Complete Angular Tutorial For Beginners

Welcome to the **Angular Tutorial**. This Tutorial covers all versions of Angular Starting from Angular 2, Angular 4, Angular 5, Angular 6, Angular 7, Angular 8, Angular 9.

Table of Content	

Introduction to Angular

- **Angular 2/4/5/6.. latest version** is now known as **Angular.**
- The Angular is an open-source and helps us build dynamic & single-page applications (SPAs).

Table of Content

What is Angular?

- Is a UI framework for building mobile and desktop web applications. It is built using JavaScript.

Features of Angular

Angular is loaded with Power-packed Features. Some of the features are listed below

- Two-Way **Data Bindin**
- Powerful **Routing** Support
- Expressive HTML
- Modular by Design
- 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.
- This makes a lot of difference as your queries are quickly resolved.

Key differences between AngularJs & Angular

Support for ES6

- Angular is completely written in **Typescript** and meets the **ECMAScript 6 specification**.
- It has support for ES6 Modules, Class frameworks, Components are new controllers.
- In AngularJS we had **Controllers**. In Angular **“Controllers”** are replaced with Angular Controller
- The controllers and view in AngularJS is defined as follows.

```
//View
<body ng-controller=’appController’>
  <h1>vm.message<h1>
</body>

//Controller angular.module(‘app’).controller(‘appController’,
  message=’Hello Angular’;
}
```

In Angular, we are using Components. The simple component is shown below written using Typescript.

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

@Component({
  selector: 'app',
  template: '<h1>{{message}} </h1>'
})
export class AppComponent  {
   message: string=’Hello Angular’;
}
```

In Angular, a component represents a UI element. You can have many such components on a single web page. Each component is independent of each other and manages a section of the page. 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**

```
<ul>
  <li ng-repeat =customer in vm.customers>
    {{customer.name}}
  </li>
</ul>
<div ng-if=”vm.isVIP”>     <h3> VIP Customer </h3> </div
```

The Angular also has **directives,** but with a different syntax. It has a astrisk **(*)** **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**

Data Bindings

The powerful angular data bindings stay the same, with minor syntax changes.

Interpolation

```
1
2 //AngularJS
3 <h3> {{vm.customer.Name}}</h3>
4
5 //Angular
6 <h3> {{customer.Name}}</h3>
7
```

Note that we used controller alias VM to specify the controller instance in AngularJS. In Angular, the context is implied.

One way Binding

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

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

The Angular can bind to any property of the HTML element.

Event Binding

```
1
2 //AngularJS
3 <button ng-click=”vm.save()”>Save<button>
4
5 //Angular
6 <button (click)=”save()”>Save<button>
7
```

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

Two- way binding

```
1
2 //AngularJS
3 <input ng-model=”vm.customer.name”>
4
5 //Angular
6 <input [(ng-model)]=”customer.name”>
7
```

$scopes are out

- Angular is not using **$scope** anymore to **glue view and controlle**r.
- **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.**

Filters are renamed to Pipes

In **AngularJS**, we used Filters and as shown below

```
1
2 <td>{{vn.customer.name | uppercase}}</td>
3
```

Angular uses the same syntax but names them as **pipes**

```
1
2 <td>{{customer.name | uppercase}}</td>
3
```

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
- Angular all the above is merged into a **Service class**.

Mobile Support

AngularJS was not built with mobile support in mind. Angular designed with mobile development in mind.

Angular Components

Table of Content

What is an Angular Component ?

- Component is the main building block of an Angular Application or The Component is responsible to provide the data to the view.
- The component contains the data & user interaction logic that defines how the View looks and behaves. A view in Angular refers to a template (HTML).
- The Angular Components are plain javascript classes and defined using **@component Decorator**.
- The Angular does this by using **data binding,** to get the data from the **Component to the View**.
- This is done using the special HTML markup knows as the Angular Template Syntax. The Component can also get notified when the View Changes.

The Components consists of three main building blocks

- Template
- Class
- MetaData

Building blocks of the Angular Component

Template (View)

- Template defines the layout of the **View and defines what is rendered on the pag**e. Without the template, there is nothing for Angular to render.
- The Templates are created **using HTML**. You can add **Angular directives** and **bindings** on the template.
- There are two ways you can specify the Template in Angular.

1.Defining the Template Inline

2.Provide an external Template

Class

- The class is the code associated with Template (View).
- The Class is created with the Typescript, but you can also use javascript directly in the class. Class Contains the Properties & Methods.
- The Properties of a class can be bind to the view using Data Binding.

The simple Angular Class

