Angular @Input, @Output & EventEmitter

15 Comments / March 9, 2023 / 7 minutes of reading

⟵ ng-content & Content Projection   Angular Tutorial   Template Reference Variable ⟶

In this guide let us learn how to make use of `@input` , `@output` & `EventEmitter` in Angular. We use these decorators to pass data from parent to child component & vice versa. `@Input` defines the input property in the component, which the parent component can set. The `@output` defines the output property (event), which we raise in the child component using the `EventEmitter` . The parent listens to these events.

Table of Contents

@input

Input decorator marks the property as the input property. I.e it can receive data from the parent component. The parent component uses the property binding to bind it to a component property. Whenever the value in the parent component changes angular updates the value in the child component.

Example

Consider the following component class

```

@Component({
  selector: 'app-customer-detail',
  templateUrl: './customer-detail.component.html',
  styleUrls: ['./customer-detail.component.css']
})
export class CustomerDetailComponent implements OnInit {
  @Input() customer:Customer;
}

```

We have Input decorator on the customer property. The component expects that the parent component will supply its value.

```html

<app-customer-detail [customer]="selectedCustomer"></app-customer-detail>

```

@output

`Output` decorates the property as the output property. We initialize it as an `EventEmitter`. The child component raises the event and passes the data as the argument to the event. The parent component listens to events using event binding and reads the data.

Example

```typescript

//Declare the property
@Output() customerChange:EventEmitter<Customer> =new EventEmitter<Customer>();

//Raise the event to send the data back to parent
update() {
  this.customerChange.emit(this.customer);
}

```

The `customerChange` is the Output property and is of type `EventEmitter`.

```
<app-customer-detail [customer]="selectedCustomer" (customerChange)="update($event)"></app-customer-detail>
```

Remember you must use the argument name as `$event`.

EventEmitter

`EventEmitter` is responsible for raising the event. The `@output` property normally is of type `EventEmitter`. The child component will use the `emit()` method to emit an event along with the data.

```
//Define  output property
@Output() customerChange:EventEmitter<Customer> =new EventEmitter<Customer>();

//Raise the event using the emit method.
update() {
  this.customerChange.emit(this.customer);
}
```

Now let us build an app to learn how to use `Input`, `output` & `EventEmitter`

@input, @output & Eventemitter Example

The app we build has two components. The parent component shows a list of customers. The user has the option to click on the edit button, which results in a child component displaying the customer form Once the user updates the records, the child component raises the event. The parent captures the event. The parent then updates the list with the new data.

Create a new application using the following command

```

ng new InputOutputExample


cd InputOutputExample

```

```
ng g c customerList
ng g c customerDetail
ng g class customer
```

Customer

```
export class Customer {

  customerNo: number=0;
  name: string="";
  address: string="";
  city: string="";
  state: string="";
  country: string="";

}
```

app.module.ts

The `ngModel` needs the `FormsModule`. Hence import it and add it in import metadata.

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { CustomerListComponent } from './customer-list/customer-list.component';
import { CustomerDetailComponent } from './customer-detail/customer-detail.component';

@NgModule({
  declarations: [
    AppComponent,
    CustomerListComponent,
    CustomerDetailComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent],
})
export class AppModule { }


```

Child Component

The child component gets an instance of the customer in its input property `customer`. The parent needs to set it using the property binding

Users can edit the customer. Once finished they will click the update button. The update method raises the customerChange event. We pass the customer as the argument to the event. The parent component listens to the event and receives the data.

The following is the complete code of the CustomerDetailComponent.

```

import { Component, OnInit, Input, Output,EventEmitter } from '@angular/core';
import { Customer } from '../customer';

@Component({
  selector: 'app-customer-detail',
  templateUrl: './customer-detail.component.html',
  styleUrls: ['./customer-detail.component.css']
})
export class CustomerDetailComponent implements OnInit {

  @Input() customer:Customer = new Customer();
  @Output() customerChange:EventEmitter<Customer> =new EventEmitter<Customer>();

  constructor() { }

  ngOnInit() {
  }

  update() {
    this.customerChange.emit(this.customer);
  }

```

'app-customer-detail' is the name of the selector for this component.

The customer property is the input property decorated with Input.

