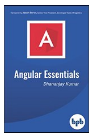


Angular Essentials: The Essential Guide to Learn Angular

by Dhananjay Kumar

BPB Publications. (c) 2019. Copying Prohibited.

Chapter 3: Components Communications

In this chapter, you will learn about Angular Components communication. Following topics will be covered in this chapter:

- Component Communication
- @Input
- @Output and Event Emitter
- Temp Ref Variable

Component Communination

In Angular, components communicate with each other to share data such as object, string, number, array, or html.

To understand component communication, first we need to understand the relationship) betweencomponents. For example, when two components are not related to each other, they communicate through Angular Service.

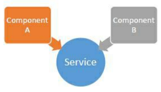
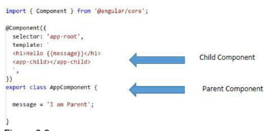


Figure 3.1

When you use a component inside another component, thus creating a component hierarchy, the component being used inside another component is known as the child component and the enclosing component is known as the parent component. As shown in the *figure 3.2*, in context of **`AppComponent,`** app-child is a child component and **`AppComponent`** is a parent component.



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

@Component({
  selector: 'app-root',
  template: `
  <h1>Hello {{message}}</h1>
  <app-child></app-child>
  `,
})
export class AppComponent {

  message = 'I am Parent';

}
```

Figure 3.2

Parent and Child components can communicate with each other in following ways:

- @Input()
- @Output()
- Temp Ref Variable
- ViewChild and ContentChild

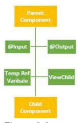


Figure 3.3

When components are not related to each other, they communicate using services. Otherwise,theycommunicate using one of the various options depending on the communication criteria. Let us explore all options one by one.

@Input

You can pass data from parent component to child component using **@ Input** decorator Data could be of any form such as primitive type's string, number, object, array etc.

Parent Component
@Input()
Child Component

Figure 3.4

To understand the use of **@Input,** let us create a component as shown in *code listing 3.1.*

Code Listing 3.1

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

@Component({
   selector:   'app-child',
   template:  `<h2>Hi {{greetMessage}}</h2>"
})
export class AppChildComponent {
   greetMessage =  'I am Child';
}
```

Use **`AppChild`** component inside **`AppComponent`** as shown in *code listing 3.2.*

Code Listing 3.2

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

@Component({
  selector:   'app-root',
  template:  `
  <h1>Hello {{message}}</h1>
  <app-child></app-child>
  `,
})
export class AppComponent {
  message =  'I am Parent';
}
```

`AppComponent` `is using` **`AppChildComponent,`** `hence` **`AppComponent`** `is the parent component and` **`AppChildComponent`** `is the child component. To pass data,` **`@Input`** `decorator uses the child component properties. To do this, we'll need to modify child` **`AppChildComponent`** `as shown in the` *`code listing 3.3.`*

Code Listing 3.3

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

@Component({
   selector:   'app-child',
   template:  `<h2>Hi {{greetMessage}}</h2>"
})
export class AppChildComponent implements OnInit {
   @Input()  greetMessage:  string;
   constructor()   {
   }
   ngOnInit() {
   }
}
```

As you notice, we have modified the **`greetMessage`** property with the **`@Input()`** decorator. So essentially, in the child component, we have decorated the **`greetMessage`** property with the **@Input()** decorator so that value of the **`greetMessage`** property can be set from the parent component. Next, let us modify the parent component **`AppComponent`** to pass data to the child component as shown in *code listing 3.4.*