```
1
2 export class AppComponent
3 {
4   title : string ="app"
5 }
6
```

Note : The Component classes in Angular are prefixed with the name "Component", To easily identify them.

Metadata

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

@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.

Important Component metadata properties

Selector

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

Providers

The Providers are **the services,** and used by **component** . The **Services provide** service to the **Components or to the other Services.**

Styles/styleUrls

- The **CSS Styles or style sheets**, that this component needs.
- **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

Creating the Component

We will not create the Angular Component, but let us see the Component creation process in detail. The Angular CLI has automatically created the root component **app.component.ts**. 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**

1. Creating the Component File

The Component app.component.ts. is already been created for us by Angular CLI under the folder src.

By Convention, the file name starts with the **feature name** (app) and then followed by the **type of class** (component). These are separated by a dot. The extension used is **ts** indicating that this is a typescript module file.

2. Import the Angular Component Object

Before we use any Angular (or external) functions or classes, we need to tell Angular how and where to find it. This is done using the Import statement. Which allows us to use the external modules in our class

To define the Component class, we need to use the @Component decorator. This function is found in the Angular core library. So we import it in our class as shown below

```
1
2 import { Component } from '@angular/core';
3
```

3. Create the Component Class and export it

The third step is to create the Component class using the export keyword. The Export keyword allows this component class to be used in other components by importing. The AppComponent class is shown below

```
export class AppComponent {
 title = 'app';
}
```

4. Add @Component decorator

The next step is to inform the Angular that this is a Component class. This is done by adding the @Component decorator to the above class. The decorator must be added immediately above the class as shown

```
@Component({
})
export class AppComponent {
 title = 'app';
}
```

5. Add meta data to @Component decorator

The next step is to add the metadata to the @component decorator. Add the following to the component metadata

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

Selector : The angular places the view (template) inside the selector app-root

TemplateUrl : In the above example, we have used an external template using templateUrl metadata. The templateUrl points to the external HTML file **app.component.html**.

StyleUrls : Defines the styles for our template. The metadata points to the external CSS file **app.component.css**.

6. Create the Template (View)

- Template is nothing but an HTML file, which component must display it to the user
- The Angular knows which template display, using the templateUrl metadata, which points to **app.component.html**.

```
<!--The content below is only a placeholder and can be replaced.-
<div style="text-align:center">
 <h1>
  Welcome to {{title}}!
 </h1>
 <img width="300"
src="
JbiAuIFNWRyBWZXJzaW9uOiA2LjAwIEJ1aWxkIDApICAtLT4NCjxzdm
ovL3d3dy53My5vcmcvMTk5OS94bGluayIgeD0iMHB4IiB5PSIwcHgi
NlPSJwcmVzZXJ2ZSI+DQo8c3R5bGUgdHlwZT0idGV4dC9jc3MiPg0K
CjxnPg0KCTxwb2x5Z29uIGNsYXNzPSJzdDAiIHBvaW50cz0iMTI1LDM
sNjMuMiAJIi8+DQoJPHBvbHlnb24gY2xhc3M9InN0MSIgcG9pbnRzPS
NjMuMiAxMjUsMzAgCSIvPg0KCTxwYXRoIGNsYXNzPSJzdDIiIGQ9Ik0x
uMUwxMjUsNTIuMUwxMjUsNTIuMQ0KCQlMMTI1LDUyLjF6IE0xNDIsL
</div>
<h2>Here are some links to help you start: </h2>
<ul>
 <li>
  <h2><a target="_blank" href="https://angular.io/tutorial">Tour
 </li>
 <li>
  <h2><a target="_blank" href="https://github.com/angular/angu
 </li>
 <li>
  <h2><a target="_blank" href="https://blog.angular.io//">Angula
 </li>
</ul>
```

By Convention, the file is named after the component file it is bound to with HTML extension.

This is a simple HTML file, except for the initial h1 tag

```
1
2 <h1>
3  Welcome to {{title}}!
4 </h1>
5
```

Note that **title *inside the double curly bracket***. When rendering the view, the Angular looks for **title Property** in our c**omponent and binds the property to our view**. This is called **data binding.**

The double curly bracket syntax is known as **interpolation,**

7. Add the Styles

The next step is to add the CSS Styles. The styleUrls metadata tells Angular, where to find the CSS File. This property points to external file **app.component.css**

By Convention, the file is named after the component file it is bound to with .css extension

Note that styleUrls metadata can accept multiple CSS Files.

8. Register the Angular Component in Angular Module

We have created the Angular Component. The next step is to register it with the Angular Module

It is a class that is adorned with @ngModule decorator

```
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.

The AppComponent imported

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

and then added to the declarations array.

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

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

```
bootstrap: [AppComponent]
```

Thats it

Finally, run the application from the command line using **ng serve** (or npm start).

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.