```
@Input() customer:Customer = new Customer();
```

customerChange is decorated as the output property of type EventEmitter

```
@Output() customerChange:EventEmitter<Customer> =new EventEmitter<Customer>();
```

Whenever the user updates the customer, we raise the event customerChange. We pass the updated customer as the argument to it.

```
update() {
  this.customerChange.emit(this.customer);
}
```

```

<p>Customer No : {{customer.customerNo}}</p>
<p>Name        : <input [(ngModel)]="customer.name"></p>
<p>Address     : <input [(ngModel)]="customer.address"></p>
<p>city     : <input [(ngModel)]="customer.city"></p>
<p>state     : <input [(ngModel)]="customer.state"></p>
<p>country     : <input [(ngModel)]="customer.country"></p>

<button (click)="update()">Update</button>

```

The ngModel binds the `customer` to the input element. It is a two-way binding. The `click` event of the button is bound to `update()` method in the component.

Parent Component

The job of the parent component is to display a list of customers. When the user clicks on the edit button pass the selected customer to the child component. Then wait for the `customerChange` event. Update the customer's list on receipt of data from the child.

The following is the `customer-list.component.html`

```

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

```
    <tr>
      <th>No</th>
      <th>Name</th>
      <th>Address</th>
      <th>City</th>
      <th>State</th>
      <th>Country</th>
      <th>Edit</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>
      <td>{{customer.country}}</td>
      <td><button (click)="showDetails(customer)">Edit</button></td>
    </tr>
  </tbody>
</table>

<h3>Details</h3>
<app-customer-detail [customer]="selectedCustomer" (customerChange)="update($event)"></app-customer-detail>

```

Use the ngFor directive to loop through the customer list and display the customer details.

The event binding to capture the click event. We pass the customer object to the showDetails method

```
<td><button (click)="showDetails(customer)">Edit</button></td>
```

app-customer-detail is the selector for the CustomerDetailComponent. We use the property binding to send the selectedCustomer to the child component. The child component raises the customerChange event, which we listen to using the event binding and call the update method.

Customer-list.component.ts

The component code of the parent component. It has two method showDetails & update

```
import { Component, OnInit } from '@angular/core';
import { Customer } from '../customer';
import { element } from 'protractor';
import { ObjectUnsubscribedError } from 'rxjs';

@Component({
  selector: 'app-customer-list',
  templateUrl: './customer-list.component.html',
  customers: Customer[] = [

    {customerNo: 1, name: 'Rahuld Dravid', address: '', city: 'Banglaore', state: 'Karnataka', country: 'India'},
    {customerNo: 2, name: 'Sachin Tendulkar', address: '', city: 'Mumbai', state: 'Maharastra', country: 'India'},
    {customerNo: 3, name: 'Saurrav Ganguly', address: '', city: 'Kolkata', state: 'West Bengal', country: 'India'},
    {customerNo: 4, name: 'Mahendra Singh Dhoni', address: '', city: 'Ranchi', state: 'Bihar', country: 'India'},
    {customerNo: 5, name: 'Virat Kohli', address: '', city: 'Delhi', state: 'Delhi', country: 'India'},

  ]

  selectedCustomer:Customer = new Customer();

  constructor() { }

  ngOnInit() {
  }

  showDetails(customer:Customer) {
    this.selectedCustomer=Object.assign({},customer)
  }

  update(customer:Customer) {
    console.log(customer)
    var cust=this.customers.find(e => e.customerNo==customer.customerNo)
    Object.assign(cust,customer)
    alert("Customer Saved")
  }
}
```

Since the customer is an object it is **Passed by Reference** . When you make any modification to the `customer` it will also be reflected in the customer's collection. We want the update the customer's only when we get the data from the child. Hence we clone the customer and send it to the child component.

If you are passing primitive data types like numbers are **Passed by Value** .

