

# Semana 8

## Curso de Angular



**Semana 8**  
**Angular Bootstrap. Angular**  
**Material. Google Maps.**  
**Angular Charts.**

Desarrollo de aplicaciones web con Angular  
Cefire 2017/2018

Autor: Arturo Bernal Mayordomo

## Index

|  |    |
|--|----|
| Angular Bootstrap.....                           | 3  |
| Angular material 2 + flex layout.....            | 6  |
| Soporte para pantallas táctiles.....             | 6  |
| Iconos.....                                      | 6  |
| Cargando un tema.....                            | 7  |
| Creando una estructura básica (flex-layout)..... | 7  |
| Mostrando una ventana modal.....                 | 9  |
| Integración con Google Maps.....                 | 11 |
| Google Places Autocomplete.....                  | 12 |
| Angular Charts (ngx-charts).....                 | 14 |

# Angular Bootstrap

---

Bootstrap es una herramienta de diseño CSS/JavaScript para crear diseños web adaptables muy conocida. La parte de JavaScript de Bootstrap está implementada usando JQuery, pero puede ser sustituida por otras implementaciones como ng-bootstrap (Angular): <https://ng-bootstrap.github.io>

ng-bootstrap contiene varios servicios y directivas de Angular. Como pequeña introducción, vamos a ver como mostrar una **ventana modal**. Tendrá 2 botones (sí, no), y la respuesta (true/false) se devolverá dentro de una **promesa**. Combinaremos esto con el CanDeactivate guard que creamos (ya que puede devolver boolean, Promise<boolean>, Observable<boolean>).

Lo primero de todo será instalar el plugin de Angular:

```
npm i @ng-bootstrap/ng-bootstrap
```

Para registrar los servicios de ng-bootstrap en nuestra aplicación, importamos el módulo **NgbModule** (llamando al método **forRoot()** que exportará los servicios) en el módulo de la aplicación. Otros módulos que lo requieran también lo deben importar (sin llamar a **forRoot()**).

```
import { NgbModule } from '@ng-bootstrap/ng-bootstrap';
...
@NgModule({
  ...
  imports: [
    ...
    NgbModule.forRoot(),
  ],
  ...
})
export class AppModule { }
```

Vamos a mostrar una [Ventana Modal](#) en la ruta **product/edit** cuando el usuario intente abandonarla. Hasta ahora hemos usado una ventana **confirm**, pero a nivel de diseño no es lo más recomendable.

Como esta ventana modal la queremos poder usar desde cualquier parte de nuestra aplicación (cualquier módulo), lo mejor es crear el componente que representa el contenido de dicha ventana en un módulo compartido. Además vamos a aprovechar para mover la directiva **minDate** que creamos la semana anterior a dicho módulo también.

```
ng g module shared
```

Desde la carpeta del módulo que acabamos de crear:

```
ng g component confirm-modal
```

Como este componente no se va a usar en ninguna ruta, ni tampoco se usará su selector **<confirm-modal>** en ninguna plantilla, Angular no lo compilará ni lo incluirá

por defecto. Para forzar dicha inclusión en estos casos, debemos poner el componente en el array **entryComponents**.

```
@NgModule({
  imports: [
    CommonModule,
    NgbModule
  ],
  declarations: [
    MinDateDirective,
    ConfirmModalComponent
  ],
  entryComponents: [
    ConfirmModalComponent
  ],
  exports: [
    MinDateDirective,
    ConfirmModalComponent
  ]
})
export class SharedModule { }
```

En el componente que acabamos de crear, vamos a añadir 2 valores de entrada. El título de la ventana modal y el texto del cuerpo. Además, necesitamos inyectar el servicio NgbActiveModal para manipular la ventana modal (y cerrarla por ejemplo).

```
@Component({
  selector: 'confirm-modal',
  templateUrl: './confirm-modal.component.html',
  styleUrls: ['./confirm-modal.component.css']
})
export class ConfirmModalComponent implements OnInit {
  @Input() title: string;
  @Input() body: string;

  constructor(public activeModal: NgbActiveModal) {}

  ngOnInit() {}
}
```