Code Listing 3.4

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

@Component({
   selector:   'app-root', template:  `
   <h1>Hello {{message}}</h1>
   <appchild  [greetMessage]="childmessage"></appchild>
   `,
})
export class AppComponent {
  message =  'I am Parent';
    childmessage  =   'I   am passed  from  Parent   to  child
component';
}
```

From the parent component, we are setting the value of the child component's property **`greetMessage.`** To pass a value to the child component, we need to pass the child component property inside a square bracket and set its value to any property of parent component. We are passing the value of the **`childmessage`** property from the parent component to the **`greetMessage`** property of the child component.

Intercept input from Parent Component in the Child Component

We may have a requirement to intercept data passed from the parent component inside the child component. This can be done:

1. Using **`@Input`** decorator on getter and setter.
2. Using **`ngOnChanges()`** life cycle hook.

We will discuss about **`ngOnChanges`** life cycle hook in further sections. However, let us see how we can use **`@Input`** with setter to intercept passed data to the child component. We have modified **`AppComponent`** as shown in the *code listing 3.5.*

Code Listing 3.5

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

@Component({
  selector:   'app-root',
  template:  `
  <h1>Hello {{message}}</h1>
 <app-child *ngFor="let n of childNameArray" [Name]="n">
 </app-child>
`,
})
```

```
export class AppComponent {

  message =  'I am Parent';
   childmessage  =   'I   am passed  from  Parent   to  child
component';
  childNameArray =   ['foo',
                     'koo',
                     'moo',
                     ', '
                     'too',
                     'hoo',
                  ''
                        ];
}
```

Inside `AppComponent,` we are looping the `AppChildComponent` through all items of `childNameArray` property. A few items of the `childNameArray` are empty strings, these empty strings would be intercepted by the child component setter and set to the default value.

Let us modify `AppChildComponent` to use `@1nput` decorator with setter and getter as shown in the *code listing 3.6.*

Code Listing 3.6

```
import { Component, Input, OnInit } from '@angular/
core';
@Component({
   selector:   'app-child',
   template:  `<h2> {{_name}}</h2> `
})
export class AppChildComponent implements OnInit {
   _name:  string;
   constructor() {

   }

   ngOnInit()   {
   }
   @Input()
   set Name(name:  string)   {
      this._name =   (name && name.trim()) 'default
name';
      }
      get Name()  {
         return this._name;
      }
}
```

As you notice in the `@Input()` setter, we are intercepting the value passed from the parent, and checking whether it is an empty string. If it is an empty string, we are assigning *the* default value for the name in the child component.

In this way `@Input` can be used to pass data to the child component.

@Output

You can emit event from child component to parent component using @ `Output` decorator.

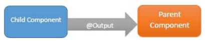


Figure 3.5

Angular is based on a one-directional data flow and does not have two-way data binding. So, we use `@Output` in a component to emit an event to another component. Let us modify `AppChildComponent` as shown in the *code listing 3.7.*

Code Listing 3.7

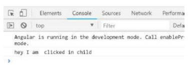
```
import { Component, Input, EventEmitter, Output } from
'@angular/core';

@Component({
   selector:   'app-child',
       template:   "<button   (click)="handleclick()">Click
me</button> `
})
export class AppChildComponent {
   handleclick()   {
      console.log(`hey I am   clicked in child');
   }
}
```

There is a button in the `AppChildComponent` template which is calling the function `handleclick.` Let's use the app-child component inside the `AppComponent` as shown in *code listing 3.8.*

Code Listing 3.8

```
import  {  component,  OnInit  }  from '@angular/core';
@Component({
   selector:   `app-root' ,
   tempi ate:  `<app-child></app-child> `
})
export class AppComponent implements OnInit {
   ngOnInit() {
   }
}
```

Here we're `using AppChildCopmponent inside AppComponent,` *thereby* creating a parent-child kind of relationship, in which `AppComponent` is the parent and `AppChildComponent` is the child. When we run the application with a button click, you'll see this message in the browser console as shown in *figure 3.6.*



```
Angular is running in the development mode. Call enablePr
mode.
hey I am  clicked in child
```

Figure 3.6

So far, it's very simple to use event binding to get the button to call the function in the component. Now, let's tweak the requirement a bit. What if you want to execute a function of `AppComponent` on the click event of a button inside `AppChildComponent?`

To do this, you will have to emit the button click event from `AppChildComponent.` Import `EventEmitter` and Output from **@ angular/core.**

Here we are going to emit an event and pass a parameter to the event. Modify `AppChildComponent` as shown in *code listing 3.9.*

Code Listing 3.9

```
import   {   Component,   EventEmitter,   Output   }   from   `@
angular/core';
@Component({
   selector:   'app-child',
   template:   `<button   (click)="valueChanged()">Click
me</button> '
})
export class AppChildComponent  {
   @Output()  valueChange = new EventEmitter();
   counter = 0;
   valueChanged()   {
      this.counter = this.counter + 1;
      this.valueChange.emit(this.counter);
   }
}
```

Right now, we are performing the following tasks in the `AppChildComponent` class:

- Created a variable called counter, which will be passed as the parameter of the emitted event.
- Created an `EventEmitter valueChange,` which will be emitted to the parent component on the click event of the button.
- Created a function named `valueChanged().` This function is called on the click event of the button, and inside the function event `valueChange` is emitted.
- While emitting `valueChange` event, value of counter is passed as parameter.

In the parent component `AppComponent,` the child component `AppChildComponent` can be used as shown in the *code listing 3.10.*

Code Listing 3.10

```
import  {  Component,  OnInit  }  from '@angular/core';
@Component({
   selector:   'app-root',
                          template:          `<app-child
(valueChange)='displayCounter($event)'></app-child> `
})
export class AppComponent implements OnInit {
   ngOnInit() {
   }
   displayCounter(count)   {
      console.log(count);
   }
}
```

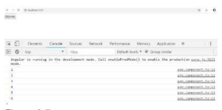
Right now, we are performing the following tasks in the `AppComponent` class:

- Using `<app-child>` in the template.
- In the `<app-child>` element, using event binding to use the `valueChange` event.
- Calling the `displayCounter` function on the `valueChange` event.
- In the `displayCounter` function, printing the value of the counter passed from the `AppChildComponent.`

As you can see, the function of `AppComponent` is called on the click event of the button placed on the `AppChildComponent.` This is can be done with `@Output` and `EventEmitter.` When you run the application and click the button, you can see the value of the counter in the browser console. Each time you click on the button, the counter value is increased by 1 as shown in the *figure 3.7.*

A Real Time Example using @Input and @Output

Let's take a real time example to find how `@Input, @Output,` and `EventEmitter` are more useful. Consider that AppComponent is rendering a list of products in tabular form. To create the product table above, we have a very simple AppComponent class with only one function: to return a list of products.



```
Angular is running in the development mode. Call enableProdMode() to enable the production   core.js:3121
mode.
1                                                                                    app.component.ts:11
2                                                                                    app.component.ts:11
3                                                                                    app.component.ts:11
4                                                                                    app.component.ts:11
5                                                                                    app.component.ts:11
6                                                                                    app.component.ts:11
```

Figure 3.7

Code Listing 3.11

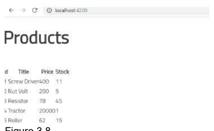
```
export class AppComponent implements OnInit  {
   products =   [];
   title =  'Products';
   ngOnInit() {
      this.products = this.getProducts();
}
getProducts()   {
   return   [
         {   'id':   '1', 'title': 'Screw Driver',
'price':  400,   'stock':  11  },
         {   'id':   '2', 'title': 'Nut Volt',   'price':
200,   'stock':   5   },
         {   'id':   '3', 'title': 'Resistor',   'price':
78,   'stock':   45   },
         {   'id':   '4', 'title':   'Tractor',   'price':
20000,   'stock':   1   },
         {   'id':   '5', 'title': 'Roller',   'price':  62,
'stock':   15   },
      ];
   }
}
```

In the **`ngOnInit`** life cycle hook, we are calling the **`getProducts()`** function and assigning the returned data to the products variable so it can be used on the template. There, we are using the *ngFor directive to iterate through the array and display the products. Template is created as shown in the *code listing 3.12*.

Code Listing 3.12

```
import  {  Component,  OnInit  }  from '@angular/core';
@Component({
   selector:   'app-root',
   template:   `
   <h1 class="text-center">{{title}}</h1>
   <table>
      <thead>
      <th>Id</th>
      <th>Title</th>
      <th>Price</th>
      <th>Stock</th>
      </thead>
      <tbody>
         <tr *ngFor="let p of products">
            <td>{{p.id}}</td>
            <td>{{p.title}}</td>
            <td>{{p.price}}</td> <td>{{p.stock}}</td>
         </tr>
      </tbody>
   </table>
    `
})
export class AppComponent implements OnInit  {
   products =   [];
   title =  'Products';
   ngOnInit() {
      this.products = this.getProducts();
   }
   getProducts()   {
      return   [
         { 'id': '1', 'title': 'Screw Driver', 'price':
400,   'stock':  11  },
         { 'id': '2', 'title': 'Nut Volt', 'price':
200,   'stock':  5  },
         {   'id':   '3',   'title':   'Resistor',   'price':
78,   ' stock' :  45},
         {   °id':   '4',   'title'   'Tractor",   'price':
20000,   'stock':  1  },
         {   'id':   '5',   'title':   'Roller',   'price':  62,
'stock':  15 },
      ];
}
}
```

With this code, products are rendered in a table as shown in the *figure 3.8*.



Products

d	Title	Price	Stock
1	Screw Driver	400	11
2	Nut Volt	200	5
3	Resistor	78	45
4	Tractor	20000	1
5	Roller	62	15

Figure 3.8

Now we have requirements on the above table as follows:

- If the value of stock is more than 10, then the button color should be green.
- If the value of stock is less than 10, then the button color should be red.
- The user can enter a number in the input box, which will be added to that particular stock value.
- The color of the button should be updated on the basis of the changed value of the product stock.

To achieve this task, let us create a new child component called **`StockStatusComponent.`** Essentially, in the template of **`StockStatusComponent,`** there is one button and one numeric input box. In **`StockStatusComponent:`**

- We need to read the value of stock passed from **`AppComponnet.`** For this, we need to use @ **`Input.`**
- We need to emit an event so that a function in **`AppComponent`** can be called on the click of the **`StockStatusComponent.`** For this, we need to use **`@Output`** and **`EventEmitter.`**

`StockStatusComponent` is created as shown in the **code listing 3.13.**

Code Listing 3.13

```
import    {    Component,    Input,    EventEmitter,    Output,
OnChanges  }  from '@angular/core';
@Component({
   selector:   'app-stock-status',
         template:   '<input        type='number'
[(ngModel)]='updatedstockvalue'/> <button class='btn
btn-primary'
   [style.background]='color'
   (click)="stockValueChanged()">Change Stock Value</
button> '
})
export class StockStatusComponent implements OnChanges
{
   @Input()   stock:  number;
   @Input()  productId:  number;
   @Output()   stockValueChange = new EventEmitter(); color =   '';
   updatedstockvalue: number;
   stockValueChanged()   {
      this.stockValueChange.emit({ id: this.productId,
updatdstockvalue: this.updatedstockvalue });
         this.updatedstockvalue = null;
   }
   ngOnChanges()   {
      if   (this.stock > 10)   {
         this.color =  'green'; }
      else {
         this.color =  'red';
      }
   }
}
```

Let's explore the above class line by line.

- In the first line we are importing everything required: **`@Input,`** **@** **`Output`** etc.
- In the template, there is one numeric input box which is bound to the **`updatedStockValue`** property using **`[(ngModel)].`** We need to pass this value with an event to the **AppComponent.**
- In the template, there is one button. On the click event of the button, an event is emitted to the **`AppComponent.`**
- We need to set the color of the button on the basis of the value of product stock. So, we must use property binding to set the background of the button. The value of the color property is updated in the class.
- We are creating two **`@Input()`** decorated properties - stock and productId - because value of these two properties will be passed from **`AppComponent`**.
- We are creating an event called **`stockValueChange.`** This event will be emitted to **`AppComponent`** on the click of the button.
- In the **`stockValueChanged`** function, we are emitting the **`stockValueChange`** event and also passing the product id to be updated and the value to be added in the product stock value.
- We are updating the value of color property in the **`ngOnChanges()`** life cycle hook because each time the stock value gets updated in the **`AppComponent,`** the value of the color property should be updated.

Here we are using the **`@Input`** decorator to read data from **`AppComponent`** class, which happens to be the parent class in this case. So, to pass data from the parent component class to the child component class, use **@** **`Input`** decorator.

In addition, we are using **`@Output`** with **`EventEmitter`** to emit an event to **`AppComponent.`** So to emit an event from the child component class to the parent component class, use **`EventEmitter`** with **`@Output()`** decorator.

Therefore, **`StockStatusComponent`** is using both **`@Input`** and **@** **`Output`** to read data from **`AppComponent`** and emit an event to **`AppComponent`**.

Let us first modify the template. In the template, add a new table column. Inside the column, the **`<app-stock-status>`** component is used.

Code Listing 3.14

```
<h1 class="text-center">{{title}}</h1>
      <table>
         <thead>
         <th>Id</th>
         <th>Title</th>
         <th>Price</th>
         <th>Stock</th>
         </thead>
         <tbody>
            <tr *ngFor="let p of products">
            <td>{{p.id}}</td>
            <td>{{p.title}}</td>
            <td>{{p.price}}</td>
            <td>{{p.stock}}</td>
            <td><app-stock-status  [productId]='p.id'
               [stock]='p.stock'
         (stockValueChange)='changeStockValue($event)'>
               </app-stock-status>
            </td>
            </tr>
      </tbody>
   </table>
```

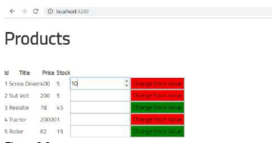
We are passing the value to `productId` and stock using property binding (remember, these two properties are decorated with `@Input()` in `StockStatusComponent)` and using event binding to handle the `stockValueChange` event (remember, this event is decorated with **@** `Output()` in `StockStatusComponent)`.

Next, we need to add `changeStockValue` function in the `AppComponent`. Add the code as shown in listing 3.15 in the `AppComponent` class:

Code Listing 3.15

```
productToUpdate:  any;
   changeStockValue(p)   {
         this.productToUpdate   =   this.products.find(this.
findProducts,    [p.id]);
      this.productToUpdate.
stock = this.productToUpdate. stock + p.updatdstockvalue;
   }
   findProducts(p)   {
      return p.id = = = this [0] ;
   }
```

In the function, we are *using* the JavaScript ***Array.***`prototype.find` method to find a product with a matched `productId` and then updating the stock count of the matched product. When you run the application, you'll get the following output as shown in the *figure 3.9*.



localhost:4200

Products

Id	Title	Price	Stock		
1	Screw Driver	400	5	10	Change Stock Value
2	Nut Volt	200	5		Change Stock Value
3	Resistor	78	45		Change Stock Value
4	Tractor	20000	1		Change Stock Value
5	Roller	62	15		Change Stock Value

Figure 3.9

When you enter a number in the numeric box and click on the button, you prform a task in the child component that updates the operation value in the parent component. Also, on the basis of the parent component value, the style is being changed in the child component. All this is possible using Angular `@Input`, `@Output`, and **`EventEmitter.`**

Temp Ref Variable

Temp Ref Variable is used to read properties or call methods of child component in the template of parent component.

Child Component
Temp Ref Variable
Parent Component

Figure 3.10

It is used to access properties and methods of Child Component inside the template of Parent Component. Let us consider child component as shown in the *code listing 3.16.*

Code Listing 3.16

```
export class AppChildComponent  {
   message =  'I am Child';
   sayChildHello()   {
      console.log('I am clicked in the child');
   }
}
```

You want to use message property and `sayChildHello` function of `AppChildComponent` on the template of parent component; you can do that using Tem Ref Variable as shown in the *code listing 3.17.*

Code Listing 3.17

```
import  {  Component,  OnInit  }  from '@angular/core';
@Component({
   selector:   'app-root',
   template:  `
      <app-child #childtemp></app-child>
      <h2>{{childtemp.message}}</h2>
      <button   (click)='childtemp.sayChildHello()'>call
child function</button>
})
        `
export class AppComponent implements OnInit {
   ngOnInit() {
   }
}
```

To use **Temp Ref** Variable give a name to child component using #. Here we gave name `childtemp`. Using this name you can use any properties or methods using dot in the template of parent component.

ViewChild and ContentChild

ViewChild or Content Child is used to read properties or call methods of child component in the class of parent component.

Child Component
ViewChild
Parent Component

Figure 3.11

In further chapter, we will cover these topics with content projection in detail.

Summary

Understanding of communication between components is essential. In real time applications, you always send data between components. In this chapter, we learnt about following topics:

- @Input
- @Output
- Temp Ref Variable