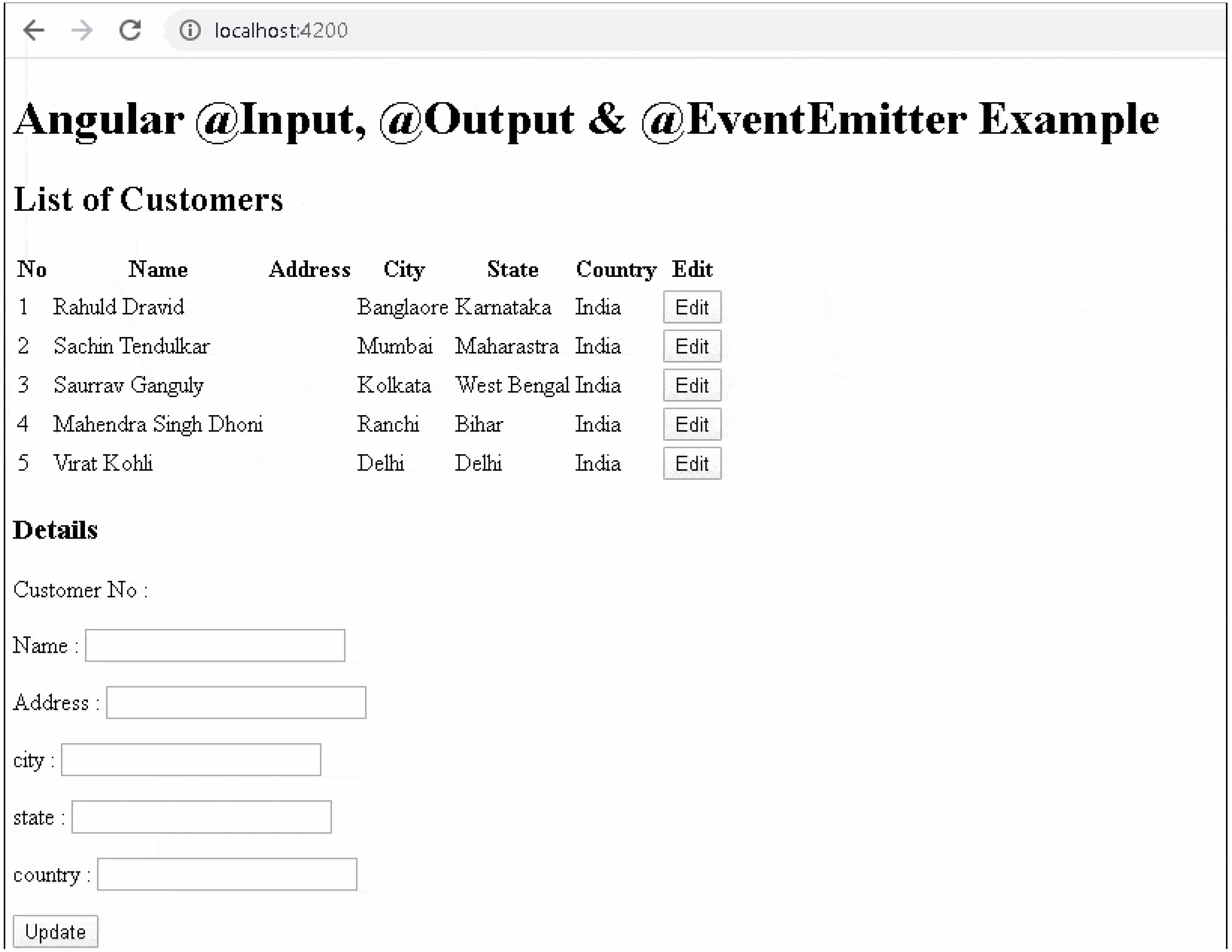
Finally in the root component (i.e. `app.component.html`) copy the following

```

<app-customer-list></app-customer-list>

```

Run the app



Angular @Input, @Output & @EventEmitter Example

List of Customers

No	Name	Address	City	State	Country	Edit
1	Rahuld Dravid		Banglaore	Karnataka	India	Edit
2	Sachin Tendulkar		Mumbai	Maharastra	India	Edit
3	Saurrav Ganguly		Kolkata	West Bengal	India	Edit
4	Mahendra Singh Dhoni		Ranchi	Bihar	India	Edit
5	Virat Kohli		Delhi	Delhi	India	Edit