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

@Component({
 selector: 'app-root',
 template: '<h1> {{title}} works </h1>',
 styles: ['h1 { font-weight: bold; }']
})
export class AppComponent {
 title = 'app';
}
```

The Template can get pretty long. In the case of a Multi-line template, you can use **BackTicks** (`) to enclose the template string.

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

@Component({
 selector: 'app-root',
 template: `
     <h1> {{title}} works </h1>
     <p> a long inline template </p>
     `,
 styles: ['h1 { font-weight: bold; }']
})
export class AppComponent {
 title = 'app';
}
```

Specifying the Templates and styles inline has few disadvantageous. The Template can get pretty big and clutter your code. You will also not get the IntelliSense help while editing the template. In such a case you can write your Html template in an external file and link that file in your component code.

The component selector

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

```
1
2 @Component({
3  selector: 'app-root',
4
```

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

```
<!doctype html>
<html lang="en">
 <head>
  <metacharset="utf-8">
  <title>Angular4</title>
  <basehref="/">
  <metaname="viewport"content="width=device-width, initial-sca
  <linkrel="icon"type="image/x-icon"href="favicon.ico">
 </head>
 <body>
  <app-root></app-root>
 </body>
</html>
```

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.

That gives several options to use our component selector

Using the CSS class name

```
@Component({
 selector: '.app-root'
})
```

And in the HTML markup use the CSS class name

```
<div class="app-root"></div>
```

Using attribute name

```
1
2 @Component({ selector: '[app-root]' })
3
```

And you can now the attribute as follows

```
<div app-root></div>
```

Using attribute name and value

```
@Component({
 selector: 'div[app=components]'
})
```

now you can use it as follows

```
<div app="components"></div>
```

Data Binding in Angular

In this tutorial, we are going to look at the How Data Binding works in Angular with examples. Angular Components are useless if they do not show any dynamic data. They also need to respond to user interactions and react to events. The data binding kees both component & view in sync with each other. We use techniques like Interpolation, Property Binding, Event Binding & Two Way Binding to bind data. We also learn how to

use the ngModel directive to achieve the two-way binding in Angular Forms.

Table of Content

What is Angular 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

Data Binding in Angular 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 View to Component : To bind data from view to the component, we make use of **interpolation** or **Property Binding.**

Interpolation

Interpolation 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**

The Angular first evaluates the **Template Expression and converts it into a string.** Then it **replaces Template expression** with the result in the **original string in the HTML**. Whenever the template expression changes, the Angular updates the original string again

Example

```
1
2 Welcome,  {{firstName}} {{lastName}}
3
```

```
4
5 import { Component } from '@angular/core';
6
7 @Component({
8  selector: 'app-root',
9  templateUrl: './app.component.html',
1  styleUrls: ['./app.component.css']
0 })
1 export class AppComponent {
1
  firstName= 'Sachin';
1
2  lastName=”Tendulkar”
1 }
3
1
4
```

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

The Property binding allows us to **bind HTML element property to a property in the component**.

Whenever the value of the component changes, the Angular updates the element property in the View. You can set the properties such as class, href, src, textContent, etc using property binding. 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.

Example

app.component.html

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

app.component.ts

```
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;

}
```

The `title` property of the component class is bound to the `innerText` property of the `h1` tag. And d`isabled` 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
- ngStyle directive

Attribute binding

Sometimes there is no HTML element property to bind. The examples are aria (accessibility) Attributes & 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

```
1
2 <button [attr.aria-label]="closeLabel" (onclick)="closeMe()">X</bu
3
```

From Component to View

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. By tracking the user events in the view and responding to it, we can keep our component in sync with the view. 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.

For Example,

```
<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 t**wo-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.

Interpolation in Angular

We use interpolation to bind a **component** property, method or to a **template reference variable** to a string literal in the view.

We also call this as **string interpolation**. It is **one way** from **component to view** as the **data flow from the component to view.**

What is Interpolation in Angular

Interpolation 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.

Angular interpolation is also known by the name string interpolation. Because you incorporate expressions inside another string.

Interpolation syntax

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**

The Angular first evaluates the **Template Expression and converts it into a string**. Then it **replaces Template expression with the result in the original string in the HTML**. Whenever the template expression changes, the Angular updates the original string again

Interpolation Example

Create a new angular application using the following command

```

ng new interpolation

```

Open the `app.component.html` and just add the following code

