DZone

Understanding @Output and EventEmitter in Angular

by Dhananjay Kumar MVB · Jan. 05, 18 · Web Dev Zone · Tutorial

In Angular, a component can emit an event using **@Output** and **EventEmitter**. Both are parts of the **@angular/core**.

Confused by the jargon? Let's simplify it together. Consider the **AppChildComponent** as shown below:

appchild.component.ts

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

@Component({
    selector: 'app-child',
    template: `<button class='btn btn-primary' (click)="handleclick()">Click me</butt

})
export class AppChildComponent {

    handleclick() {

        console.log('hey I am  clicked in child');
    }
}
```

There's a button in the AppChildComponent template which is calling the function handleclick. Let's use the app-child component inside the App Compontent as shown below:

appcomponent.ts

```
import { Component, OnInit } from '@angular/core';
@Component({
    selector: 'app-root',
```

```
    template: `<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:

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. Consider the code below:

appchild.component.ts

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

@Component({
    selector: 'app-child',
    template: `<button class='btn btn-primary' (click)="valueChanged()">Click me</but

})
export class AppChildComponent {

    @Output() valueChange = new EventEmitter();
    Counter = 0;

    valueChanged() { // You can give any function name

        this.counter = this.counter + 1;
        this.valueChange.emit(this.counter);
```

```
    }
}
```

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

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

In the parent component, **AppComponent**, the child component, **AppChildComponent,** can be used as shown in the code below:

appcomponent.ts

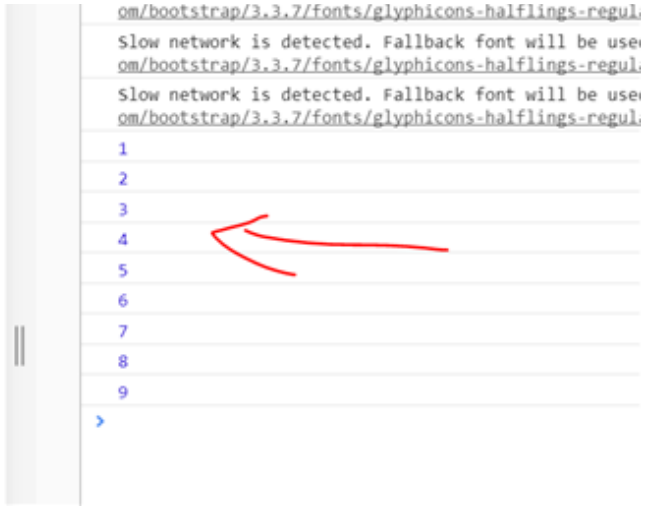
```
import { Component, OnInit } from '@angular/core';
@Component({
    selector: 'app-root',
    template: `<app-child (valueChange)='displayCounter($event)'></app-child>`
})
export class AppComponent implements OnInit {
    ngOnInit() {

    }
    displayCounter(count) {
        console.log(count);
    }
}
```

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

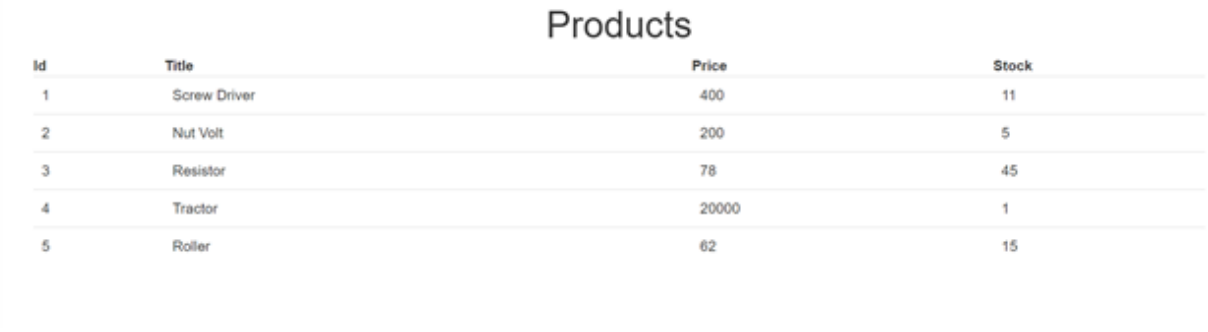
1. Using <app-child> in the template.
2. In the <app-child> element, using event binding to use the `valueChange` event.
3. Calling the `displayCounter` function on the `valueChange` event.
4. In the `displayCounter` function, printing the value of the counter passed from the AppChildComponent.

As you can see, the function **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.



A Real-Time Example

Let's take a real-time example to find out how @Output and EventEmitter can be more useful. Consider that **AppComponent** is rendering a list of products in a tabular form as shown in the image below:



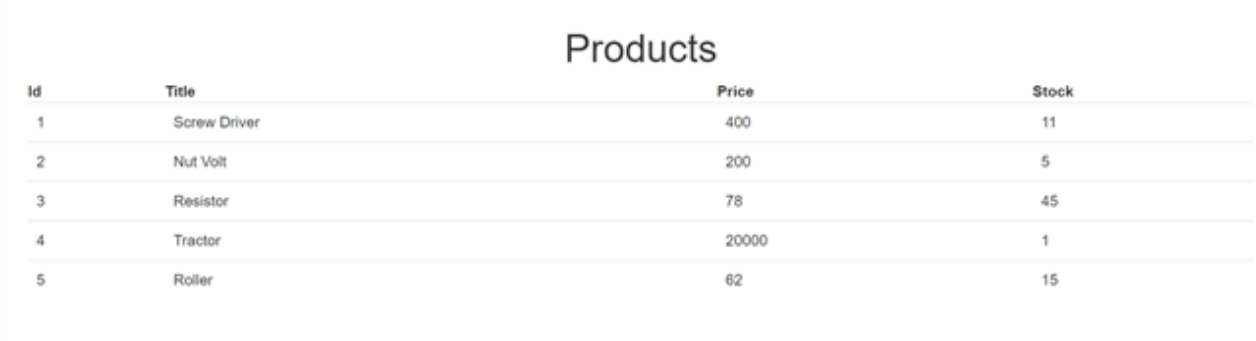
To create the product table above, we have a very simple **AppComponent** class with only one function: to return a list of products.