Details

Customer No :

Name :

Address :

city :

state :

country :

You can also pass the optional name

`Input` decorator allows us to pass an option name, which you can use it while binding in the parent

For Example

```

@Input(‘customerData’) customer:Customer;

```

Intercept input property changes with a setter

You can also create a setter property

```

private _customerData = '';
@Input()
set customer(customer: Customer) {
  //You can add some custom logic here
  this._customerData = customer;
  console.log(this._customerData)
}
get customer(): string { return this._customerData; }

```

You can also subscribe to the changes using ngOnChanges life cycle hook.

EventEmitters are observable

`EventEmitters` are RxJs Subjects. Hence you can make use of RxJs operators to manipulate them. Read more about it from this link.

Pass by reference

The objects are `passed by reference`. Hence if you modify the object, you are updating the original object. The primitive data types like numbers are **Passed by Value**.

References

- https://angular.io/api/core/Input
- https://angular.io/api/core/Output
- https://angular.io/api/core/EventEmitter
- https://angular.io/guide/component-interaction

Read More

- Angular Observable
- Pass data to child component
- ngOnchanges
- Components
- Two-way binding
- Property Binding
- Event Binding
- Child / Nested Components
- Subjects in Angular

Summary

This article shows how to make use of Input, output & EventEmitter in Angular. We use them to communicate with parent & child components. The Child component defines the input & output property using @Input & @output decorators. The Parent sets the input property using property binding. The child component raises the event using EventEmitter whenever it wants to send data to the parent. The parent listens to output property using event binding.

← ng-content & Content Projection   Angular Tutorial   Template Reference Variable →

ARUN

OCTOBER 21, 2022 AT 10:28 PM

Object.assign(cust,customer)

this line giving error, update not working

Reply

ANONYMOUS

JANUARY 14, 2023 AT 11:55 PM

this may works Object.assign(cust!, customer) . used ! mark after the cust variable name.

Reply

TANVEER

JANUARY 27, 2022 AT 5:25 PM

Reply

ANONYMOUS

JUNE 20, 2021 AT 5:00 AM

why my model changed before press update button?

Reply

ARUN PRASAD

APRIL 28, 2021 AT 10:58 AM

Few things missed here.

ng n c customerList use ng g c customerList.

Ignore this error “ERROR TypeError: Cannot read property ‘customerNo’ of undefined”, once we start edit this will be cleared.

Thanks

Reply

ANONYMOUS
APRIL 28, 2021 AT 11:15 AM

add => app.list.component in app.component.html.

Reply

ANONYMOUS
APRIL 30, 2021 AT 6:52 AM

You can get rid of the “ERROR TypeError:….” by defining a blank value for selectedCustomer in customer-list.component.ts. For example:
ngOnInit() {

this.showDetails(myCustomer);
}

Reply

ANONYMOUS

APRIL 30, 2021 AT 6:54 AM

Sorry, should be
let myCustomer = {customerNo:0, … etc.

ANONYMOUS

APRIL 30, 2021 AT 7:02 AM

Sorry again, The reply editor won't let me type angle brackets. So the let statement should be myCustomer = angleBracket Customer angleBracket{ customerNo:0, …etc.

Thanks, super tutorial!!

Reply

TEKTUTORIALSHUB

MAY 1, 2021 AT 8:50 AM

Thanks Updated the Article

Reply

ANONYMOUS

JANUARY 21, 2021 AT 10:39 PM

this is a godsend! thank you for the detailed tutorial

Reply

RICHARD

NOVEMBER 25, 2020 AT 9:54 PM

This is so clearly written and I particularly appreciate the followup links at the end. Thanks, well done!!

Reply

MICHAEL WASSERMANN

NOVEMBER 12, 2020 AT 3:44 AM

Have anything on dynamic components?

Reply

yes

Reply

Leave a Comment

Your email address will not be published. Required fields are marked *

Type here..

Post Comment »

This site uses Akismet to reduce spam. Learn how your comment data is processed.