Angular Data Binding

In this article, we will discuss the concept of Data Binding in Angular 8. Data Binding is one of the most important features of the Angular framework. This is because, at any web-based application, data pushing and pulling is always performed a key role in the development. In Angular, this can be done with the help of the Data Binding concept very easily.

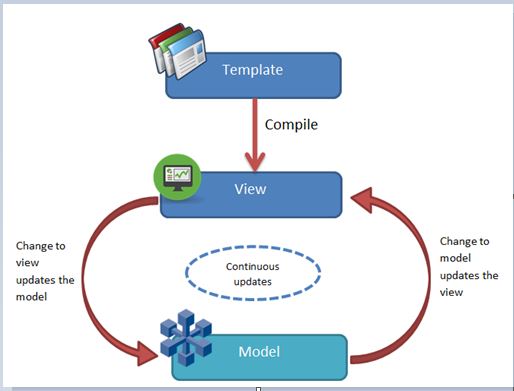
**What is Data Binding?**

Data binding is one of the finest and useful features of the Angular framework. Due to this feature, the developer needs to write less code compared to any other client-side library or framework. Data binding in an Angular application is the automatic synchronization of data between the model and view components. In Angular, we can always treat the model as a single-source-of-truth of data in our web application with the data binding. In this way, the UI or the view always represents the data model at all times. With the help of data binding, developers can be established a relation between the application UI and business logic. If we establish the data binding in a correct way, and the data provides the proper notifications to the framework, then, when the user makes any data changes in the view, the elements that are bound to the data reflect changes automatically.

**Basic Concept of Data Binding**

In case of any web-based application development, we always need to develop a connection bridge between the back end, where data is stored and the front end or user interface through which user performs their application data manipulation. Now, this entire process always depends on consecutive networking interactions, repetitive communication between the server (i.e. back end) and the client (i.e. front end).

Due to the consecutive and repetitive communication between client and server, most web framework platforms focus on one-way data binding. Basically, this involves reading the input from DOM, serializing the data, sending it to the back end or server, and waiting for the process to finish. After that, the process modifies the DOM to indicate if any errors occurred or reload the DOM element if the call is successful. While this process provides a traditional web application all the time it needs to perform data processing, this benefit is only really applicable to web apps with highly complex data structures. If your application has a simpler data structure format, with relatively flat models, then the extra work can needlessly complicate the process and decrease the performance of your application.



The angular framework addresses this with the data binding concept. Data binding provides a continuous data upgradation process so that when the user makes any changes in the interface that will automatically update the data and vice-versa. In this way, the data model of the application always acts as an atomic unit so that the view of the application can be always updated in respect of the data. This process can be done with the help of a complex series of event handlers and event listeners in many frameworks. But that approach can be fragile very quickly. In Angular Framework, this process related to the data becomes one of the primary parts of its architecture. Instead of creating a series of call-backs to handle the changing data, Angular does this automatically without any needed intervention by the programmer. Basically, this feature is a great relief for the programmer and reduces much more development time.

So, the first and foremost advantage of data binding is that it updates the data models automatically in respect of the view. So, when the data model updates, that will automatically update the related view element in the application. In this way, angular provides us a correct data encapsulation on the front end and it also reduced the requirement to perform complex and destructive manipulation of the DOM elements.

**Why Data Binding Required?**

From day one, the Angular framework provides this special and powerful feature called Data Binding which always brings smoothness and flexibility in any web application. With the help of data binding, developers can gain better control over the process and steps which are related to the data binding process. This process makes the developer's life easier and reduces development time with respect to other frameworks. Some of the reasons related to why data binding is required for any web-based application are listed below –

1. With the help of data binding, data-based web-pages can be developed quickly and efficiently.
2. We always get the desired result with a small volume of coding size.
3. Due to this process, the execution time increases. As a result, it improves the quality of the application.
4. With the help of event emitter, we can achieve better control over the data binding process.

**Different Types of Data Binding**

In Angular 8, there are four different types of Data binding processes available. They are:

* Interpolation
* Property Binding
* Two-Way Binding
* Event Binding

**Interpolation**

Interpolation data binding is the most popular and easiest way of data binding in Angular 8. This feature is also available in previous Angular framework versions. Actually, the context between the braces is the template expression that Angular first evaluates and then convert into strings. Interpolation uses the braces expression i.e. {{ }} to render the bound value to the component template. It can be a static string, numeric value, or an object of your data model. In Angular, we use it like this: {{firstName}}.  
The below example shows how we can use the interpolation in the component to display data in the front end.

