )]** to work, we need to have a property with the change event as<nameofProperty>Change.

create a new component and name it as counter.component.ts. Copy the following code.

import { Component, Input, Output, EventEmitter } from '@angular/core';

@Component({

selector: 'counter',

template: `

<div>

<p>

Count: {{ count }}

<button (click)="increment()">Increment</button>

</p>

</div>

`

})

export class CounterComponent {

**@Input() count: number = 0;**

**@Output() countChange : EventEmitter<number> = new EventEmitter<number>();**

increment() {

this.count++;

this.countChange.emit(this.count);

}

}

The component has two properties one is input property count decorated with @Input(). The other in is an event (or output property), which we decorate with @Output(). We name the input property as count. Hence the output property becomes countChange

Now we can use this component and create two-way binding to the count property using the syntax [(count)].

<h2>Example 3</h2>

<counter [(count)]="count"></counter>

<p> Current Count {{count}}</p>

<button (click)="clearCount()">Clear</button>

**Directives :** The [Angular directive](https://www.tektutorialshub.com/angular/angular-directives/) helps us to manipulate the DOM. You can change the appearance, behavior, or layout of a DOM element using the directives. They help you to extend HTML. The [Angular directives](https://www.tektutorialshub.com/angular/angular-directives/) are classified into three categories based on how they behave.  They are **Component, Structural and Attribute Directives**.

The [ngFor](https://www.tektutorialshub.com/angular/angular-ngfor-directive/) is an Angular structural directive, which repeats a portion of the HTML template once per each item from an iterable list (Collection). The [ngSwitch](https://www.tektutorialshub.com/angular/angular-ngswitch-directive/) allows us to Add/Remove DOM Element. It is similar to the switch statement of Javascript.  The [ngIf](https://www.tektutorialshub.com/angular/angular-ngif-directive/) allows us to Add/Remove DOM Element.

The [ngClass](https://www.tektutorialshub.com/angular/angular-ngclass-directive/) Directive is an Angular Attribute Directive, which allows us to add or remove CSS classes to an HTML element. The [ngStyle](https://www.tektutorialshub.com/angular/angular-ngstyle-directive/) directive allows you to modify the style of an HTML element using the expression.  Using the [ngStyle](https://www.tektutorialshub.com/angular/angular-ngstyle-directive/) you can dynamically change the style of your HTML element.

[Angular Directives](https://www.tektutorialshub.com/angular/angular-directives/)

1. [ngFor](https://www.tektutorialshub.com/angular/angular-ngfor-directive/)
2. [ngSwitch](https://www.tektutorialshub.com/angular/angular-ngswitch-directive/)
3. [ngIf](https://www.tektutorialshub.com/angular/angular-ngif-directive/)
4. [ngClass](https://www.tektutorialshub.com/angular/angular-ngclass-directive/)
5. [ngStyle](https://www.tektutorialshub.com/angular/angular-ngstyle-directive/)
6. [ngFor Trackby](https://www.tektutorialshub.com/angular/angular-track-by-to-improve-ngfor-performance/)
7. [Custom Directive](https://www.tektutorialshub.com/angular/custom-directive-in-angular/)

## **Component Directive :**Components are special directives in Angular. They are the directive with a template (view).

## **Structural Directives :** Structural directives can change the DOM layout by adding and removing DOM elements. All structural Directives are preceded by Asterix symbol.

## **ngFor :T**he [ngFor](https://www.tektutorialshub.com/angular/angular-ngfor-directive/) is an Angular structural directive, which repeats a portion of the HTML template once per each item from an iterable list (Collection). The [ngFor](https://www.tektutorialshub.com/angular/angular-ngfor-directive/) is similar to [ngRepeat](https://docs.angularjs.org/api/ng/directive/ngRepeat) in AngularJS.

<tr \*ngFor="let customer of customers;">

    <td>{{customer.customerNo}}</td>

    <td>{{customer.name}}</td>

    <td>{{customer.address}}</td>

    <td>{{customer.city}}</td>

    <td>{{customer.state}}</td>

</tr>

#### **ngSwitch :**The [ngSwitch](https://www.tektutorialshub.com/angular/angular-ngswitch-directive/) directive lets you add/remove HTML elements depending on a match expression. [ngSwitch](https://www.tektutorialshub.com/angular/angular-ngswitch-directive/) directive used along with [ngSwitchCase](https://www.tektutorialshub.com/angular/angular-ngswitch-directive/" \l "ngswitchcase) and [ngSwitchDefault](https://www.tektutorialshub.com/angular/angular-ngswitch-directive/" \l "ngswitchdefault).

<div [ngSwitch]="Switch\_Expression">

    <div \*ngSwitchCase="MatchExpression1”> First Template</div>

    <div \*ngSwitchCase="MatchExpression2">Second template</div>

    <div \*ngSwitchCase="MatchExpression3">Third Template</div>

    <div \*ngSwitchCase="MatchExpression4">Third Template</div>

    <div \*ngSwitchDefault?>**Default** Template</div>

</div>

#### **ngIf :** The [ngIf](https://www.tektutorialshub.com/angular/angular-ngif-directive/) Directives is used to add or remove HTML elements based on an expression. The expression must return a boolean value. If the expression is false then the element is removed, else the element is inserted.

<div \*ngIf="condition">

**This** **is** shown **if** condition **is** **true**

</div>

## **Attribute Directives :** An Attribute or style directive can change the appearance or behavior of an element.

#### **ngModel :**The ngModel directive is used the achieve the[two-way data binding](https://www.tektutorialshub.com/angular/angular-data-binding/).

#### **ngClass :**The [ngClass](https://www.tektutorialshub.com/angular/angular-ngclass-directive/) is used to add or remove the CSS classes from an HTML element. Using the [ngClass](https://www.tektutorialshub.com/angular/angular-ngclass-directive/) one can create dynamic styles in HTML pages.

<div [ngClass]="'first second'">...</div>

#### **ngStyle :** [ngStyle](https://www.tektutorialshub.com/angular/angular-ngstyle-directive/)is used to change the multiple style properties of our HTML elements. We can also bind these properties to values that can be updated by the user or our components.