```

{{title}}

```

Open the `app.component.ts` and add the following code

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

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

Run the app. You will see the “Angular Interpolation Example” on the screen

In the example above the title is the Template Expression. We also have title property in the component. The Angular evaluates {{title}} and replaces it with the value of the title property from the component class.

If the user changes the title in the component class, the Angular updates the view accordingly.

The interpolation is much more powerful than just getting the property of the **component.** You can use it to invoke any method on the component class or to do some mathematical operations etc.

Notes on Interpolation

Interpolation is one-way binding

Interpolation is one way as values go from the component to the template. When the component values change, the Angular updates the view. But if the values changes in the view components are not updated.

Should not change the state of the app

The Template expression should not change the state of the application. The Angular uses it to read the values from the component and populate the view. If the Template expression changes the component values, then the rendered view would be inconsistent with the model

It means that you cannot make use of the following

- Assignments (`=`, `+=`, `-=`, ...)
- Keywords like `new`, `typeof`, `instanceof`, etc
- Chaining expressions with `;` or `,`
- The increment and decrement operators `++` and `--`
- bitwise operators such as `|` and `&`

The expression must result in a string

Interpolation expression must result in a string. **If we return an object it will not work.** If you want to bind the expression that is other than a string (for example – boolean), then Property Binding is the best option.

Works only on Properties & not attributes

Interpolation and property binding can set only properties, not attributes. For Attributes use attribute binding

Examples of interpolation

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

Invoke a method in the component

We can invoke the component’s methods using interpolation.

```
1
2
3  //Template
4
5  {{getTitle()}}
6
7
8  //Component
9  title = 'Angular Interpolation Example';
1  getTitle(): string {
0    return this.title;
1  }
1
1
2
```

Concatenate two string

```
<p>Welcome to {{title}}</p>
<p>{{ 'Hello & Welcome to '+ ' Angular Interpolation '}}</p>
<p>Welcome {{firstName}}, {{lastName}}</p>
<p>Welcome {{getFirstName()}}, {{getLastName()}}</p>
```

Perform some mathematical operations

```
<h2>Mathematical Operations</h2>

<p>100x80 = {{100*80}}</p>
<p>Largest: {{max(100, 200)}}</p>

//Component
max(first: number, second: number): number {
 return Math.max(first, second);
}
```

Bind to an element property

We can use it to bind to a property of the HTML element, a component, or a directive. in the following code, we bind to the `style.color` property of the `<p>` element. We can bind to any property that accepts a string.

```
<p>Show me <span class = "{{giveMeRed}}">red</span></p>
<p style.color={{giveMeRed}}>This is red</p>
```

Bind to an image source

```
<div><img src="{{itemImageUrl}}"></div>
```

href

```
<a href="/product/{{productID}}">{{productName}}</a>
```

Use a template reference variable

You can also use the template reference variable. The following example creates a template variable **#name** to an input box.

You can use it get the value of the input field `{{name.value}}`

```
<label>Enter Your Name</label>
<input (keyup)="0" #name>
<p>Welcome {{name.value}} </p>
```

We also use `(keyup)="0"` on the `input` element. It does nothing but it forces the angular run the change detection, which in turn updates the view.

Property Binding in Angular

Property binding is **one way from component to view.** It lets you set a property of an element in the view to property in the component. You can set the properties such as `class`, `href`, `src`, `textContent`, etc using property binding. You can also use it to set the properties of custom components or directives (properties decorated with `@Input`).

Table of Content

Property Binding Syntax

The Property Binding uses the following Syntax

```
1
2 [binding-target]=”binding-source”
3
```

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.

Property Binding Example

Create a new application

```

ng new property

```

Open `app.component.html`

```

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

```

Open the `app.component.ts`

```
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;

}
```

We have two property binding in the example above

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` is the component, the Angular automatically updates the view.

Property Binding is one way

Property binding is one way as values go from the component to the template. When the component values change, the Angular updates the view. But if the values changes in the view, the Angular does not update the component.

Should not change the state of the app

The Angular evaluates the template expression (binding-source) to read the values from the component. It then populates the view. If the expression changes any of the component values, then the view would be inconsistent with the model. Hence we need to avoid using expression which will alter the component state.

It means that you cannot make use of the following

- Assignments (`=`, `+=`, `-=`, ...)
- Keywords like `new`, `typeof`, `instanceof`, etc
- Chaining expressions with `;` or `,`
- The increment and decrement operators `++` and `--`
- bitwise operators such as `|` and `&`

Return the proper type

The binding expression should return the correct type. The type that the target property expects. Otherwise, it will not work

Property name in camel case

There are few element property names in the camel case, while their corresponding attributes are not. For example `rowSpan` & `colSpan` properties of the table are in the camel case. The HTML attributes are all lowercase (`rowspan` & `colspan`)

Remember the brackets

The brackets, `[]`, tell Angular to evaluate the template expression. If you omit the brackets, Angular treats the expression as a constant string and initializes the

target property with that string:

Content Security

Angular inspects the template expression for untrusted values and sanitizes them if found any. For example, the following component variable `evilText` contains the `script` tag. This is what we call the script injection attack. **The Angular does not allow HTML with script tags**. It treats the entire content as string and displays as it is.