1. <div>
2. <span>User Name : {{userName}}</span>
3. </div>

**Property-Based Binding**

In Angular 8, another binding mechanism exists, which is called Property Binding. In nature, it is just the same as interpolation. Some people also called this process as one-way binding like the previous AngularJS concept. Property binding used [] to send the data from the component to the HTML template. The most common way to use property binding is to assign any property of the HTML element tag into the [] with the component property value, i.e:

1. <input type=”text” [value]=”data.name”/>

To implement the property binding, we will just make the below changes in the previous HTML file from the interpolation sample i.e. interpolation.component.html

1. <div>
2. <input [value]="value1" />
3. <br /><br />
4. </div>

**Event Binding**

Event binding is another of the data binding techniques available in Angular. This data binding technique does not work with the value of the UI elements—it works with the event activities of the UI elements like click-event, blur-event, etc. In the previous version of AngularJS, we always used different types of directives like ng-click, ng-blur to bind any particular event action of an HTML control. But in the current Angular version, we need to use the same property of the HTML element (like click, change, etc.) and use it within parentheses. In Angular 8, for properties, we use square brackets, and in events, we use parentheses.

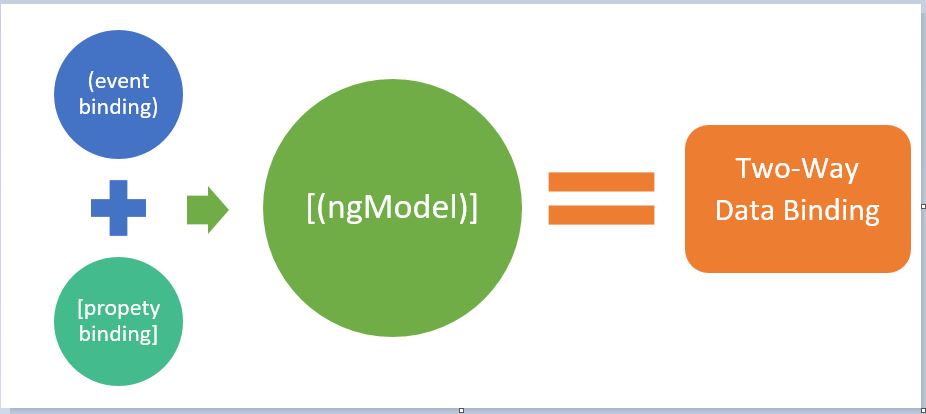
1. <div>
2. <input type="submit" value="Submit" (click)="fnSubmit()">
3. </div>

**Two-Way Data Binding**

In Angular Framework, the most used and important data binding techniques are known as Two-Way Data Binding. Two-way binding is mainly used in the input type field or any form element where the user can provide input values from the browser or provides any value or changes any control value through the browser. On the other side, the same is automatically updated into the component variables and vice versa. Similarly, in Angular 8 we have a directive called ngModel, and it needs to be used as below:

1. <input type=”text” [(ngModel)] =”firstName”/>

We use [] since it is actually a property binding, and parentheses are used for the event binding concept i.e. the notation of two-way data binding is [()].



ngModel performs both property binding and event binding. Actually, the property binding of the ngModel (i.e. [ngModel]) performs the activity to update the input element with a value. Whereas (ngModel) ( (ngModelChange) event) instructs the outside world when any change occurred in the DOM Element.

The below example demonstrates the implementation of two-way binding. In this example, we define a string variable called strName and assign that variable with a Textbox control. So, whenever we change any content in the textbox, the value of the variable will be changed automatically.

1. <div>
2. <input [(ngModel)]="strName" type="text"/>
3. </div>

**@Input() Decorator**

In Angular Framework, each and every component used either as a stateless component or stateful component. Normally, the components are used as a stateless way. But sometimes we need to use some stateful components. The reason behind using a stateful component in a web application is due to the data passing or receiving from the current component to either parent component or a child component. So, in this way, we need to intimate Angular that what type of data or which data may come to our current component. To implement this concept, we need to use the @Input() decorator against any variable. The key features of @Input() decorator are as follows:

* @Input is a decorator to mark an input property. With the help of this property, we can define an input parameter property just like normal HTML tag attributes to pass and bind that value into the component as a property binding.
* @Input decorator always provides a one-way data communications from parent component to child component. Using this feature, we can provide some of the value against any property of a child component from the parent components.
* The component property should be annotated with the @Input decorator to act as an input property. A component can receive value from another component using component property binding.