Y esta es la plantilla del componente. Fíjate que para cerrar la ventana llamamos a dismiss o a close. El método dismiss sirve para cancelar (rechaza la promesa que devuelve la ventana → error), mientras que close devuelve un valor válido dentro de la misma (en este caso true o false).

```
<div class="modal-header">
  <h4 class="modal-title">{{title}}</h4>
  <button type="button" class="close" aria-label="Close"
    (click)="activeModal.dismiss()">
    <span aria-hidden="true">&times;</span>
  </button>
</div>
<div class="modal-body">
  <p>{{body}}</p>
</div>
<div class="modal-footer">
  <button type="button" class="btn btn-success"
    (click)="activeModal.close(true)">Yes</button>
  <button type="button" class="btn btn-danger"
    (click)="activeModal.close(false)">No</button>
</div>
```

Ahora, desde el guard **LeavePage**, o lo que es lo mismo en este caso, el método **canDeactivate** del componente **product-edit**, en lugar de la función **confirm**, vamos a lanzar una ventana modal con el componente que acabamos de crear (debemos inyectar el servicio **NgbModal**). Esta ventana ya devuelve el booleano necesario dentro de una promesa (en función del botón pulsado). Si cerramos la ventana con **dismiss**, la promesa se rechaza, por lo que hay que capturar dicho error con **catch**, y devolver **false** (no abandonamos la página actual).

Si quisiéramos trabajar con observables, podríamos transformar una promesa en un Observable usando **Observable.fromPromise(promesa)**.

```
import { NgbModal } from '@ng-bootstrap/ng-bootstrap';
import { ConfirmModalComponent } from '../shared/confirm-modal/confirm-modal.component';

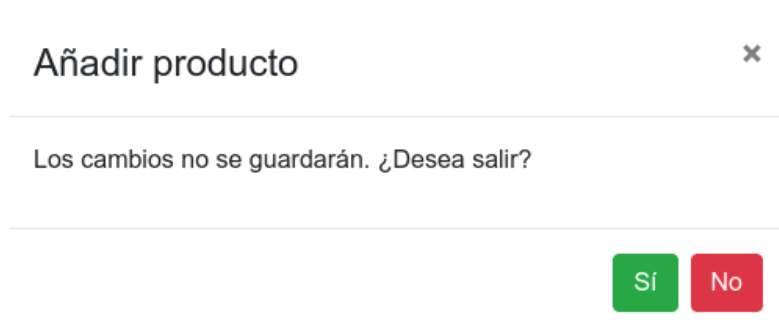
@Component({
  selector: 'product-edit',
  templateUrl: './product-edit.component.html',
  styleUrls: ['./product-edit.component.css']
})
export class ProductEditComponent implements OnInit, ComponentDeactivate {
  ...
  constructor(
    private route: ActivatedRoute,
    private productsService: ProductsService,
    private modalService: NgbModal,
    private router: Router
  ) {}

  ...

  canDeactivate(): Promise<boolean> {
    if (this.actualizado) { // Si acabamos de añadir producto -> Promise<true>
      return new Promise((resolve, reject) => resolve(true));
    }

    const modalRef = this.modalService.open(ConfirmModalComponent);
    modalRef.componentInstance.title = 'Añadir producto';
    modalRef.componentInstance.body = 'Los cambios no se guardarán. ¿Desea salir?';
    return modalRef.result // Cuando cerramos con close -> booleano recibido
      .catch(() => false); // dismiss -> Promise<false>
  }

  ...
}
```



# Angular material 2 + flex layout

---

Otra elección lógica en cuanto a diseño en una aplicación Angular sería [Angular Material](#). Son una serie de componentes que cumplen con la especificación de [Google Material Design](#). Desde la versión 6, el desarrollo de Angular Material se integra mejor con el desarrollo de Angular, usando el mismo número de versión. Hay una aplicación de ejemplo mostrando algunos componentes [aquí](#) ([github](#)).

Para obtener una gestión de posicionamiento y columnas como tiene Bootstrap, necesitas instalar [Angular Flex Layout](#) (tiene más posibilidades que Bootstrap). Además, Angular Material usa el módulo de animaciones de Angular, así que al final instalaremos lo siguiente (@angular/cdk es parte de Material):

```
npm i @angular/material @angular/cdk @angular/animations @angular/flex-layout@6.0.0-beta.16
```

Empezamos importando los módulos de Flex Layout y Animations:

```
@NgModule({
  ...
  imports: [
    ...
    FlexLayoutModule,
    BrowserAnimationsModule
  ],
  ...
})
export class AppModule { }
```

Cada componente de Angular Material tiene su propio módulo (así sólo se incluye en la compilación final lo que necesitamos y no toda la librería). Debemos importar un módulo por cada componente que queramos usar. Por ejemplo, para usar botones importamos `MatButtonModule`, para checkbox `MatCheckboxModule`. Puedes consultar los componentes disponibles [aquí](#).