```
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 },
        ];
    }
}
```

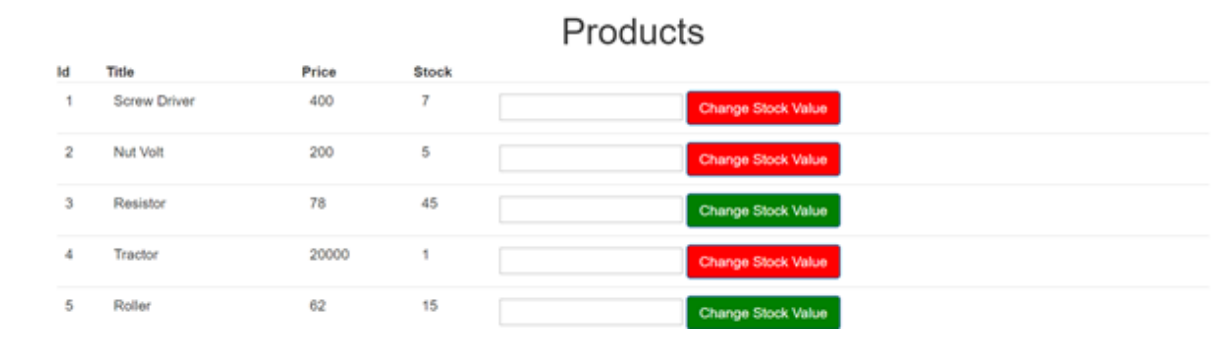
In the ngOnInit lifecycle hook, we are calling the `getPrdoducts()` function and assigning the returned data to the products variable so it can be used in the template. There, we are using the *ngFor directive to iterate through the array and display the products. See the code below:

```
<div class="container">
    <br />
    <h1 class="text-center">{{title}}</h1>
    <table class="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>
</div>
```

With this code, products are rendered in a table as shown in the image below:



Now let's say we want to add a new column with a button and input box as shown in the image below:



Our requirements are as follows:

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

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

1. We need to read the value of **stock** passed from **AppComponnet**. For this, we need to use @Input
2. 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.

Consider the code below:

stockstatus.component.ts

```
    import { Component, Input, EventEmitter, Output, OnChanges } from '@angular/core';
1
2   @Component({
3       selector: 'app-stock-status',
        template: `<input type='number' [(ngModel)]='updatedstockvalue'/> <button class='
4
5        [style.background]='color'
6        (click)="stockValueChanged()">Change Stock Value</button> `
7   })
8   export class StockStatusComponent implements OnChanges {
9
10      @Input() stock: number;
11      @Input() productId: number;
12      @Output() stockValueChange = new EventEmitter();
13      color = '';
14      updatedstockvalue: number;
15      stockValueChanged() {
            this.stockValueChange.emit({ id: this.productId, updatdstockvalue: this.updat
16
17          this.updatedstockvalue = null;
18      }
19      ngOnChanges() {
20
21          if (this.stock > 10) {
22              this.color = 'green';
```

```
        } else {
            this.color = 'red';
        }
    }
}
```

Let's explore the above class line by line.

1. In the first line, we are importing everything that's required: @Input, @Output, etc.
2. 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**.
3. In the template, there is one button. On the click event of the button, an event is emitted to the **AppComponent**.
4. 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.
5. We are creating two `@Input()` decorated properties - **stock** and **productId** - because the value of these two properties will be passed from **AppComponent**.
6. We are creating an event called **stockValueChange**. This event will be emitted to **AppComponent** on the click of the button.
7. 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 to the product stock value.
8. We are updating the value of the color property in the `ngOnChanges()` lifecycle 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 the @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 the `@Output()` decorator.

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

Modify AppComponent to Use StockStatusComponent

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