It can be annotated as any type of property, such as number, string, array, or user-defined class. To use an alias for the binding property name, we need to assign an alias name as @Input(alias). Find the use of @Input with the string data type.

1. @Input() caption : string;
2. @Input('phoneNo') phoneNo : string;

In the above example, testValue is the alias name. The alias name is needed when we want to pass the value to the input properties. If an alias is not defined, then we need to use the input property name to pass the value.  
Now, when we use this component, we need to pass the input values as below:

1. <demo-app [name]="'Debasis Saha'" [phoneNo]="9830098300"></demo-app>

**@Output() Decorator**

In Angular 8, @Output is a decorator to normally used as an output property. @Output is used to define output properties to achieve custom event binding. @Output will be used with the instance of Event Emitter. The key features of the @Output() decorator are as follows:

* The @Output decorator binds a property of a component to send data from one component (child component) to a calling component (parent component).
* This is one-way communication from a child to a parent component.
* @Output binds a property of the EventEmitter class. If we assume component as a control, then output property will act as an event of that control.
* The @Output decorator can also provide options to customize the property name by using the alias name as @Output(alias). In that case, this custom alias name will act as an event binding property name of the component.

It can be entered in any type of property such as number, string, array, or user-defined class. To use an alias for the binding property name, we need to assign an alias name as @Output(alias). Find the use of @Output with the string data type.  
Just like Input decorator, output decorator also need to first declare as below –

1. @Output('onSubmit') submitEvent = **new** EventEmitter<any>();

Now, we need to raise the emitter from a point within the component so that the event can be raised and tracked by the parent component.

1. **this**.submitEvent.emit();

If we want to pass any value through this event emitter, then we need to pass that value as a parameter through the emit().  
In the parent component, that output property will be defined as below –

1. <demo-app (onSubmit)="receiveData($event)"></demo-app>

**Demo 1: Interpolation**

Now, in this demo, we will show how to use or implement Interpolation in Angular 8 applications for different data types.

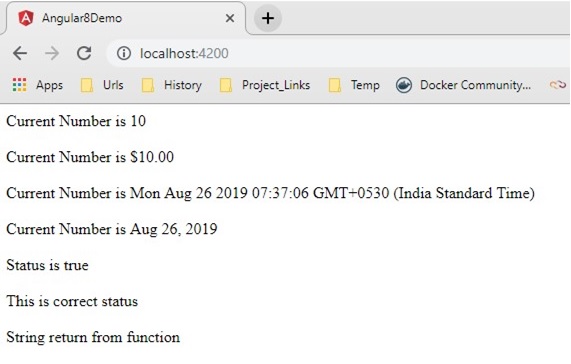
**app.component.ts**

1. **import** { Component } from '@angular/core';
3. @Component({
4. selector: 'app-root',
5. templateUrl: './app.component.html',
6. styleUrls : ['./custom.css']
7. })
8. **export** **class** AppComponent {
9. **public** value1: number = 10;
10. **public** array1: Array<number> = [10, 22, 14];
11. **public** dt1: Date = **new** Date();
13. **public** status: **boolean** = **true**;
15. **public** returnString(): string {
16. **return** "String return from function";
17. }
18. }

**app.component.html**

1. <div>
2. <span>Current Number is {{value1}}</span>
3. <br /><br />
4. <span>Current Number is {{value1 | currency}}</span>
5. <br /><br />
6. <span>Current Number is {{dt1}}</span>
7. <br /><br />
8. <span>Current Number is {{dt1 | date}}</span>
9. <br /><br />
10. <span>Status is {{status}}</span>
11. <br /><br />
12. <span>{{status ? "This is correct status" :"This is false status"}}</span>
13. <br /><br />
14. <span>{{returnString()}}</span>
15. </div>

Now, the output of the above code is as below -



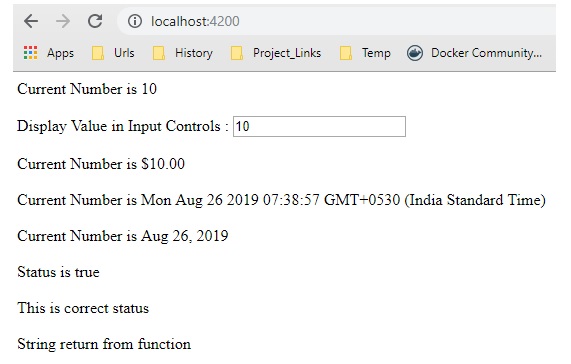
**Demo 2: Property-Based Binding**

Now, in this demo, we will demonstrate how to use Property-Based binding in Angular 8. For that, we will add one input type text box in the app.component.html file and bind the text box with variable value1 using property binding.

**app.component.html**

1. <div>
2. <span>Current Number is {{value1}}</span>
3. <br/><br />
4. Display Value **in** Input Controls : <input [value]="value1" />
5. <br /><br />
6. <span>Current Number is {{value1 | currency}}</span>
7. <br /><br />
8. <span>Current Number is {{dt1}}</span>
9. <br /><br />
10. <span>Current Number is {{dt1 | date}}</span>
11. <br /><br />
12. <span>Status is {{status}}</span>
13. <br /><br />
14. <span>{{status ? "This is correct status" :"This is false status"}}</span>
15. <br /><br />
16. <span>{{returnString()}}</span>
17. </div>

Now, check the output in the browsers –



**Demo 3: Event-Based Binding**

In this demo, we will demonstrate how to implement event-based binding in Angular 8.

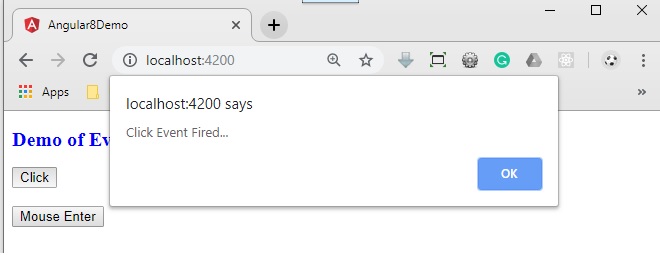
**app.component.ts**

1. **import** { Component } from '@angular/core';
3. @Component({
4. selector: 'app-root',
5. templateUrl: './app.component.html',
6. styleUrls : ['./custom.css']
7. })
8. **export** **class** AppComponent {
10. **public** showAlert() : **void** {
11. console.log('You clicked on the button...');
12. alert("Click Event Fired...");
13. }
14. }

**app.component.html**

1. <div>
2. <h2>Demo of Event Binding **in** Angular 8</h2>
3. <input type="button" value="Click" **class**="btn-block" (click)="showAlert()" />
4. <br /><br />
5. <input type="button" value="Mouse Enter" **class**="btn-block" (mouseenter)="showAlert()" />
6. </div>

Now, check the output in the browser:-



**Demo 4: Two Way Data Binding**

Now, in this demo, we will demonstrate how to use two-way data binding in angular. For this purpose, create the below-mentioned components.

**app.component.ts**

1. **import** { Component } from '@angular/core';
3. @Component({
4. selector: 'app-root',
5. templateUrl: './app.component.html',
6. styleUrls : ['./custom.css']
7. })
8. **export** **class** AppComponent {
10. **public** val: string = "";
11. }

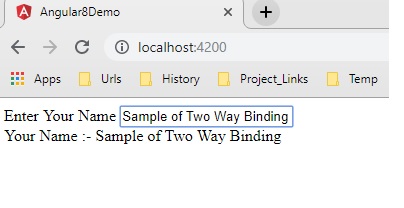
**app.component.html**

1. <div>
2. <div>
3. <span>Enter Your Name </span>
4. <input [(ngModel)]="val" type="text"/>
5. </div>
6. <div>
7. <span>Your Name :- </span>
8. <span>{{val}}</span>
9. </div>
10. </div>

Now, when we use ngModel or two-way data binding in our components, then we need to include FormsModule of Angular in our own defined app module file as below. Since ngModel will not work without this FormsModule.

1. **import** { BrowserModule } from '@angular/platform-browser';
2. **import** { NgModule } from '@angular/core';
3. **import** { FormsModule } from '@angular/forms';
5. **import** { AppComponent } from './app.component';
7. @NgModule({
8. declarations: [
9. AppComponent
10. ],
11. imports: [
12. BrowserModule,FormsModule
13. ],
14. providers: [],
15. bootstrap: [AppComponent]
16. })
17. **export** **class** AppModule { }

Now, check the output in the browser –



**Demo 5: Pass Input Value into Component**

In this demo, we will demonstrate how to use or implement the input property of a component. For that purpose, we need to develop the first one component in which input property will be defined.

**message.component.ts**

1. **import** { Component, Input } from '@angular/core';
3. @Component({
4. selector: 'message-info',
5. templateUrl: './message.component.html',
6. styleUrls : ['./custom.css']
7. })
8. **export** **class** MessageComponent {
10. @Input() **public** message :string = '';
12. @Input('alert-pop') **public** message1 :string= ''
14. **public** showAlert():**void**{
15. alert(**this**.message1);
16. }
17. }

**message.component.html**

1. <div>
2. Message : <h2>{{message}}</h2>
3. <input type="button" value="Show Alert" (click)="showAlert()"/>
4. </div>

Now, we need to consume this message-info component into another component and need to pass the input value using input properties.

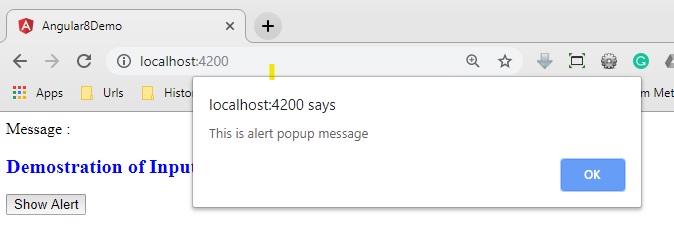
**app.component.ts**

1. **import** { Component } from '@angular/core';
3. @Component({
4. selector: 'app-root',
5. templateUrl: './app.component.html',
6. styleUrls : ['./custom.css']
7. })
8. **export** **class** AppComponent {
10. **public** val: string = "This is alert popup message";
12. }

**app.component.html**

1. <div>
2. <message-info [message]="'Demostration of Input Property of a Component'" [alert-pop]="val"></message-info>
3. </div>

Now, check the output at the browser –



**Demo 5: Return Output Value from a Component**

Now in this demo, we will discuss how to use the output property of any component. For that purpose, we make the following changes in the message-info component to define output property.

**message.component.ts**

1. **import** { Component, Input, EventEmitter, Output } from '@angular/core';
3. @Component({
4. selector: 'message-info',
5. templateUrl: './message.component.html',
6. styleUrls : ['./custom.css']
7. })
8. **export** **class** MessageComponent {
10. @Input() **public** message :string = '';
11. @Input('alert-pop') **public** message1 :string= ''
13. @Output() onSignup  = **new** EventEmitter<any>();
15. **public** data:any={};
17. **public** showAlert():**void**{
18. alert(**this**.message1);
19. }
21. **public** onSubmit() :**void**{
22. **this**.onSignup.emit(**this**.data);
23. }
24. }

**message.component.html**

1. <div>
2. Message : <h2>{{message}}</h2>
3. <input type="button" value="Show Alert" (click)="showAlert()"/>
4. <br/><br/>
5. Provide Full Name : <input type="text" [(ngModel)]="data.name">
6. <br/>
7. Provide Email Id : <input type="email" [(ngModel)]="data.email">
8. <br>
9. <input type="button" value="Sign Up" (click)="onSubmit()"/>
10. </div>

So, in the above part, we defined an output property called onSignup() within the message-info component. Now, we need to consume that event in the parent component as shown below –

**app.component.html**

1. <div>
2. <message-info [message]="'Demostration of Input Property of a Component'" [alert-pop]="val" (onSignup)="onSignup($event)"></message-info>
3. </div>

**app.component.ts**

1. **import** { Component } from '@angular/core';
3. @Component({
4. selector: 'app-root',
5. templateUrl: './app.component.html',
6. styleUrls : ['./custom.css']
7. })
8. **export** **class** AppComponent {
10. **public** val: string = "This is alert popup message";
12. **public** onSignup(data:any):**void**{
13. let strMessage:string ="Thanks for Signup " + data.name + ". ";
14. strMessage += "Email id " + data.email + " has been registered successfully.";
15. alert(strMessage);
16. }
17. }

Now refresh the browser to check the output –