<div [ngStyle]="{'color': 'blue', 'font-size': '24px', 'font-weight': 'bold'}">

    some text

</div>

**Building Custom Directives :**You can also build custom directives in Angular. The Process is to create a JavaScript class and apply the @Directive attribute to that class. You can write the desired behavior in the class.

**How to Create Custom Directive in Angular 9?**

**Step 1: Create Custom Directive :** After creating successfully app, we need to create directive using angular cli command. Command is  **ng generate directive btn**

now they created btn.directive.ts and in this file we will write code for custom directive. we will import ElementRef, Renderer2 and HostListener. we will also write mouseenter and mouseleave event.

**src/app/btn.directive.ts**

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

**@Directive({**

selector: '[appBtn]'

})

export class BtnDirective {

constructor( private elementRef: ElementRef, private renderer: Renderer2 ) {

this.setFontColor('red')

}

setFontColor(color: string) {

this.renderer.setStyle(

this.elementRef.nativeElement, 'color', color )

}

@HostListener('mouseenter') onMouseEnter() {

this.setFontColor('blue')

}

@HostListener('mouseleave') onMouseLeave() {

this.setFontColor('red')

}

}

**Step 2: Import BtnDirective to module.ts file**

In last step, we will simply import BtnDirective to module.ts file,

import { BtnDirective } from './btn.directive';

@NgModule({

imports: [ BrowserModule, FormsModule ],

declarations: [ AppComponent, BtnDirective ],

bootstrap: [ AppComponent ]

})

export class AppModule { }

**Step 3: Update Component HTML File**

here, we have to update our app component html file, we need to add simple button with custom directive,

****src/app/app.component.html****

<button appBtn>My Button</button>

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

Angular Pipes takes data as input and formats or transform the data to display in the template. We use them to change the appearance of the data before presenting it to the user. The most common [use case of pipes is displaying the dates](https://www.tektutorialshub.com/angular/formatting-dates-with-angular-date-pipe/) in the correct format as per the user’s locale.

Angular Pipes Syntax

The syntax of the pipe is as follows

**Expression | pipeOperator[:pipeArguments]**

Where

**Expression:** is the expression, which you want to transform

**| :** is the Pipe Character

**pipeOperator :** name of the Pipe

**pipeArguments:** arguments to the Pipe

Angular built in date pipe to transform the date.

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

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

@Component({

    selector: 'app-root',

    templateUrl: `<p> Unformatted date : {{toDate }} </p>

                  <p> Formatted date : {{toDate | date}} </p>`

})

export **class** AppComponent

{

    title: **string** = 'pipe Example' ;

    toDate: Date = **new** Date();

}

**Chaining Pipes :** Pipes can be chained together to make use of multiple pipes in one expression. For example in the following code, the toDate is passed to the Date Pipe. The output of the Date pipe is then passed to the uppercase pipe.

toDate | date | uppercase

**DatePipe :** The Date pipe formats the date according to locale rules. The syntax of the date pipe is as shown below

date\_expression | date[:format]

Where

date\_expression is a date object or a number

date is the name of the pipe

format is the date and time format string which indicates the format in which date/time components are displayed.

**Example of Datepipe :**

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

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

@Component({

    selector: 'app-root',

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

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

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

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

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

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

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

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

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

})

export **class** AppComponent{

    title: **string** = 'Angular pipes Example' ;

    toDate: Date = **new** Date();

}

**UpperCasePipe & LowerCasePipe :** As the name suggests, these pipes transform the string to Uppercase or lowercase.

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

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

@Component({

    selector: 'app-root',

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

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

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

})

export **class** AppComponent{

    title: **string** = 'Angular pipes Example' ;

    msg: **string**= 'Welcome to Angular';

}

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

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

Where

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

**slice**is the name of the pipe

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

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

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

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

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

@Component({

    selector: 'app-root',

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

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

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

})

export **class** AppComponent

{

    title: **string** = 'Angular pipes Example' ;

    msg: **string**= 'Welcome to Angular ';

}

**DecimalPipe / NumberPipe :** The Decimal Pipe is used to Format a number as Text. This pipe will format the number according to locale rules.

Syntax

**number\_expression | number[:digitInfo]**

Where

number\_expression is the number you want to format

number is the name of the pipe

digitInfo is a string which has the following format

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

Where

**minIntegerDigits** is the minimum number of integer digits to use. Defaults to 1.

**minFractionDigits** is the minimum number of digits after fraction. Defaults to 0.

**maxFractionDigits** is the maximum number of digits after fraction. Defaults to 3.

**PercentePipe :**Formats the given number as a percentage according to locale rules.

### **CurrencyPipe :** Formats a number as currency using locale rules.

# How to Create Custom Pipe in Angular : The [Pipes](https://www.tektutorialshub.com/angular/angular-pipes/) are a great way to transform the appearance of elements in the template. The [Angular](https://www.tektutorialshub.com/angular-tutorial/)comes with some great built-in pipes like Date pipe, Currency pipe, and Number pipe, etc.

To create a custom pipe, first we need to create a pipe class. The pipe class must implement the PipeTransform interface. We also decorate it with @pipe decorator. Give a name to the pipe under name metadata of the @pipe decorator. Finally, we create the transform method, which transforms given value to the desired output.

To create a Custom Pipe, first, You need to follow these steps

1. Create a pipe class
2. Decorate the class with @pipe decorator.
3. Give a name to the pipe in the name meta data of the @pipe decorator. We will use this name in the template.
4. The pipe class must implement the PipeTransform interface. The interfaces contain only one method transform.
5. The first parameter to the transform method is the value to be transferred. The transform method must transform the value and return the result. You can add any number of additional arguments to the transform method.
6. Declare the pipe class in the Angular Module (app.module.ts)
7. Use the custom pipe just as you use other pipes.

## **Temparature Convertor Custom Pipe Example :**Create a new file temp-convertor.pipe.ts. Under the folder src/app. Copy the following code and paste it.

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

@pipe({

    name: 'tempConverter'

})

export **class** TempConverterPipe implements PipeTransform {

    transform(value: number, unit: **string**) {

**if**(value && !isNaN(value)) {

            if (unit === 'C') {

                var temperature = (value - 32) /1.8 ;

**return** temperature.toFixed(2);

            } **else** **if** (unit === 'F'){

**var** temperature = (value \* 1.8 ) + 32

**return** temperature.toFixed(2);

            }

        }

**return**;

    }

}

The PipeTransform interface defines only one method transform. The interface definition is as follows.

interface PipeTransform {

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

}

**Declare the Pipe :** Before using our pipe, we need to tell our component, where to find it. This is done by first by importing it and then including it in declarations array of the AppModule.