```
Component

evilText = 'Template <script>alert("You are hacked")</script> Sy

Template

<p [textContent]="evilText"></p>
```

DOM Properties, not attributes

The property binding binds to the *properties* of DOM elements, components, and directives and not to HTML attributes. The angular has a special syntax for attribute binding.

Special Binding

The Angular has a special syntax for class, styles & attribute binding

The classes & styles are special because they contain a list of classes or styles. The bindings need to be more flexible in managing them. Hence we have a class

& style binding.

The Property bindings cover all the properties, but there are certain HTML attributes that do not have any corresponding HTML property. Hence we have attribute binding

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

Similar to the class, the style also can be set using the following ways. Click on the links to find out more

- Style Property Binding
- ngStyle directive

Attribute Binding

Sometimes there is no HTML element property to bind to. The examples are aria (accessibility) Attributes & 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

```
//Template

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

//Table colspan
<table border="1">
 <tr>
  <td>Col 1</td>
  <td>Col 2</td>
  <td>Col 3</td>
 </tr>
 <tr>
   <td [attr.colspan]="2">Col 1 & 2</td>
   <td>Col 3</td>
 </tr>
 <tr>
   <td>Col 1</td>
   <td bind-attr.colspan = "getColspan()">Col 2 & 3 </td>
 </tr>
 <tr>
   <td>Col 1</td>
   <td>Col 2</td>
   <td>Col 3</td>
  </tr>
```

```
//Component

closeLabel="close";
getColspan() {
 return "2"
}
```

Property Binding Vs Interpolation

Everything that can be done from interpolation can also be done using the Property binding. Interpolation is actually a shorthand for binding to the `textContent` property of an element.

For example the following interpolation

```
1
2 <h1> {{ title }} </h1>
3
```

Is same as the following Property binding

```
1
2 <h1 [innerText]="title"></h1>
3
```

In fact, Angular automatically translates interpolations into the corresponding

property bindings before rendering the view.

Interpolation is simple and readable. For example, the above example of setting the `h1` tag, the in interpolation is intuitive and readable than the property binding syntax

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

Property Binding Example

Binding to innerHTML with HTML tags

Here the Angular parses the `b` &`p` tags and renders it in the view.

```

//Template
<p [innerHTML]="text1"></p>
<div [innerHTML]="text2"></div>

```

```

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

```

```

//Component
itemImageUrl="https://angular.io/assets/images/logos/angular/logo

```

Concatenate two string

```

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

```

Mathematical expressions

```

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

```

setting the color

```

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

```

```
1
2 //Component
3 color='red'
4
```

Event Binding in Angular

In this guide, we will explore the Event Binding in Angular. Event binding is one way from view to component. We use it to perform an action in the component when the user performs an action like clicking on a button, changing the input, etc in the view.

Table of Content

Event Binding

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

It is one way from view to component. By tracking the user events in the view and responding to it, we can keep our component in sync with the view. For Example, when the user changes to an 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.

Syntax

The Angular event binding consists of two parts

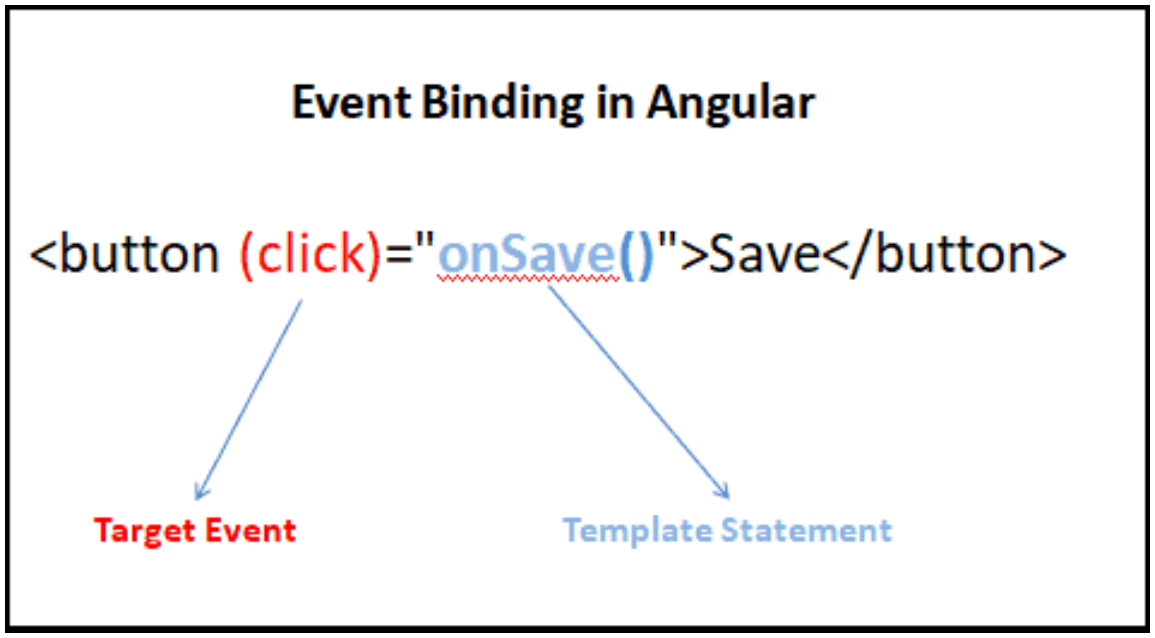
```
1
2 <(target-event)="TemplateStatement">
3
```

- We enclose the target event name in parentheses on the left side
- Assign it to a template statement within a quote on the right side
-

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.

The following event binding listens for the button's `click` events, calling the component's `onSave()` method whenever a click occurs

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



Event Binding Example

Create a new angular application