Una buena idea sería crear un módulo (compartido) que importara (y exportara) todos los módulos de Angular Material que vamos a necesitar en nuestra aplicación. Así sólo tendríamos que importar dicho módulo para tener acceso a todo.

## Soporte para pantallas táctiles

Algunos componentes como los sliders o tooltips usan gestos táctiles (desplazamiento o swipe, etc.). Para habilitar este soporte instalamos HammerJS.

```
npm i hammerjs
```

Cargamos la librería desde **app/main.ts** de la siguiente forma.

```
import 'hammerjs';
```

## Iconos

Por defecto, Angular Material usa la fuente [Material icons](#) para mostrar iconos.

Si quieres usar esta fuente debes añadir lo siguiente a **index.html**:

```
<link href="https://fonts.googleapis.com/icon?family=Material+Icons"
      rel="stylesheet">
```

Ahora, sólo hace falta poner el nombre del icono entre la etiqueta **<mat-icon>**:

```
<mat-icon>home</mat-icon>
```



Si prefieres usar otra fuente como Font Awesome, instálala primero (**npm install font-awesome**), e importa el CSS en angular-cli.json:

```
"styles": [
  "styles.css",
  "../node_modules/font-awesome/css/font-awesome.css"
],
```

Ahora puedes crear iconos con esta fuente de la siguiente manera:

```
<mat-icon fontSet="fa" fontIcon="fa-home"></mat-icon>
```

Para otras opciones consulta [aquí](#). Font Awesome tiene una versión que se integra con Angular de forma independiente sin Material ([angular-font-awesome](#)).

## Cargando un tema

Angular Material viene con una serie de temas de estilo. Se puede crear un tema propio, pero está fuera del ámbito del presente curso. Para usar un tema predefinido, sólo necesitas cargar uno de estos archivos CSS en tu archivo styles.css (o angular-cli.json):

```
"~@angular/material/prebuilt-themes/deeppurple-amber.css"
"~@angular/material/prebuilt-themes/indigo-pink.css"
"~@angular/material/prebuilt-themes/pink-bluegrey.css"
"~@angular/material/prebuilt-themes/purple-green.css"
```

## Creando una estructura básica (flex-layout)

Una estructura básica consiste en un contenedor (fxLayout) que define la dirección de los elementos hijos (fila o columna) y como se alinean según estas directivas:

- [fxLayout](#) → row | column | row-reverse | column-reverse
- [fxLayoutAlign](#) → <main-axis> <cross-axis>. Main axis: start | center | end | space-around | space-between. Cross axis: start | center | end | stretch.
- [FxFlexGap](#) → % | px | vw | vh

Y estas otras directivas se aplican a los elementos hijos (**fxFlex**), que por defecto ocupan todo el espacio disponible de forma proporcional:

- [fxFlex](#) → (empty) | px | % | vw | vh | <grow> <shrink> <basis>

- [flexOrder](#) → number
- [flexOffset](#) → % | px | vw | vh
- [flexAlign](#) → start | baseline | center | end
- [flexFill](#)

Se pueden crear [estructuras responsive](#) usando sufijos para el tamaño de la pantalla (flex.tamaño) como: **xs** (extra small), **sm** (small), **md** (medium), **lg** (large), **xl** (extra large), gt-md (greater than medium), lt-lg (lower than large), etc.

```
<div flexLayout="row" flexLayoutWrap class="zero">
  <div flex="33" flex.lt-md="100" class="one"></div>
  <div flex="33" flex.xs="" flexLayout="column" flexLayout.xs="row" class="two">
    <div flex="20" class="two_one"></div>
    <div flex="40px" class="two_two"></div>
    <div flex class="two_three"></div>
  </div>
  <div flex flexHide.xs class="three"></div>
</div>
```

```
.zero { height: 300px; }
.one { background-color: green; }
.two_one { background-color: darkblue; }
.two_two { background-color: blue; }
.two_three { background-color: violet; }
.three { background-color: red; }
```