**Using the Custom Pipe :** The custom pipes are used in the same as the Angular built-in pipes are used. Add the following HTML code to your app.component.html file

<div **class**="row">

      <h3>Fahrenheit to Celsius </h3>    </div>

    <div **class**="row">

      <p> Fahrenheit : <input type="text" [(ngModel)]="Fahrenheit" />

      Celsius : {{Fahrenheit | tempConverter:'C'}} </p>

    </div>

    <div **class**="row">

      <h3>Celsius to Fahrenheit </h3>    </div>

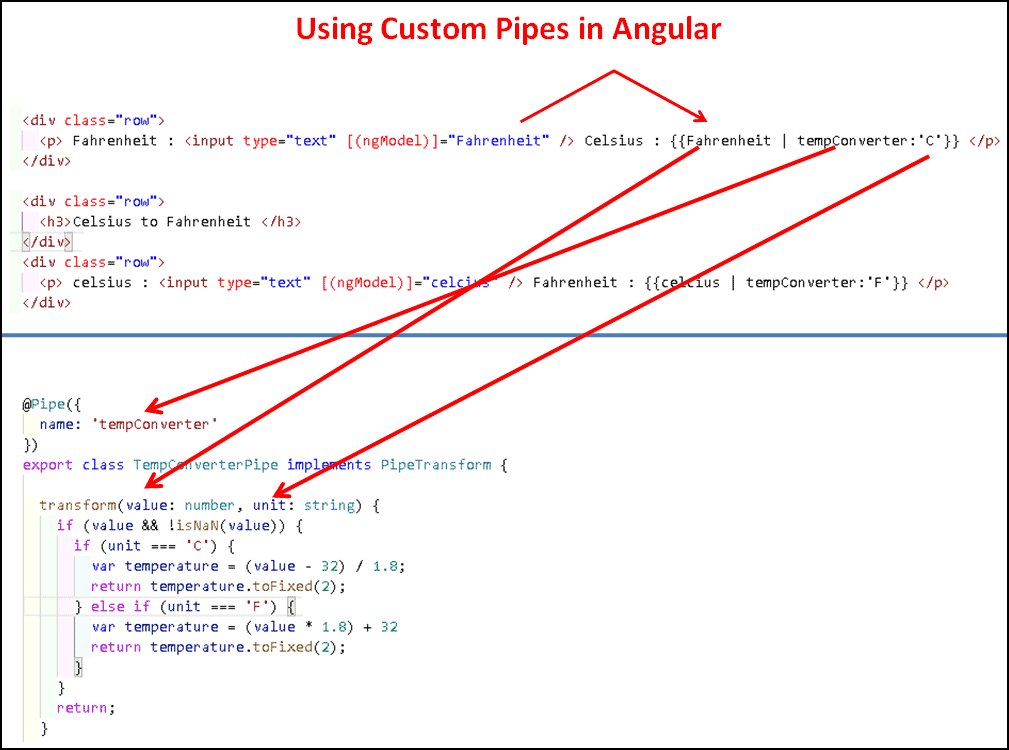
    <div **class**="row">

      <p> celsius : <input type="text" [(ngModel)]="celcius" />

       Fahrenheit : {{celcius | tempConverter:'F'}} </p>

    </div>

We use our pipe as follows. Fahrenheit is sent to the tempConverter as the first argument value. We use the | to indicate that the tempConverter is a pipe to angular. The C after the colon is the first argument. You can pass more than argument to the pipe by separating each argument by a : colon.



**Child/Nested Components in Angular :** It creates AppComponent, which is the root component of our application. The AppComponent is [bootstrapped](https://www.tektutorialshub.com/angular/angular-bootstrapping-application/) in the AppModule  and loaded in the index.html file using the selector <app-root>Loading...</app-root>.

**What is a Child/Nested Component :** The Angular follows component-based Architecture, where each component manages a specific task or workflow. Each component is an independent block of the reusable unit.

In real life, angular applications will contain many components. The task of the root component is to just host these child components. These child components, in turn, can host the more child components creating a Tree-like structure called Component Tree.

ng new childComponent

Applies to: Angular 5 to the latest edition i.e. Angular 8, Angular 9. Angular 10, Angular 11

Tell angular where to display the component : Finally, we need to inform the Angular, where to display the child Component

We want our child Component as the child of the AppComponent. Open the app.component.html and add the following template

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

<customer-list></customer-list>

The @Component decorator of the CustomerListComponent , we used the customer-list as the selectorin the metadata for the component. This CSS selector name must match the element tag that specified within the parent component’s template.

**What is parent and child component in angular?**

**Components** Relationship : -- **Angular** is built up of many **components**. There is a root **component** that serves as the **parent component**, every other **components** can be called a **Child Component** to the root **component** since the root **component** is the **Parent Component**.

**Component Life Cycle Hook :** complete list of life cycle hooks, which angular invokes during the component life cycle.

* ngOnChanges
* ngOnInit
* ngDoCheck
* ngAfterContentInit
* ngAfterContentChecked
* ngAfterViewInit
* ngAfterViewChecked
* ngOnDestroy

**Constructor**

Life Cycle of a component begins, when Angular creates the component class. First method that gets invoked is class Constructor.

Constructor is neither a life cycle hook nor it is specific to Angular.  It is a Javascript feature. It is a method which is invoked, when a class is created.

Angular makes use of a constructor to [inject dependencies](https://www.tektutorialshub.com/angular/angular-dependency-injection/).

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

@Component({

  selector: 'app-root',

  template: `

      <h2>Life Cycle Hook</h2>` ,

  styleUrls: ['./app.component.css']

})

export **class** AppComponent implements OnInit {

  constructor() {

    console.log("AppComponent:Constructor");

  }

  ngOnInit() {

    console.log("AppComponent:OnInit");

  }

}

AppComponent:Constructor

AppComponent:OnInit

The Angular executes the hooks in the following order

**On Component Creation**

1. OnChanges
2. OnInit
3. DoCheck
4. AfterContentInit
5. AfterContentChecked
6. AfterViewInit
7. AfterViewChecked

When the Component with Child Component is created

1. OnChanges
2. OnInit
3. DoCheck
4. AfterContentInit
5. AfterContentChecked
   1. Child Component -> OnChanges
   2. Child Component -> OnInit
   3. Child Component -> DoCheck
   4. Child Component -> AfterContentInit
   5. Child Component -> AfterContentChecked
   6. Child Component -> AfterViewInit
   7. Child Component -> AfterViewChecked
6. AfterViewInit
7. AfterViewChecked

After The Component is Created

1. OnChanges
2. DoCheck
3. AfterContentChecked
4. AfterViewChecked

[@Input](https://www.tektutorialshub.com/angular/angular-input-output-eventemitter/) decorator marks the customer as input property. It will receive the data from the parent.

### **ngOnChanges :** ngOnChanges triggers following the modification of @Input bound class members. With this update, ngOnChanges immediately fires. It also fires upon initialization of input data.

**ngOnInit :**ngOnInit fires once upon initialization of a component’s input-bound (@Input) properties.ngOnInit is a one-and-done hook. Initialization is its only concern.

**Dependency Injection (DI)** is a technique in which provide an instance of an object to another object, which depends on it. This is technique is also known as “Inversion of Control” (IoC).

export **class** AppComponent {

   products:Product[];

   constructor(private productService:ProductService) {

   }

   getProducts() {

**this**.products=**this**.productService.getProducts();

   }

}

**Parts of Angular Dependency Injection Framework  :**There are five main parts of the Angular Dependency injection Framework.

**Consumer :**The Component that needs the Dependency. In the above example, the AppComponent is the Consumer

**Dependency :**The Service that is being injected. In the above example the ProductService is the Dependency

**DI Token :**The DI Token uniquely identifies a Dependency. We use DI Token when we register dependency

**Provider :**The [Providers](https://www.tektutorialshub.com/angular/angular-providers/) Maintains the list of Dependencies along with their Tokens. The DI Token is used to identify the Dependency.

**Injector :**[Injector](https://www.tektutorialshub.com/angular/angular-injector-injectable-inject/) holds the Providers and is responsible for resolving the dependencies and injecting the instance of the Dependency to the Consumer

**How Dependency Injection works in Angular :**The dependencies are registered with the Provider. This is done in the Providers metadata of the Injector.

[Angular Provides](https://www.tektutorialshub.com/angular/angular-providers/) an instance of Injector & Provider to every Consumer.

Consumer when instantiated, It declares the Dependencies it needs in its constructor.

Injector reads the Dependencies from the constructor of the Consumer and looks for the dependency in the provider. The Provider provides the instance and injector, then injects it into the consumer. If the instance of the Dependency is already exists, then it is reused making the dependency singleton.

**NOTE :**

* The services injected at the module level are app-scoped, which means that they can be accessed from every component/service within the app.Any service provided in the Child Module is available in the entire application.
* The services is provided in a lazy module are module scoped and available only to the lazy loaded module.
* The services provided in the Component level are available only to the Component & and to the child components.

**How an Angular App Work Behind The Scenes — The Angular Flow**

By flow, I mean how the files are called and in which sequence of files the app gets executed when we are developing it.

1. **ANGULAR.JSON File :** ANGULAR.JSON is the file which has various properties and configuration of your Angular project. This is the file which is first referred by the builder to all the paths and configurations and to check which is the main file.

Inside the angular.json file of this project, under the ****build**** section, you can see the ****options****object as follows:

"options":{   
"outputPath":"dist/hello-world",  
"index":"src/index.html",  
****"main":"src/main.ts", // THIS LINE****"polyfills":"src/polyfills.ts",  
"tsConfig":"src/tsconfig.app.json",  
"assets":[   
"src/favicon.ico",  
"src/assets"  
],  
"styles":[   
"node\_modules/bootstrap/dist/css/bootstrap.min.css",  
"src/styles.css"  
],  
"scripts":[],  
"es5BrowserSupport":true  
}

It has a reference to the ****main.ts**** file which tells the builder to start the app from there.

1. **MAIN.TS :** This file acts as the entry point of the application. This entry point is defined in the internals of webpack that is used by Angular to support the modular functionality.

The path/name of the main file can be changed but it should also be changed in angular.json file.

Main.ts helps in creating the browser environment for the application to run.

This is done by:

**import { platformBrowserDynamic } from ‘@angular/platform-browser-dynamic’;**

After this, main.ts file calls the function ****bootstrapModule(AppModule)**** which tells the builder to bootstrap the app.

**platformBrowserDynamic().bootstrapModule(AppModule)**

1. **APP.MODULE.TS :** From the main.ts file, it is very clear that we are bootstrapping the app with AppModule.

This is the module, created with the ****@NgModule decorator,****which has declarations of all the components we are creating within the app module so that angular is aware of them. Here, we also have imports array where we can import other modules and use in our app.

import { BrowserModule } from '@angular/platform-browser';  
import { NgModule } from '@angular/core';  
import { FormsModule } from '@angular/forms';  
import { AppComponent } from './app.component';  
import { TestComponent } from './test/test.component'

**@NgModule**({  
 declarations: [  
 AppComponent,  
 TestComponent  
 ], imports: [  
 BrowserModule,  
 FormsModule  
 ],  
 providers: [],  
  ****bootstrap: [AppComponent]****  
})

export class AppModule { }

example of app.module.ts file with a test component declared and two modules imported.

1. **APP.COMPONENT.TS :**From the app.module.ts file above, we can clearly see that the module asks to bootstrap the app component. This app component is in app.component.ts file.

This is the file which interacts with the html of the webpage and serves it with the data. The component is made by using @Component decorator which is imported from @angular/core.

The component has a selector, which is like a custom html tag which we can use to call that component. It then has template or templateUrl which contains the html of the page to be displayed. It also has the styleUrls array where component specific style sheets can be placed. This is how a component file looks:

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

@Component({  
 selector: 'app-root',  
 templateUrl: './app.component.html',  
 styleUrls: ['./app.component.css']  
})

export class AppComponent {  
 title = 'hello-world';  
}

By this time, compiler has all the details about the components of the app and now they are ready to be used.

**5. INDEX.HTML :** Now, since angular is well aware of the modules, components, styles, scripts etc. which are required to display the page, it’s show time!

Here, the index.html file is called. It is found in the src folder of the app. Compiler dynamically adds all the javascript files at the end of this file. Since all the components are now known,

the html file calls the root component that is **app-root**. The root component is defined in app.components.ts which targets app.component.html. This is how index.html file looks like in the coding environment:

<!doctype html>  
<html lang="en">  
<head>  
<meta charset="utf-8">  
<title>My Hello World App!</title>  
<base href="/">  
<meta name="viewport" content="width=device-width, initial-scale=1">  
<link rel="icon" type="image/x-icon" href="favicon.ico">  
</head>  
<body>  
****<app-root></app-root>****  
</body>  
</html>

In the body tag, you can see that a html like element ****<app-root></app-root>**** is present. Well, this is our component selector for the AppComponent which is defined in app.component.ts file. It asks angular to load that component.

**6 .APP.COMPONENT.HTML :** This is the file which contains all the html elements and their binding which are to be displayed when the app loads. Contents of this file are the first things to be displayed.

**Angular 6 Features :**

The major highlights of Angular 6 include the

Angular Command Line Interface (CLI),

The Component Development KIT (CDK),

Angular Material Package update and with new TypeScript version 2.7, it is much easier to code with conditional type declarations, default declarations, and strict class initialization.

#### **Angular Element**

Creating a widget or component that can be included in any existing web page was made possible with Angular Elements.  
The Angular Elements package provided the ability to create an Angular component and then publish that component as a web component which can be used in any HTML page

#### **Service Worker**

Service workers were first introduced in Angular 5. It is the scripts which run in the web browser and manage to cache an application. Service workers come with some bug fixes, including some new functionalities as there was no straight way for deactivating the service worker and to overcome this, Angular 6 came up with a new script file safety-worker.js in production bundle. This script file helps them to unregister an existing service worker.

#### **Internationalization (i18n)**

One of the major changes introduced to Angular 6 is internationalization (i18n) feature with runtime rendering so that there is no requirement to build one application per locale.

#### **Bazel Compiler**

Bazel optimizes your Angular compilation, it only rebuilds what is necessary to build as it does not make any sense to rebuild the entire application for every little change with this so we can archive fast and incremental builds.

#### **ng-add**

ng add helps you install and download new packages in your angular apps.

#### **ng-update**

ng update is used to update and upgrade your packages it will help you to upgrade from Angular 5 to Angular 6, or any other package in your Angular app.

#### **Tree Shaking**

Tree shaking is a build optimization step which tries to ensure any unused code does not get used in our final bundle which helps you to make your app smaller. It uses new injectable services where we can register a provider directly inside the @Injectable() decorator, using the new providedIn attribute.

#### Some Major updates in Angular 6

* Typescript 2.7.x supports
* Improved decorator error messages
* Fix platform-detection example for Universal
* Added to supports of Native-Element
* Added Optional generic type for ElementRef
* Updates on NgModelChange
* Add type and hooks to directive def
* Enable size tracking of a minimal CLI render3 application
* Add canonical view query
* <ng-template> now becomes <template>
* Long-term support(LTE) added to Angular from v4

### **Angular 7 Features**

Google has released Angular version 7 in Oct 2018 with a lot of optimum features and significant changes like Angular Material, CLI prompts, Scrolling, Drag, and Virtual and Drop & Component Dev Kit (CDK)

#### **CLI prompts**

In Angular 7, the command-line interface (CLI) prompts have been updated to v7.0.2, When the user executes common commands like ng add @angular/material or ng new it will automatically prompt users commands like ng add @angular/material help you discover built-in features like routing or SCSS support.

#### **New ng-compiler**

Angular 7 added a new compiler called the Angular Compatibility Compiler (ngcc). Just like the name suggests, the angular compiler offers an 8-phase rotating ahead-of-time compilation(AOT) and most of the angular applications noticed a massive reduction (95-99%) in bundle sizes.

#### **Angular Material CDK (Component Dev Kit)**

* Virtual scrolling  
  The scrolling package enables loads and unloads items from the DOM depending upon visible parts of lists, resulting into a much faster experience for users having huge scrollable lists.
* Drag & drop  
  Now you can re-order the list just by dragging and dropping with new @angular/cdk/drag-drop module which provides free dragging, sorting within a list, transferring items between lists, animations and much more.

#### **Application performance**

Many developers include the reflect-metadata polyfill in the production, so they decided to fix this part by automatically removing it from your polyfills.ts file and to speed up the performance new applications will warn when the initial bundle is more than 2MB and will error at 5MB which user can modify it in angular.json file.

#### **Angular Do-Bootstrap**

Angular 7 added a new life-cycle hook (ngDoBootstrap) and interface (DoBootstrap), It's used for bootstrapping modules that need to bootstrap a component.

#### **Router**

In Angular 7, If you try to trigger navigation outside of the Angular zone it logs a warning (only in development mode). Also, adds navigation execution context info to activation hooks.

#### Some Major updates in Angular 7

* Angular 7 now supporting to TypeScript 3.1
* Added a new ability to recover from malformed URLs
* Downloadable console for starting and running Angular projects on your local machine
* compiler-cli: update tsickle to 0.29.x
* Export defaultKeyValueDiffers to private API

#### Major features included in Angular 8

* AngularJS API Migration Improvements with $location service
* Updated Typescript to 3.4.x
* @angular/platform-webworker and@angular/platform-webworker-dynamic both the packages are deprecated
* @angular/http removed from the list of packages
* ng-build, ng-test, and ng-run are equipped to be extended by 3rd-party libraries and tool.
* Angular router backward compatibility
* Dart-sass for Sass files
* The ViewChild and ContentChild decorators now must have a new option called static.

### **Angular 9 Features**

Angular 9 was released on the 7th of Feb 2020, with some exciting new features. Along with these existing features have also been modified. The highlight of this version is the default IVY compiler which forms the core structure of the entire framework.

#### **IVY as a Default Compiler**

The default IVY compiler reduces the bundle size by 25-40 percent which enables better performance. This enables the developers to decrease the size of files making it user-friendly. IVY checks the bindings within the templates of any application and then reports any kind of issues it finds. this helps in detecting any of bugs earlier in the development process. It then compiles the codebases with tools and techniques to debug its applications. In angular 9, the IVY manages the style binding without being dependent on any timings. Angular 9 maintains most of the drawbacks without losing performance.

#### **More reliable ng-update**

The ng update is the basic update to the current CLI framework takes advantage of newer updates automatically. This will also enable to give updates and information about any kind of migration.

#### **API Extractor Updates**

Angular depends on various services and libraries that are difficult to track. API landscape of these libraries helps find the missing updates so that they are communicated easily.

#### **New options for 'providedIn'**

This angular 9 feature provides us with certain options while creating services in Angular. for e. g

* platform— The providedIn: 'platform' makes the service available in a special single platform that is shared by all applications on the page.
* any— It provides a unique instance in every module, instilling the token.

#### **IDE and Language Service Improvements**

The Angular language service extension includes a set of useful extensions in order to build up the development experience with Angular. The extensions include Angular snippets, ESLint, and debugging extensions. Along with these performance and stability issues have also been fixed.

#### **Updated to Typescript 3.7**

The typescript 3.7 was released in November 2019 and it gave a major hit to the angular 9. The angular 9 is updated to work with the Nullish coalescing and optional chaining feature of the Typescript 3.7. This feature helps to stay in sync with the environment.

#### **Component Harness**

Angular 9 provides alternative ways to test components to make sure that the unit tests are correctly audited and less frail. Angular finds the component on its own.the compile rand runtime have been updated in angular 9, so there is no longer a need to identify this in the entryComponents array.

#### **The Phantom Template Variable Menace**

In the previous versions, phantom variables were not cited in the template’s related component. Building the phantom variables can usually threaten the applications. With this version, a compiler error shows up, when a template variable is created that has not been characterized in a component.

#### **Major updates in Angular 9**

* Selector-less directives were already supported in the old version but were missing in the Ivy preview in Angular 8. This has now been added to Angular 9.
* The AOT builds will be noticeably faster ensuring a significant change in the compiler's performance.
* With IVY we see an improvement in the build error. It gives less time for the builds to complete enabling all the error messages easier to read.
* The Angular framework in the new version uses unique techniques and tools to debug its applications by themselves.
* The deprecated versioned files option for service workers in the service worker asset group config has been removed.
* The ViewEngine in this version translates the templates and components into regular HTML and JavaScript for the browser can interpret and display them.

| **Angular 6** | **Angular 7** | **Angular 8** | **Angular 9** |
| --- | --- | --- | --- |
| Angular Element | CLI Prompts | Ivy Engine | Default Ivy in v9 |
| Service worker | Virtual Scrolling | Web Workers | Phantom Template Variable Menace |
| Internationalization (i18n) | Drag and Drop | Lazy Loading | Dependency Injection Changes in Core |
| Bazel Compiler | Bundle Budget | Improvement in ng-upgrade | Service Worker Updates |
| ng-add / ng-update | Angular Compiler | Support for Node 10 | i18n Improvements |
| ng-update | Angular Do-Bootstrap | CLI workflow improvements | More reliable ng update |
| ngModelChange | Better Error Handling | Upgrading Angular Material | API Extractor Updates |
| TypeScript 2.7 support | TypeScript 3.1 support | TypeScript 3.4 support | Typescript 3.7 support |
| Improved decorator error messages | New ng-compiler | Differential Loading | Component Harness |
| <ng-template> updated to <template> | Native Script | Improved Web Worker Bundling | ModuleWithProviders Support |

**Angular Pass data parent to child component :** In Angular, the Parent Component can communicate with the child component by setting its Property.

The Child Component does this by using the ****@Input decorator.****

In the Child Component

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

In the Parent Component

* Bind the Child component property in the Parent Component when instantiating the Child

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

@Component({

    selector: 'child-component',

    template: `<h2>Child Component</h2>

               current count **is** {{ count }}

    `

})

export **class** ChildComponent {

    @Input() count: number;

}

**Bind to Child Property in Parent Component :**

<child-component [count]=Counter></child-component>

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

@Component({

  selector: 'app-root',

  template: `

        <button (click)="increment()">Increment</button>

        <button (click)="decrement()">decrement</button>

        <child-component [**count**]=Counter></child-component>` ,

  styleUrls: ['./app.component.css']

})

export **class** AppComponent {

  title = 'Component Interaction';

  Counter = 5;

  increment() {

**this**.Counter++;

  }

  decrement() {

**this**.Counter--;

  }

}

Here, we are using count property, which is a property of the child Component inside the square bracket. We bind it to Counter property of the Parent Component.

Remember square bracket represents the [Property Binding in Angular](https://www.tektutorialshub.com/angular/angular-data-binding/" \l "property-binding).

**How to Pass data to parent component using @Output**

In the child component

1. Declare a property of type EventEmitter and instantiate it
2. Mark it with a @Output annotation
3. Raise the event passing it with the desired data

In the Parent Component

1. Bind to the Child Component using [Event Binding](https://www.tektutorialshub.com/angular/angular-data-binding/" \l "Event-Binding) and listen to the child events
2. Define the event handler function

import { Component, Input, Output, EventEmitter  } from '@angular/core';

@Component({

    selector: 'child-component',

    template: `<h2>**Child** Component</h2>

               current count **is** {{ count }}

    `

})

export **class** ChildComponent {

    @Input() count: number;

@Output() countChanged: EventEmitter<number> =   **new** EventEmitter();

}

**Parent Component :**

<child-component [count]=Counter (countChanged)="countChangedHandler($event)"></child-component>

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

@Component({

  selector: 'app-root',

  template: `

        <h1>Welcome to {{title}}!</h1>

        <p> current count **is** {{ClickCounter}} </p>

        <child-component [count]=Counter (countChanged)="countChangedHandler($event)"></child-component>` ,

  styleUrls: ['./app.component.css']

})

export **class** AppComponent {

  title = 'Component Interaction';

  Counter = 5;

  countChangedHandler(count: number) {

**this**.Counter = count;

    console.log(count);

  }

}

Or

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

We now update the code from previous section

**Child Component :** There is no change in the child component

**Parent Component :** In parent component, we need to import the viewChild annotation. We also need to import the child component.

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

import { ChildComponent } from './child.component';

@Component({

  selector: 'app-root',

  template: `

        <h1>{{title}}</h1>

        <p> current count **is** {{child.count}} </p>

        <child-component></child-component>` ,

  styleUrls: ['./app.component.css']

})

export **class** AppComponent {

  title = 'Parent calls an @ViewChild()';

  @ViewChild(ChildComponent) child: ChildComponent;

}

**What is @ViewChild in angular?**

The @**ViewChild** decorator allows us to inject into a component class references to elements used inside its template, that's what we should use it for. Using @**ViewChild** we can easily inject components, directives or plain DOM elements.

**Angular Observable :**how to use RxJS Observables in [Angular](https://www.tektutorialshub.com/angular-tutorial/) application. When we talk about Angular Observable, we hear a lot of terms like Reactive programming, data streams, Observable, Observers, RxJS, etc.

Rx stands from Reactive programming. It is defined as programming with asynchronous data streams.

**What is RxJS :** The [RxJS](https://rxjs.dev/guide/overview) (Reactive Extensions Library for JavaScript) is a javascript library, that allows us to work with asynchronous data streams

The Angular uses the RxJS library heavily in its framework to implement Reactive Programming.

Some of the examples where reactive programming used are

* Reacting to an [HTTP request in Angular](https://www.tektutorialshub.com/angular/angular-httpclient/)
* [Value changes](https://www.tektutorialshub.com/angular/valuechanges-in-angular-forms/) / [Status Changes](https://www.tektutorialshub.com/angular/statuschanges-in-angular-forms/) in Angular Forms
* The Router and Forms modules use observables to listen for and respond to user-input events.
* You can define custom events that send observable output data from a child to a parent component.
* The HTTP module uses observables to handle AJAX requests and responses.

The RxJs has two main players

1. Observable
2. Observers ( Subscribers)

**What is an Observable in Angular :** Observable is a function that converts the ordinary stream of data into an observable stream of data.

Observable as a wrapper around the ordinary stream of data.

Observable emits the value from the stream asynchronously. It emits the complete signals when the stream completes or an error signal if the stream errors out.

call them observers or subscr

Observables are declarative. You define an observable function just like any other variable. The observable starts to emit values only when someone subscribes to it.

**Who are observers (subscribers) :** someone consumes the value emitted by the observable. Weibers.

The observers communicate with the Observable using callbacks

The observer must subscribe with the observable to receive the value from the observer. While subscribing it optionally passes the three callbacks. next(), error() & complete().

The observable invokes the next() callback whenever the value arrives in the stream. It passes the value as the argument to the next callback. If the error occurs, then the error() callback is invoked. It invokes the complete() callback when the stream completes.

RxJs library is installed automatically when you create the Angular project. Import the Observable from the rxjs library

import { Observable } from 'rxjs';

**Observable Creation :** There are few ways in which you can create observable in angular. Simplest is to use the Observable constructor.

obs = **new** Observable((observer) => {

     console.log(“Observable starts”)

     observer.next("1")

     observer.next("2")

     observer.next("3")

     observer.next("4")

     observer.next("5")

   })

**Subscribing to the observable**

We subscribe to the observable, by invoking the subscribe method on it. We can optionally, include the three callbacks next(), error() & complete() as shown below

ngOnInit() {

**this**.obs.subscribe(

      val => { console.log(val) }, *//next callback*

      error => { console.log("error") }, *//error callback*

      () => { console.log("Completed") } *//complete callback*

    )

}

**Unsubscribing from an Observable**

We need to [unsubscribe to close the observable](https://www.tektutorialshub.com/angular/unsubscribing-from-an-observable-in-angular/) when we no longer require it. If not it may lead to memory leak & Performance degradation.

To Unsubscribe from an observable, we need to call the Unsubscribe() method on the subscription. It will clean up all listeners and frees up the memory.

ngOnDestroy() {

**this**.obs.unsubscribe();

}

When we destroy the component, the observable is unsubscribed and cleaned up.

**What is Observable?**

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

The observables implement the [observer design pattern](https://en.wikipedia.org/wiki/Observer_pattern), where observables maintain a list of dependents. We call these dependents as observers. The observable notifies them automatically of any state changes, usually by calling one of their methods.

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

When an Observer subscribes to an observable, it needs to pass (optional) the three callbacks. next(),  error()  &  complete(). The observable invokes the next() callback, when it receives an value. When the observable completes it invokes the complete() callback. And when the error occurs it invokes the error() callback with details of error and subscriber finishes.

The Observables are used extensively in Angular. The new HTTPClient Module and Event system are all Observable based.

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

**What is an observable in angular?**

**Observables** provide support for passing messages between parts of your application. Observables are a technique for event handling, asynchronous programming, and handling multiple values.

an **Observable** is a stream of events or data. **Observables** are a blueprint for creating streams and plumbing them together with operators to create **observable** chains.

**RxJS** is a library that lets us create and work with **observables**. We can subscribe to an **observable** chain and get a callback every time something is pushed onto the last stream.

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

HTTP GET

HTTP Post

HTTP PUT

HTTP PATCH

HTTP DELETE

**difference between observable and promise  :**

|  |  |
| --- | --- |
| **observables** | **Promises** |
| **observables** handle a sequence of asynchronous events over a period of time. observable emit multiple values over a period of time. | **Promises** deal with one asynchronous event at a time,promise emit a single value at a time. |
| Are lazy: they’re not executed immediately until we subscribe to them using the subscribe() method. | Are not lazy: execute immediately after creation. |
| Have subscriptions that are cancellable using the unsubscribe() method, which stops the listener from receiving further values. | Are not cancellable. |
| Provide the map for forEach, filter, reduce, retry, and retryWhen operators. | Don’t provide any operations. |
| Deliver errors to the subscribers. | Push errors to the child promises. |
| const obs = new Observable((observer) ={    observer.next(10);    }) ; | const promise = new Promise(() = {    resolve(10);    }); |
| An Observable 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). Compared to a promise, an observable can be canceled. | Disadvantages of Promise:   1. User could not cancel a request to the API. 2. User could not retry a failed call. 3. As our application gets bigger, promises become hard to manage. |
| RxJS is all about unifying the ideas of promise callbacks and data flow and making them easier to work with. Observables provide operators, like map, forEach, reduce...similar to an array.  There are also powerful operators like retry(),  reply(), retryWhen(), delay(). |  |

**What services are used for**

* Features that are independent of components such a logging services
* Share logic or data across components
* Encapsulate external interactions like data access

An Angular service is simply a Javascript function. All we need to do is to create a class and add methods & properties. We can then create an instance of this class in our component and call its methods.

One of the best uses of services is to **get the data from the data source.**In real life, you would send an HTTP request to your back end API to get the data.

**Invoking the ProductService**

The Next step is to invoke the ProductService from the component.

@Component({

  selector: 'app-root',

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

})

export **class** AppComponent{

   products:Product[];

   productService;

   constructor(){

**this**.productService=**new** ProductService();

   }

   getProducts() {

**this**.products=**this**.productService.getProducts();

   }

}

### **Injecting Services into Component :**

**this**.productService=**new** ProductService();

Directly instantiating the service, as shown above, has many disadvantageous

1. The ProductService is tightly coupled to the Component. If we change the ProductService class definition, then we need to update every code where service is used
2. If we want to change ProductService with BetterProductService, then we need to search wherever the ProductService is used and manually change it
3. Makes Testing difficult. We may need to provide mockProductService for testing and use the ProductService for Production.

## **What is Angular Dependency Injection:**

Dependency Injection (DI) is a technique in which we provide an instance of an object to another object, which depends on it. This is technique is also known as “Inversion of Control” (IoC)

export **class** AppComponent {

   products:Product[];

   constructor(private productService:ProductService) {

   }

   getProducts() {

**this**.products=**this**.productService.getProducts();

   }

}

**Parts of Angular Dependency Injection Framework**

There are five main parts of the Angular Dependency injection Framework.

**Consumer :**The Component that needs the Dependency. In the above example, the AppComponent is the Consumer

**Dependency :** The Service that is being injected. In the above example the ProductService is the Dependency

**DI Token :**The DI Token uniquely identifies a Dependency. We use DI Token when we register dependency

**Provider :**The [Providers](https://www.tektutorialshub.com/angular/angular-providers/) Maintains the list of Dependencies along with their Tokens. The DI Token is used to identify the Dependency.

**Injector :** [Injector](https://www.tektutorialshub.com/angular/angular-injector-injectable-inject/) holds the Providers and is responsible for resolving the dependencies and injecting the instance of the Dependency to the Consumer

**How Dependency Injection works in Angular**

The dependencies are registered with the Provider. This is done in the Providers metadata of the Injector.

[Angular Provides](https://www.tektutorialshub.com/angular/angular-providers/) an instance of Injector & Provider to every Consumer.

Consumer when instantiated, It declares the Dependencies it needs in its constructor.

Injector reads the Dependencies from the constructor of the Consumer and looks for the dependency in the provider. The Provider provides the instance and injector, then injects it into the consumer. If the instance of the Dependency is already exists, then it is reused making the dependency singleton.

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

@Injectable()

export **class** LoggerService {

  log(message:any) {

    console.log(message);

  }

}

we need to tell angular that our component needs dependency injection. This is done by using the [@Injectable()](https://www.tektutorialshub.com/angular/angular-injector-injectable-inject/) decorator.

@Injectable() decorator is not needed, if the class already has other Angular decorators like @Component, @pipe or @directive etc. Because all these are a subtype of Injectible.

Since our AppComponent is already decorated with @Component, we do not need to decorate with the @Injectable.

@NgModule({

  declarations: [

    AppComponent

  ],

  imports: [

    BrowserModule,

    HttpModule,

    FormsModule

  ],

  providers: [ProductService,LoggerService],

  bootstrap: [AppComponent]

})

export **class** AppModule { }

Providing the service in the root module will create a single, shared instance of service and injects into any class that asks for it.

The services injected at the module level are app-scoped, which means that they can be accessed from every component/service within the app.Any service provided in the Child Module is available in the entire application.

The services is provided in a lazy module are module scoped and available only to the lazy loaded module.

The services provided in the Component level are available only to the Component & and to the child components.

**HTTP  :** The newly designed HttpClient Module allows us to query the Remote API source to get data into our Application. It requires us to Subscribe to the returned response using RxJs observables.

**HttpClient**to make HTTP requests like GET & POST, etc. to the back end server. The Angular HTTP client module is introduced in the Angular 4.3.  This new API is available in package **@angular/common/http**. It replaces the older HttpModule. The HTTP Client makes use of the RxJs Observables. The Response from the HttpClient is observable, hence it needs to be Subscribed.

The **HttpClient**is a separate model in Angular and is available under the @angular/common/http package.

The following steps show you how to use the HttpClient in an Angular app.

**Import HttpClient Module in Root Module :** We need to import it into our root module app.module. Also, we need to add it to the imports metadata array.

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

import { HttpClientModule } from '@angular/common/http';

@NgModule({

    declarations: [

        AppComponent

    ],

    imports: [

        HttpClientModule

    ],

    providers: [],

    bootstrap: [AppComponent]

})

export **class** AppModule { }

**Import Required Module in Component/Service**

Then you should import HttpClient the @angular/common/http in the component or service.

**import { HttpClient } from '@angular/common/http';**

**Inject HttpClient service :** Inject the HttpClient service in the constructor.

constructor(public http: HttpClient) {

}

**Call the HttpClient.Get method**

Use HttpClient.Get method to send an [HTTP Request](https://www.tektutorialshub.com/http/http-get-and-post-methods/). The request is sent when we Subscribe to the get() method. When the response arrives map it the desired object and display the result.

public getData() {

**this**.HttpClient.**get**<any[]>(**this**.baseUrl+'users/'+**this**.userName+'/repos')

           .subscribe(data => {

**this**.repos= data;

           },

           error => {

           }

  );

}

**What is EventEmitter in angular?**

Data flows into your component via property bindings and flows out of your component through event bindings. If you want your component to notify his parent about something you can use the Output decorator with **EventEmitter** to create a custom event.

**Angular Router :** The Router module handles the navigation & Routing in Angular. The Routing allows you to move from one part of the application to another part or one View to another View.