```
<div class="container">
    <br />
    <h1 class="text-center">{{title}}</h1>
    <table class="table">
        <thead>
```

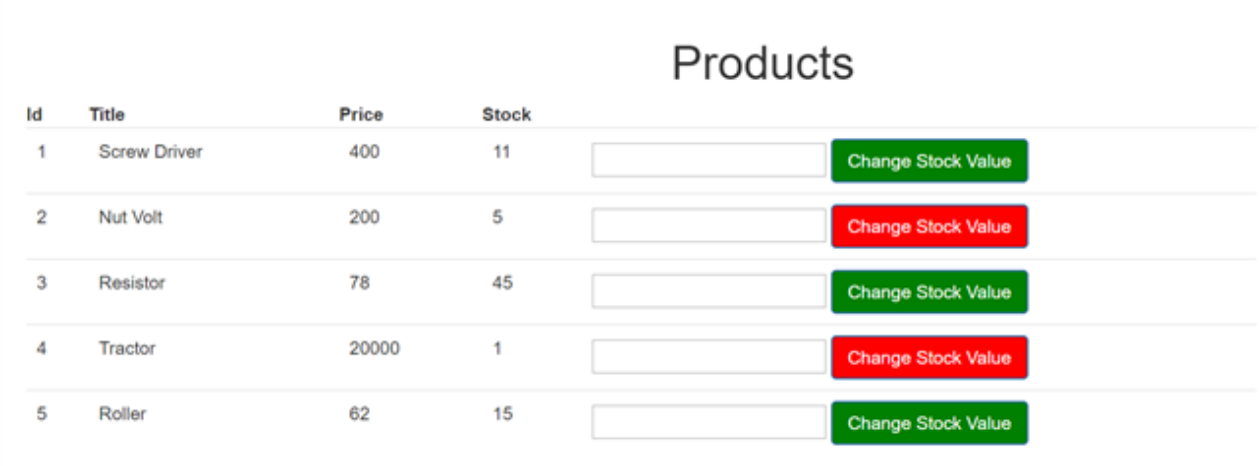
```
        <th>Id</th>
        <th>Title</th>
        <th>Price</th>
        <th>Stock</th>
        </thead>
        <tbody>
            <tr *ngFor="let p of products">
                <td>{{p.id}}</td>
                <td>{{p.title}}</td>
                <td>{{p.price}}</td>
                <td>{{p.stock}}</td>
                <td><app-stock-status [productId]='p.id' [stock]='p.stock' (stockValu

            </tr>
        </tbody>
    </table>
</div>
```

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**).

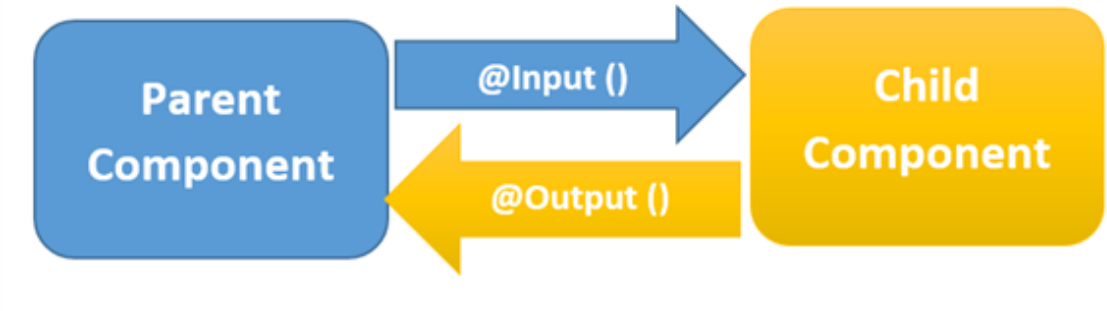
```
productToUpdate: any;
changeStockValue(p) {
    this.productToUpdate = this.products.find(this.findProducts, [p.id]);
    this.productToUpdate.stock = this.productToUpdate.stock + p.updatdstockvalue;
}
findProducts(p) {
    return p.id === this[0];
}
```

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:



When you enter a number in the numeric box and click on the button, you perform 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.

In summary:



Stay tuned for future articles where we go into more depth on other features of Angular!

Like This Article? Read More From DZone

Real-World Angular Series, Part 2a: Authentication

Creating a Front-End for Your User Profile Store With Angular and TypeScript

Why You Should Use TypeScript for Developing Web Applications

Free DZone Refcard

Getting Started With Java-Based CMS

Topics: ANGULAR , WEB DEV , WEB APPLICATION DEVELOPMENT , TYPESCRIPT

Published at DZone with permission of Dhananjay Kumar , DZone MVB. See the original article here.