md, lg and xl



sm



xs

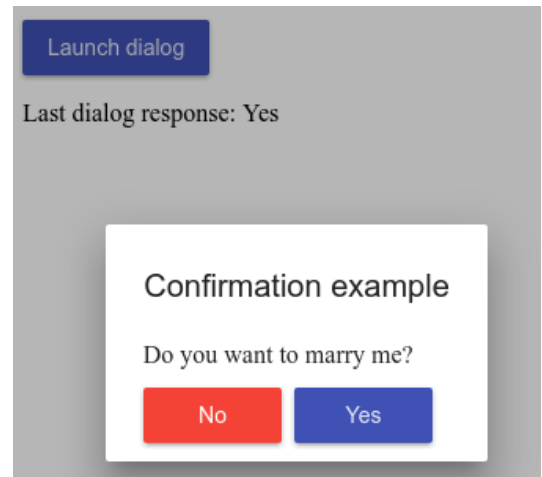


## Mostrando una ventana modal

Crear una ventana modal es similar a cómo lo hicimos con Bootstrap. Crearemos un componente con el contenido de la ventana modal. Como no se usa en ninguna ruta ni plantilla de otro componente, debemos incluirlo dentro de **entryComponents**.

Necesitaremos importar al menos 2 módulos de Material (Button y Dialog):

```
@NgModule({
  declarations: [
    ...
    ModalConfirmComponent
  ],
  imports: [
    ...
    MatButtonModule,
    MatDialogModule
  ],
  entryComponents: [
    ModalConfirmComponent
  ],
  ...
})
export class AppModule { }
```



El servicio para crear y mostrar modales se llama **MatDialog**. Devuelve una referencia a la ventana modal (MatDialogRef), que permite, por ejemplo, obtener el valor devuelto por la misma (dentro de un **Observable** esta vez):

```
export class AppComponent {
  response = '';

  constructor(private dialog: MatDialog) {}

  openDialog() {
    const dialogRef = this.dialog.open(ModalConfirmComponent, {
      data: {
        title: 'Confirmation example',
        body: 'Do you want to marry me?'
      }
    });

    dialogRef.afterClosed().subscribe(
      resp => this.response = resp ? 'Yes' : 'No'
    );
  }
}
```

Plantilla del componente (ModalConfirmComponent):

```
<h2 mat-dialog-title>{{data.title}}</h2>
<mat-dialog-content>{{data.body}}</mat-dialog-content>
<mat-dialog-actions>
  <button mat-raised-button color="warn" [mat-dialog-close]="false">No</button>
  <button mat-raised-button color="primary" [mat-dialog-close]="true">Yes
</button>
</mat-dialog-actions>
```

Y aquí la clase del componente:

```
export class ModalConfirmComponent implements OnInit {
  constructor(@Inject(MAT_DIALOG_DATA) public data: any,
```

```
    private dialogRef: MatDialogRef<ModalConfirmComponent>) { }  
  
    ngOnInit() {}  
  
}
```

La forma de recibir los datos en el constructor (donde está el título y el cuerpo de la ventana) es una forma especial de inyección de dependencias. Se utiliza cuando la dependencia no es el objeto de una clase (servicio), y se pone el nombre del dato “global” dentro del decorador `@Inject`. Como también recibimos la referencia a la ventana modal, podemos cerrarla a partir de la misma, por ejemplo (la directiva `mat-dialog-close` hace eso mismo, devolviendo el valor indicado).

# Integración con Google Maps

La integración con Google Maps es muy sencilla gracias a este plugin: <https://angular-maps.com/>. Para instalarlo:

```
npm install @agm/core
```

Lo primero será crear una [clave de Google Maps API](#). Posteriormente importaremos el módulo **AgmCoreModule** en nuestro módulo de aplicación:

```
import { AgmCoreModule } from '@agm/core';
...

@NgModule({
  ...
  imports: [
    ...
    AgmCoreModule.forRoot({
      apiKey: 'CLAVE_DE_LA_API'
    })
  ],
  ...
})
export class AppModule { }
```

Si queremos usar Google Maps en otro módulo, también importaremos **AgmCoreModule** (sin llamar a `forRoot`). En el componente donde queramos mostrar un mapa, añadimos lo siguiente a la plantilla:

```
<agm-map [latitude]="lat" [longitude]="lng" [zoom]="zoom">
  <agm-marker [latitude]="lat" [longitude]="lng"></agm-marker>
</agm-map>
```