```
ng new event
```

Copy the following code to app.component.html

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

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

```

Add the following code to the `app.component.ts`

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

```

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

Template statements have side effects

Unlike the **Property Binding & Interpolation**, where we use the template expression is used, in case of event binding we use template statement.

The Template statement can change the state of the component. Angular runs the change detection and updates the view so as to keep it in sync with the component.

on-

Instead of parentheses, you can also use the `on-` syntax as shown below.

```

<button on-click="clickMe()">Click Me</button>

```

Multiple event handlers

You can also bind an unlimited number of event handlers on the same event by separating them with a semicolon `:`

Add a new component property

```

 clickCount1=0;

```

And in the template, call `clickMe()` method and then an assignment `clickCount1=clickCount`

```

//Template

<h2>Example 2</h2>
<button (click)="clickMe(); clickCount1=clickCount">Click Me</bu
<p>You have clicked {{clickCount1}}</p>

```

$event Payload

DOM Events carries the event payload. I.e the information about the event. We can access the event payload by using `$event` as an argument to the handler function.

```

<input (input)="handleInput($event)">
<p>You have entered {{value}}</p>

```

And in the component

```

value=""
handleInput(event) {
 this.value=event.target.value;
}

```

The properties of a `$event` object vary depending on the type of DOM event. For example, a mouse event includes different information than an input box editing event.

Remember you need to use the variable as `$event` in the Template statement. Example `handleInput($event)`. Otherwise, it will result in an error

Template reference variable

We can also make use of the template reference variable to pass the value instead of `$event`.

```
<h2>Template Reference Variable</h2>
<input #el (input)="handleInput1(el)">
<p>You have entered {{val}}</p>
```

In the template

```
val="";
handleInput1(element) {
 this.val=element.value;
}
```

Key event filtering (with key.enter)

We use keyup/keydown events to listen for keystrokes. The following example does that

```
<input (keyup)="value1=$event.target.value">
<p>You entered {{value1}}</p>
```

But Angular also offers a feature, where it helps to filter out certain keys. For Example, if you want to listen only to the `enter` keys you can do it easily

```
<input (keyup.enter)="value2=$event.target.value">
<p>You entered {{value2}}</p>
```

Here is an interesting example. On pressing `enter` key it updates the `value3` variable and on `escape` clears the variable.

```
<input (keyup.enter)="value3=$event.target.value"
(keyup.escape)="$event.target.value='';value3=''">
<p>You entered {{value3}}</p>
```

Angular calls these pseudo-events.

You can also listen for the key combination

```
<input (keyup.control.shift.enter)="value4=$event.target.value">
<p>You entered {{value4}}</p>
```

Note : Custom events with EventEmitter

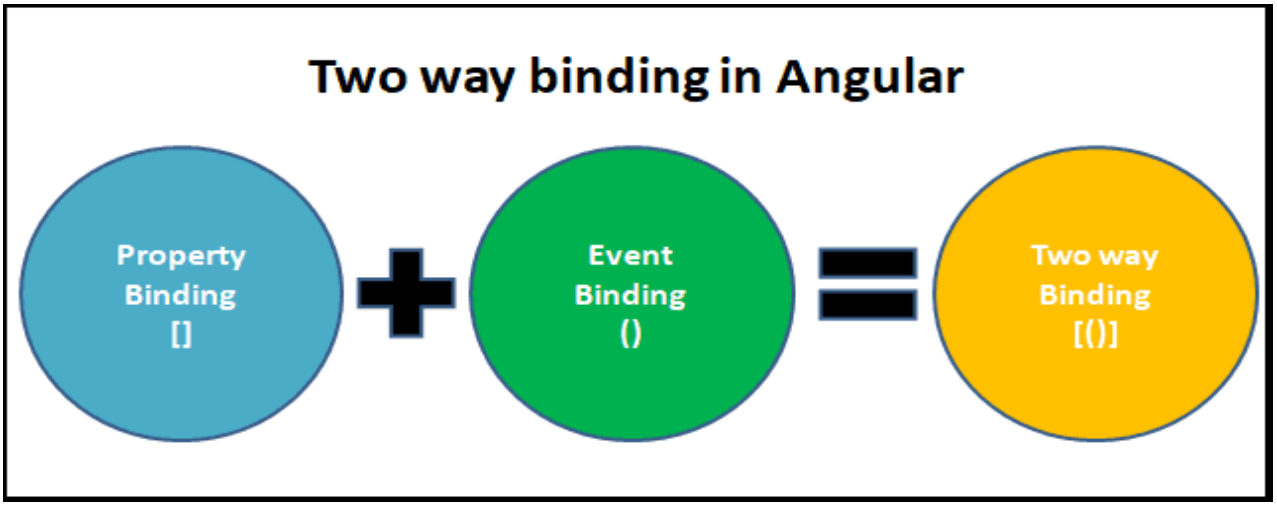
Directives & components can also raise events with EventEmitter. Using EventEmiiiter you can create a property and raise it using the `EventEmitter.emit(payload)`. The Parent component can listen to these events using the event binding and also read the payload using the `$event` argument.

What is Two way data binding

Two way data 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 data.

Two way data 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 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.



Two way using property & Event Binding

The following example shows how we can achieve two-way binding using the combination of property binding & event binding

Create a new Angular application

`app.component.html`

```
h2>Example 1</h2>
<input type="text" [value]="name" (input)="name=$event.target.
<p> You entered {{name}}</p>
<button (click)="clearName()">Clear</button>
```

Update the `app.component.ts` with the following code.

```
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.

Two-way binding syntax

The above example uses the event & property binding combination to achieve the two-way binding. But Angular does provide a way to achieve the two-way binding using the syntax [()]. Note that both square & parentheses are used here. This is now known as **Banana in a box** syntax. The square indicates the Property binding & parentheses indicates the event binding.

For Example

```
1
2 <someElement [(someProperty)]="value"></someElement>
3
```

The above syntax sets up both property & event binding. But to make use of it, the property must follow the following naming convention.

If we are binding to a settable property called `someProperty` of an element, then the element must have the corresponding change event named `somePropertyChange`.

But most HTML elements have a `value` property. But do not have a `valueChange` event, instead, they usually have an `input` event. Hence they cannot be used in the above syntax

For Example, the following will not work as there is no `valueChange` event supported by the `input` element.

Hence we have a `ngModel` directive.

What is 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.

Internally It uses the `ngModel` in property, binding to bind to the `value` property and `ngModelChange` which binds to the input event.

How to use ngModel

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

In the template use the following syntax

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

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

ngModel Example

Import FormsModule

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

```
1
2 import { FormsModule } from '@angular/forms';
3
```

Template

```
1
2 <h2>Example 2</h2>
3 <input type="text" name="value" [(ngModel)]="value">
4 <p> You entered {{value}}</p>
5 <button (click)="clearValue()">Clear</button>
6
```

Component

```
1
2 value="";
3 clearValue() {
4  this.value="";
5 }
6
7
```

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`.

We do not have any HTML Elements which follows the above naming conventions, but we can create a custom component

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

```
import { Component, Input, Output, EventEmitter } from '@angula
@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 EventEm

 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>
```

Summary

The two-way binding is a simple, but the yet powerful mechanism. We use the Property binding & Event binding to achieve the two-way binding. Angular does have a [(value)] syntax to which sets up the two-way binding. It automatically sets up property binding to value property of the element. It also sets up the event binding to valueChange Property. But since we hardly have any HTML element, which follows those naming conventions unless we create our own component. This is where ngModel directive from FormsModule steps in and provides two way binding to all the known HTML form elements.

Child/Nested Components in Angular

In this tutorial, we will learn how to Add a child Component to our Angular Application. The `AppComponent` is bootstrapped 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.

In this tutorial, we will learn how to create a Child or nested components and host it in the App Component.

Create a new application

Create a new Angular application using the following command

```

ng new childComponent

```

Run the app and verify everything is ok.

```
ng serve
```

How to add Child Component

1. Create the Child Component. In the child Component, metadata specify the selector to be used

2. Import the Child Component in the module class and declare it in declaration Array

3. Use the CSS Selector to specify in the Parent Component Template, where you want to display the Child Component

Adding a Child Component in Angular

Now, let us add a Child Component to our project. In our child component, let us display a list of customers.

Create the Child Component

Creating the Child Component is no different from creating any other Parent Component. But, first, we need a customer class

Customer Class

Go to the app folder and create a file and name it as `customer.ts`. Copy the following code

```
export class Customer {

 customerNo: number;
 name:string ;
 address:string;
 city:string;
 state:string;
 country:string;

}
```

Note that we have used the `export` keyword. This enables us to use the above class in our components by importing it.

Create Child Component

In the `app` folder and create a new file and name it as `customer-list.component.ts`.

The code for `component-list.component.ts` is shown below

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

@Component({
 selector: 'customer-list',
 templateUrl: './customer-list.component.html'
})
export class CustomerListComponent
{
 customers: Customer[] = [

  {customerNo: 1, name: 'Rahuld Dravid', address: '', city: 'Banglac
  {customerNo: 2, name: 'Sachin Tendulkar', address: '', city: 'Mum
  {customerNo: 3, name: 'Saurrav Ganguly', address: '', city: 'Kolka
  {customerNo: 4, name: 'Mahendra Singh Dhoni', address: '', city:
  {customerNo: 5, name: 'Virat Kohli', address: '', city: 'Delhi', state

 ]
}
```

First, we import the required modules & classes. Our component requires `Customer` class, hence we import it along with the Component.

```
1
2 import { Component } from '@angular/core';
3 import { Customer } from './customer';
4
```

The next step is to add the @Component directive. The selector clause has the value customer-list. We need to use this in our parent view to display our view. The templateURL is customer-list.component.html, which we yet to build.

```
1
2 @Component({
3  selector: 'customer-list',
4  templateUrl: './customer-list.component.html'
5 })
6
```

The last step is to create the Component class. We name it as CustomerListComponent. The class consists of a single property, which is a collection of customers. We initialize the customers collection with some default values. In real-life situations, you will make use of the HTTP Client to get the data from the back end server.

```
export class CustomerListComponent
{
 customers: Customer[] = [

  {customerNo: 1, name: 'Rahuld Dravid', address: '', city: 'Banglac
  {customerNo: 2, name: 'Sachin Tendulkar', address: '', city: 'Mum
  {customerNo: 3, name: 'Saurrav Ganguly', address: '', city: 'Kolka
  {customerNo: 4, name: 'Mahendra Singh Dhoni', address: '', city:
  {customerNo: 5, name: 'Virat Kohli', address: '', city: 'Delhi', state

 ]
}
```

Creating the View

The next step is to create the View to display the list of customer. Go to the `app` folder and create the file with the name `customer-list.component.html`

```
<h2>List of Customers</h2>

<table class='table'>
 <thead>
  <tr>
   <th>No</th>
   <th>Name</th>
   <th>Address</th>
   <th>City</th>
   <th>State</th>
  </tr>
 </thead>
 <tbody>
  <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>
```

To iterate through the Customer collection, we have used the ngFor Directive provided by the Angular. We have a separate tutorial, which discusses the ngFor Directive

The syntax for `ngFor` directive starts with `*ngFor`. The `*` indicates that it is a structural directive. i.e a directive that adds or removes the HTML elements to or from the DOM.

The expression `let customer of customers` is assigned to `*ngFor`. The `let` clause assigns the instance of `customer` object from the `Customers` collection to the template reference variable or local variable `customer`.

We use the template reference variable `customer` is to build the template to display the details of the customer to the user. The `ngFor` directive is applied to the `tr` element of the table. The Angular repeats everything inside the `tr` element in the DOM tree.

`{{customer.customerNo}}` is stands for interpolation in Angular. Angular evaluates everything within the `{{ }}` and replaces the string with the result.

Import the Child Component in the Module

Every Component, directive, pipes we build must belong to an Angular Module. To do that we need to register our component in the Module. A Component, directive, piles cannot be part of more than one module.

The Angular Modules (or NgModules) are Angular ways of organizing related components, directives, pipes and services etc into a group. To add a component to a module, you need to declare it in the declarations metadata of the Angular Module

Angular creates a top-level root module (AppModule in file app.module.ts) when we create a new Angular app. That is where we need to register our CustomerListComponent

Open the app.module.ts under the app folder and update the code as shown below

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

import {NgbModule} from '@ng-bootstrap/ng-bootstrap';

import { AppComponent } from './app.component';
import {CustomerListComponent} from './customer-list.component';

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

Registering the component or directive in the module requires two steps

First, import it

```
import { AppComponent } from './app.component';
import {CustomerListComponent} from './customer-list.component';

```

And, then declare it in declaration array

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

```

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 `selector`in the metadata for the component. This CSS selector name must match the element tag that specified within the parent component's template.

```
1
2 <customer-list></customer-list>
3
```

Run the application from the command line using **ng Serve**

Summary

In this tutorial, we looked at how to add a child component to our application.