En el componente estarán definidos los valores **lat** (latitud), **lng** (longitud), y **zoom**.

```
@Component({
  ...
})
export class AppComponent {
  ...
  lat = 38.4039418;
  lng = -0.5288701;
  zoom = 17;
}
```

Debemos especificar las dimensiones del mapa (al menos el alto) en el CSS del componente:

```
agm-map {
  height: 300px;
}
```



# Google Places Autocomplete

---

Como el plugin de **angular-maps** carga la librería de Google Maps en el navegador, podemos incluir otras librerías relacionadas como Google Places fácilmente. Junto con la clave de la API, podemos pasar un array con dichas librerías.

```
@NgModule ({
  ...
  imports: [
    ...
    AgmCoreModule.forRoot ({
      apiKey: 'API_KEY',
      libraries: ['places']
    })
  ],
  ...
})
export class AppModule { }
```

Como vamos a usar directamente la librería de JavaScript de Google Maps, para integrarla con TypeScript, debemos instalar su archivo de definiciones de TypeScript:

```
npm i -D @types/googlemaps
```

Posteriormente le indicamos a TypeScript que lo use (**src/tsconfig.app.json**):

```
{
  "extends": "../tsconfig.json",
  "compilerOptions": {
    ...
    "types": ["googlemaps"]
  },
  ...
}
```

Vamos a crear una directiva de atributo, para cuando se la pongamos a un `<input>`, haga consultas a google places de sitios sugeridos conforme vayamos tecleando. Además, cuando seleccionemos un lugar de los sugeridos, se emitirá un evento (EventEmitter) con la latitud y longitud del sitio seleccionado.

```
ng g directive gmaps-autocomplete
```

Cuando recibimos eventos de otra librería que no se integra con Angular, por ejemplo, el objeto Autocomplete de Google Places nos devolverá un evento 'place\_changed', se dice que está fuera de la zona de Angular (**Angular Zone**). Esto significa que si procesamos el evento y cambiamos alguna variable que afecta a la vista (nueva posición del mapa por ejemplo), Angular no se dará cuenta y no la actualizará. Para forzar esta detección, usaremos el servicio **NgZone**, y emitiremos la latitud y la longitud al componente padre dentro del método **NgZone.run** (si no, no actualizará la posición del mapa, al menos no visualmente). Esto hará que Angular esté pendiente de los cambios que genera NgZone y actualice la vista siempre.

```
@Directive ({
  selector: '[amGmapsAutocomplete]'
})
export class GmapsAutocompleteDirective {
```

```

@Output() placeChanged: EventEmitter<google.maps.LatLng> =
    new EventEmitter<google.maps.LatLng>();

constructor(private el: ElementRef, private mapsLoader: MapsAPILoader,
    private ngZone: NgZone) {
    this.mapsLoader.load().then(() => { // Cuando Google Maps está cargado
        const autocomplete =
            new google.maps.places.Autocomplete(el.nativeElement);
        autocomplete.addListener('place_changed', () => { // Evento de GMaps
            const place = autocomplete.getPlace();
            if (place.geometry && place.geometry.location) {
                this.ngZone.run(() => { // Ejecutamos dentro de Angular Zone
                    this.placeChanged.emit(place.geometry.location)
                });
            }
        });
    });
}

```

Ahora sólo tenemos que usar la directiva y escuchar el evento **placeChanged**:

```

<div>
  <input type="text" amGmapsAutocomplete (placeChanged)="changePosition($event)">
</div>
<agm-map [latitude]="lat" [longitude]="lng" [zoom]="zoom">
  <agm-marker [latitude]="lat" [longitude]="lng"></agm-marker>
</agm-map>

...
export class AppComponent {
  lat = 38.4039418;
  lng = -0.5288701;
  zoom = 17;
  ...

  changePosition(pos: google.maps.LatLng) {
    console.log(pos);
    this.lat = pos.lat();
    this.lng = pos.lng();
  }
}

```



# Angular Charts (ngx-charts)

[ngx-charts](#) es una librería de Angular para mostrar gráficas dinámicas basada en la popular librería de JavaScript [d3js](#). También tendremos que instalar el módulo de animaciones de Angular:

```
npm i @swimlane/ngx-charts d3 @angular/animations
```

Puedes ver algunos ejemplos de gráficas [aquí](#). Vamos a crear una gráfica de barras verticales que se actualice dinámicamente. Lo primero será importar el módulo de animaciones y el de ngx-charts:

```
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { NgxChartsModule } from '@swimlane/ngx-charts';
...
@NgModule({
  ...
  imports: [
    BrowserModule,
    FormsModule,
    BrowserAnimationsModule,
    NgxChartsModule
  ],
  ...
})
export class AppModule { }
```

Cada gráfica tiene su componente específico. En nuestro caso usaremos **ngx-charts-bar-vertical** con sus correspondientes opciones. El elemento padre que contiene la gráfica debe ser al menos de las mismas dimensiones que la gráfica.

```
<div style="height: 400px;">
  <ngx-charts-bar-vertical
    [view]="view" [results]="data" [gradient]="true"
    [xAxis]="true" [yAxis]="true" [legend]="legend"
    [showXAxisLabel]="true" [showYAxisLabel]="true"
    xAxisLabel="Millions" yAxisLabel="Fruits" (select)="onSelect($event)">
  </ngx-charts-bar-vertical>
</div>

<h4>Add Fruit:</h4>
<p>Type: <input type="text" [(ngModel)]="newFruit.name"></p>
<p>Quantity: <input type="number" [(ngModel)]="newFruit.value"></p>
<p><button (click)="addFruit()">Add</button></p>
```

Ten en cuenta que la gráfica no se actualiza si quitamos o añadimos una fruta del array, sino cuando la referencia al array cambia completamente. De esta manera, como ya vimos al crear pipes y directivas estructurales, la detección de cambios es mucho menos costosa. También podemos hacer esto con nuestros componentes (que sólo se actualice la vista cuando cambie la referencia de una propiedad de componente) usando una [estrategia de detección de cambios](#) diferente.

Este es el componente que contiene la gráfica:

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

```

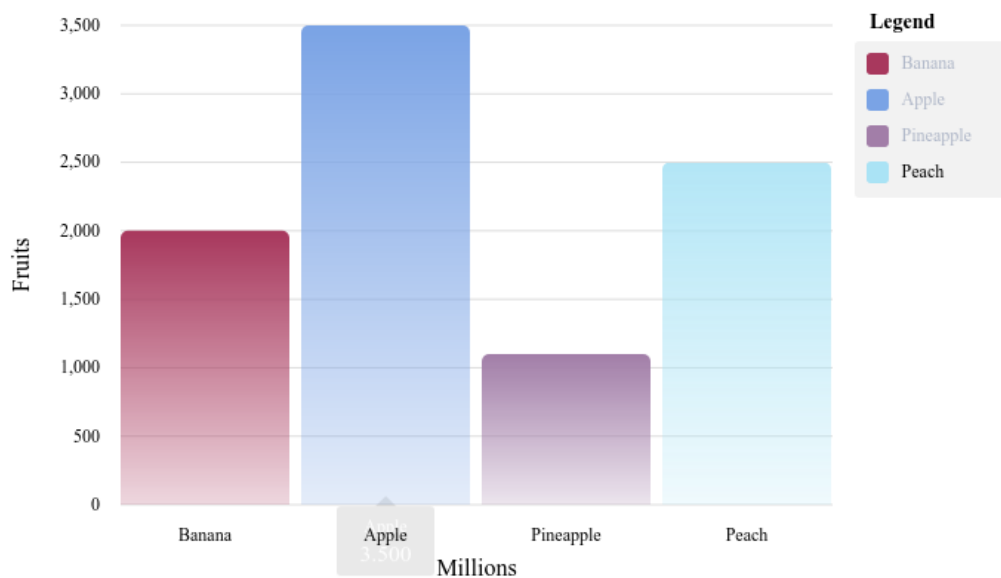
@Component({
  selector: 'ac-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  legend = 'Fruits exports';
  view = [600, 400]; // Tamaño (Por defecto → mismo tamaño que el padre)
  data: {name: string, value: number}[] = [{
    name: 'Banana',
    value: 2000
  }, {
    name: 'Apple',
    value: 3500
  }];

  newFruit: {name: string, value: number} = {
    name: '',
    value: 0
  };

  onSelect(result) {
    console.log(result);
  }

  addFruit() {
    // Para actualizar, necesitamos reemplazar el array por otro nuevo
    this.data = [...this.data, this.newFruit];
    this.newFruit = { name: '', value: 0 };
  }
}

```



#### Add Fruit:

Type:

Quantity: