¿Qué son los componentes anidados?

En Angular, los **componentes anidados** son aquellos componentes que están incluidos dentro de otros componentes, estableciendo una relación de **padre-hijo**. Esta característica permite organizar las aplicaciones en módulos más pequeños, mejorando la reutilización, la escalabilidad y el mantenimiento del código.

Concepto de Componentes Anidados

Un componente anidado (hijo) es parte de la plantilla de otro componente (padre). El componente hijo generalmente se utiliza para encapsular partes específicas de la interfaz o funcionalidades que son reutilizables en diferentes contextos.

Ventajas principales:

1. **Reutilización de Lógica y Diseño:** Permite reutilizar el mismo componente en múltiples lugares de la aplicación.
2. **Organización Modular:** Divide la aplicación en unidades pequeñas, facilitando su desarrollo y mantenimiento.
3. **Comunicación Clara:** Los datos fluyen entre componentes de manera controlada mediante decoradores como `@Input` y `@Output`.

Ejemplo Conceptual

Imaginemos que queremos construir un panel de usuarios.

- **Componente Padre:** Un panel principal que muestra la lista de usuarios.
- **Componente Hijo:** Tarjetas individuales que representan la información de cada usuario.

Estructura Visual:

```
[ Panel de Usuarios (Componente Padre) ]
```

```
├── [ Tarjeta de Usuario (Componente Hijo) ]
├── [ Tarjeta de Usuario (Componente Hijo) ]
└── [ Tarjeta de Usuario (Componente Hijo) ]
```

Funcionalidad

- **El Padre (`user-panel`)** pasa datos al Hijo (`user-card`) mediante **input properties**.
- **El Hijo (`user-card`)** puede notificar al Padre sobre eventos, como un clic, utilizando **output events**.

Comunicación entre Componentes

1. Del Padre al Hijo (Input)

El componente padre envía información al hijo utilizando el decorador `@Input`.
Por ejemplo, el padre puede enviar el nombre, imagen y descripción de cada usuario.

```
<app-user-card [nombre]="usuario.nombre"
[imagen]="usuario.imagen"></app-user-card>
```

2. Del Hijo al Padre (Output)

El componente hijo puede enviar eventos al padre mediante el decorador `@Output`.
Por ejemplo, cuando se hace clic en un botón dentro de la tarjeta, el hijo puede notificar al padre.

```
<app-user-card (onClick)="manejarClic($event)"></app-user-card>
```

Importancia de los Componentes Anidados

- **Eficiencia:** Los componentes hijos pueden diseñarse una vez y reutilizarse múltiples veces.
- **Mantenibilidad:** Si necesitas cambiar el diseño o la lógica de las tarjetas, basta con actualizar el componente hijo.

- **Escalabilidad:** Permite que equipos trabajen en diferentes componentes de manera independiente.

Los componentes anidados son una herramienta poderosa que promueve el desarrollo modular y organizado en Angular, haciendo que las aplicaciones sean más fáciles de mantener y escalar.

¿Qué son los Template-Driven Forms?

Los Template-Driven Forms son un tipo de formularios en Angular que se construyen principalmente desde la vista (HTML), lo que significa que la lógica de control y validación está estrechamente vinculada a la plantilla. Se utilizan las directivas `ngForm` y `ngModel` para gestionar los formularios de forma sencilla.

Características Principales

- **Enfoque en la vista**: La lógica se define directamente en la plantilla HTML.
- **Vinculación de datos bidireccional**: Se usa la directiva `ngModel` para enlazar los datos de la vista con la clase TypeScript.
- **Simplicidad**: Son fáciles de usar y entender, ideales para formularios pequeños.
- **Directivas clave**: `ngForm`, `ngModel`, `ngSubmit`.

Ejemplo Práctico

Paso 1: Configura el `FormsModule` en `app.module.ts`:

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


@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, FormsModule],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Paso 2: Crea la plantilla de un formulario simple:

```
<form #formPersona="ngForm" (ngSubmit)="guardar(formPersona.value)">

  <label for="nombre">Nombre</label>

  <input type="text" id="nombre" name="nombre"
[(ngModel)]="persona.nombre" required>



  <label for="edad">Edad</label>

  <input type="number" id="edad" name="edad"
[(ngModel)]="persona.edad" required>



  <button type="submit">Guardar</button>

</form>
```

¿Qué es el FormBuilder?

El FormBuilder es una clase de Angular que permite la creación de formularios reactivos de forma más sencilla y concisa. Facilita la creación de FormGroup y FormControl, reduciendo la cantidad de código repetitivo.

Características Principales

- **Facilita la creación de formularios**: Reduce el código necesario para crear formularios.
- **Definición clara de las validaciones**: Permite definir validaciones desde la creación de los campos.
- **Uso de objetos y arreglos**: Facilita la creación de estructuras de formularios dinámicos.

Ejemplo Práctico

Paso 1: Configura el ReactiveFormsModule en app.module.ts:

typescript

```
Copiar código

import { ReactiveFormsModule } from '@angular/forms';


@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, ReactiveFormsModule],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Paso 2: Crea el formulario usando FormBuilder en la clase del componente:

```
import { FormBuilder, FormGroup } from '@angular/forms';


export class AppComponent {
  formulario: FormGroup;


  constructor(private fb: FormBuilder) {
    this.formulario = this.fb.group({
      nombre: [''],
      edad: ['']
    });
  }
```

```
  guardar() {
    console.log(this.formulario.value);
  }
}
```

Paso 3: Conecta la vista con el formulario:

```
<form [formGroup]="formulario" (ngSubmit)="guardar()">
  <label for="nombre">Nombre</label>
  <input type="text" id="nombre" formControlName="nombre">

  <label for="edad">Edad</label>
  <input type="number" id="edad" formControlName="edad">

  <button type="submit">Guardar</button>
</form>
```

¿Qué son las Template Reference Variables?

Las Template Reference Variables (variables de referencia de plantilla) permiten hacer referencia a elementos del DOM desde la plantilla y acceder a sus propiedades directamente en el componente TypeScript. Se definen usando `#variable`.

Características Principales

- **Acceso a elementos DOM**: Se pueden capturar elementos HTML directamente desde la plantilla.
- **Interacción con Angular**: Usadas en combinación con **@ViewChild** para acceder a los elementos desde la clase.
- **Aplicación en componentes hijos**: Pueden hacer referencia a componentes hijos.

Ejemplo Práctico

Paso 1: Declara la variable de referencia en la plantilla:

```
<input #miCuadro type="text" placeholder="Escribe aquí">
<button (click)="cambiarColor()">Cambiar color</button>
```

Paso 2: Usa la variable de referencia con **@ViewChild** en la clase TypeScript:

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

export class AppComponent {
  @ViewChild('miCuadro') cuadro!: ElementRef;

  cambiarColor() {
    this.cuadro.nativeElement.style.backgroundColor = 'yellow';
  }
}
```

Migración de Template-Driven Forms a Reactive Forms

Inserción sugerida: Después de la sección de "Template Reference Variables".

¿Por qué migrar de Template-Driven a Reactive Forms?

Los Reactive Forms son más escalables y robustos, permitiendo un control total de la lógica de los formularios desde la clase TypeScript.

Pasos para migrar

Importar ReactiveFormsModule:
typescript

```
import { ReactiveFormsModule } from '@angular/forms';
```

1.

Actualizar la lógica de la clase: Antes (Template-Driven):

```
export class AppComponent {
```

```
  persona = { nombre: '', edad: '' };
}
```

Después (Reactive Forms):
typescript
Copiar código

```
import { FormGroup, FormBuilder } from '@angular/forms';

export class AppComponent {
  formulario: FormGroup;

  constructor(private fb: FormBuilder) {
    this.formulario = this.fb.group({
      nombre: [''],
      edad: ['']
    });
  }
}
```

2.

Actualizar la vista: Antes (Template-Driven):

```
<input type="text" [(ngModel)]="persona.nombre">
```

3. Después (Reactive Forms):
   html
   Copiar código
   ```
   <input type="text" formControlName="nombre">
   ```

¿Qué son las validaciones?

Las validaciones aseguran que los formularios tengan entradas de datos correctas. En Angular, se usa la clase **Validators**para aplicar reglas de validación, garantizando que los datos ingresados cumplan con ciertos criterios.

Tipos de Validaciones

- **Requerido**: Verifica que un campo no esté vacío (`Validators.required`).
- **Mínima longitud**: Verifica que el campo tenga una cantidad mínima de caracteres (`Validators.minLength()`).
- **Máxima longitud**: Establece un límite de caracteres (`Validators.maxLength()`).
- **Patrones**: Verifica que el campo siga un patrón específico (`Validators.pattern()`).

- **Validación personalizada**: Permite definir validaciones personalizadas mediante funciones.

Ejemplo Práctico

Paso 1: Configurar las validaciones al crear el FormGroup:

```
import { FormBuilder, Validators } from '@angular/forms';

export class AppComponent {
  formulario = this.fb.group({
    nombre: ['', [Validators.required, Validators.minLength(3)]],
    email: ['', [Validators.required, Validators.email]]
  });

  constructor(private fb: FormBuilder) {}
}
```

Paso 2: Mostrar mensajes de error en la plantilla:

```
<input type="text" formControlName="nombre">
<div *ngIf="formulario.get('nombre')?.errors?.required">El nombre es
requerido</div>
<div *ngIf="formulario.get('nombre')?.errors?.minlength">El nombre
debe tener al menos 3 caracteres</div>
```

Comunicación de Padre a Hijo

En Angular, el decorador `@Input` se utiliza para permitir que un componente hijo reciba datos desde su componente padre. Este mecanismo es esencial para crear aplicaciones dinámicas, donde los datos pueden fluir de manera controlada entre diferentes partes de la interfaz.

¿Qué es el decorador `@Input`?

`@Input` es un decorador que se aplica a las propiedades de un componente hijo, permitiendo que estas reciban valores enviados desde el componente padre. Esta funcionalidad es clave para establecer la comunicación unidireccional de **padre a hijo**.

Ventajas:

- Facilita el paso de datos personalizados al componente hijo.
- Promueve el diseño modular y reutilizable de componentes.

Ejemplo Práctico

Supongamos que tenemos una aplicación que muestra una lista de usuarios.

- **Componente Padre:** Gestiona los datos de los usuarios.
- **Componente Hijo:** Representa cada usuario como una tarjeta.

Paso 1: Configurar el Componente Hijo (`user-card`)

El componente hijo define una propiedad decorada con `@Input` para recibir datos.

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


@Component({
  selector: 'app-user-card',
  template: `
    <div>
      <h3>{{ nombre }}</h3>
      <p>{{ descripcion }}</p>
    </div>
  `
})
export class UserCardComponent {
  @Input() nombre!: string;
```

```
  @Input() descripcion!: string;
}
```

Paso 2: Configurar el Componente Padre (`user-panel`)

El componente padre utiliza el selector del hijo (`app-user-card`) y le pasa datos dinámicos.

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

@Component({
  selector: 'app-user-panel',
  template: `
    <h2>Lista de Usuarios</h2>
    <app-user-card
      *ngFor="let usuario of usuarios"
      [nombre]="usuario.nombre"
      [descripcion]="usuario.descripcion">
    </app-user-card>
  `
})
export class UserPanelComponent {
  usuarios = [
    { nombre: 'Ana López', descripcion: 'Desarrolladora Front-End' },
    { nombre: 'Carlos Pérez', descripcion: 'Diseñador UX/UI' },
```

```
    { nombre: 'María Rodríguez', descripcion: 'Gerente de Proyectos' }
  ];
}
```

Salida Esperada

El componente padre genera una lista dinámica de tarjetas de usuario:

- **Ana López:** Desarrolladora Front-End
- **Carlos Pérez:** Diseñador UX/UI
- **María Rodríguez:** Gerente de Proyectos

Resumen de Pasos

1. Define propiedades en el componente hijo con el decorador `@Input`.
2. Usa el selector del componente hijo en el componente padre.
3. Pasa los valores mediante enlaces de propiedades usando corchetes (`[propiedad]`).

Conclusión:
El uso del decorador `@Input` facilita una comunicación eficiente de padre a hijo, haciendo que los componentes sean más reutilizables y las aplicaciones más organizadas.

Comunicación de Hijo a Padre

En Angular, la comunicación de un **componente hijo** a un **componente padre** se logra mediante el decorador **`@Output`** y la clase **`EventEmitter`**. Este enfoque permite que un componente hijo notifique eventos o envíe datos al padre, fomentando una interacción dinámica entre ambos.

¿Qué es el decorador `@Output`?

`@Output` es un decorador que permite que un componente hijo emita eventos personalizados hacia su componente padre. Este evento se crea utilizando la clase `EventEmitter`.

Ventajas:

- Facilita la comunicación ascendente entre componentes.
- Permite que el padre reaccione a las acciones realizadas en el hijo.

Ejemplo Práctico

Supongamos que estamos desarrollando una aplicación donde un usuario puede seleccionar elementos de una lista.

- **Componente Hijo:** Representa cada elemento y permite seleccionarlo.
- **Componente Padre:** Recibe la selección desde el hijo y la gestiona.

Paso 1: Configurar el Componente Hijo (`item-card`)

El componente hijo emite un evento hacia el padre al hacer clic en un botón.

typescript

Copiar código

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


@Component({
  selector: 'app-item-card',
  template: `
    <div>
      <p>{{ nombre }}</p>
      <button (click)="seleccionar()">Seleccionar</button>
    </div>
  `
```

```
})
export class ItemCardComponent {
  @Output() seleccionado = new EventEmitter<string>();
  nombre: string = 'Elemento';

  seleccionar() {
    this.seleccionado.emit(this.nombre);
  }
}
```

Paso 2: Configurar el Componente Padre (`item-list`)

El componente padre escucha el evento emitido por el hijo y gestiona la acción.

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

@Component({
  selector: 'app-item-list',
  template: `
    <h2>Lista de Elementos</h2>
    <app-item-card
      *ngFor="let item of items"
      [nombre]="item"
      (seleccionado)="gestionarSeleccion($event)">
```

```
    </app-item-card>

    <h3>Seleccionado: {{ seleccionado }}</h3>

  `

})

export class ItemListComponent {

  items = ['Elemento 1', 'Elemento 2', 'Elemento 3'];

  seleccionado: string = '';


  gestionarSeleccion(item: string) {

    this.seleccionado = item;

  }

}
```

Salida Esperada

1. El componente padre muestra una lista dinámica:
    - Elemento 1
    - Elemento 2
    - Elemento 3
2. Al seleccionar un elemento, el componente hijo envía el nombre al padre, y este actualiza el valor seleccionado.

Resumen de Pasos

1. Define una propiedad en el componente hijo decorada con `@Output` y usa `EventEmitter`.
2. Usa el evento emitido por el hijo en el componente padre para gestionar la acción.
3. Vincula el evento en el HTML del padre con paréntesis `()`.

Conclusión:
La comunicación de hijo a padre en Angular permite construir aplicaciones más interactivas y dinámicas, fomentando una interacción fluida entre los componentes.

Uso del Decorador @ViewChild

En Angular, el decorador `@ViewChild` permite acceder y manipular elementos del DOM o referencias de componentes hijos desde el componente padre. Es especialmente útil para interactuar directamente con elementos HTML, plantillas o instancias de componentes secundarios.

¿Qué es `@ViewChild`?

`@ViewChild` es un decorador que busca y obtiene una referencia a un elemento del DOM o a un componente anidado definido en la plantilla de un componente. Esta referencia permite acceder a sus propiedades y métodos directamente desde la clase del componente padre.

Ventajas:

- Permite realizar cambios dinámicos en el DOM o en componentes secundarios.
- Facilita el acceso directo a métodos y propiedades de componentes hijos.
- Simplifica la manipulación de elementos en casos específicos donde el data binding no es suficiente.

Ejemplo Práctico

Supongamos que queremos resaltar un cuadro de texto cuando el usuario haga clic en un botón.

Escenario

- **Elemento del DOM:** Un cuadro de texto.
- **Acción:** Cambiar el color de fondo del cuadro al hacer clic en el botón.

Paso 1: Configurar el HTML

En el archivo de plantilla (`app.component.html`), definimos un cuadro de texto y un botón.

```
<input #miCuadro type="text" value="Texto inicial">

<button (click)="resaltar()">Resaltar</button>
```

Paso 2: Usar `@ViewChild` en el Componente

En la clase del componente (`app.component.ts`), utilizamos `@ViewChild` para obtener una referencia al cuadro de texto.

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


@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  @ViewChild('miCuadro') cuadroTexto!: ElementRef;

  resaltar() {
    this.cuadroTexto.nativeElement.style.backgroundColor = 'yellow';
  }
}
```

Explicación:

1. `@ViewChild('miCuadro')`: Busca el elemento con la referencia `#miCuadro` en la plantilla.
2. `ElementRef`: Proporciona acceso al elemento nativo del DOM.
3. `nativeElement`: Permite manipular directamente las propiedades del elemento.

Salida Esperada

1. Al cargar la aplicación, se muestra un cuadro de texto y un botón.
2. Al hacer clic en el botón, el color de fondo del cuadro de texto cambia a amarillo.

Consideraciones Importantes

- **Tiempo de Vida:** `@ViewChild` solo está disponible después de que Angular haya inicializado la vista, es decir, en el ciclo de vida `ngAfterViewInit()`.
- **Acceso Seguro:** Siempre verifica que la referencia no sea `undefined` antes de usarla.

Conclusión:
El decorador `@ViewChild` es una herramienta poderosa en Angular para interactuar directamente con elementos del DOM y componentes secundarios. Su uso adecuado permite optimizar la manipulación dinámica y mejorar la interactividad de la aplicación.

Recomendaciones para manejar múltiples niveles de comunicación, prevenir errores comunes y optimizar el flujo de datos en proyectos complejos

La comunicación entre componentes es uno de los desafíos más importantes en el desarrollo de aplicaciones con Angular, especialmente en proyectos complejos con múltiples niveles de componentes anidados. Una correcta gestión de la comunicación no solo mejora la claridad del código, sino que también previene errores comunes y optimiza el flujo de datos.

1. Múltiples niveles de comunicación

Mejores prácticas de comunicación

Cuando los componentes están jerárquicamente relacionados, la comunicación se establece entre componentes padre-hijo, hijo-padre y entre hermanos. Las siguientes recomendaciones te ayudarán a manejar esta comunicación de forma eficiente:

- **Uso de @Input() y @Output()**:
    - Utiliza @Input() para enviar datos desde un componente padre a un componente hijo.
    - Usa @Output() con EventEmitter para enviar eventos o datos desde el componente hijo hacia su componente padre.
    - Asegúrate de que los nombres de las propiedades de entrada (input) y de los eventos (output) sean claros y representativos de la acción o el dato que manejan.
- **Evita la "prop drilling"**:
    - El "prop drilling" ocurre cuando los datos tienen que pasar por varios niveles de componentes intermedios que no los necesitan directamente.
    - Para evitarlo, utiliza servicios compartidos con el patrón Singleton e Inyección de Dependencias para facilitar la comunicación entre componentes no relacionados directamente.
- **Comunicación entre hermanos**:
    - Cuando dos componentes hermanos necesitan comunicarse, es más eficiente utilizar un servicio compartido en lugar de enviar los datos a través del padre.
    - El servicio puede actuar como un "canal" que mantiene el estado y facilita la sincronización de datos entre los componentes hermanos.

2. Errores comunes y cómo prevenirlos

En proyectos complejos, es habitual cometer errores en la comunicación de componentes. Aquí algunos de los más comunes y las recomendaciones para evitarlos:

- **No inicializar las propiedades @Input() adecuadamente**:
    - Error: El componente hijo intenta acceder a una variable @Input() antes de que el padre la haya proporcionado.
    - Solución: Usa el hook de ciclo de vida ngOnChanges() para asegurarte de que la variable ha sido actualizada antes de acceder a ella.
- **Memoria no liberada por suscripciones de servicios**:
    - Error: No se eliminan las suscripciones de observables, lo que produce fugas de memoria.
    - Solución: Usa el operador takeUntil() con Subject() o ngOnDestroy() para cancelar las suscripciones al destruir el componente.
- **Emisión de eventos duplicados o innecesarios**:
    - Error: Emitir eventos en bucle o cada vez que cambia una pequeña parte del estado.
    - Solución: Usa el operador distinctUntilChanged() para evitar la emisión de eventos duplicados.
- **Uso excesivo de ViewChild()**:
    - Error: Acceder directamente a los hijos con @ViewChild(), lo que genera dependencias de implementación.
    - Solución: Siempre que sea posible, utiliza @Input() y @Output() para manejar la comunicación en lugar de ViewChild.
- **Falta de control de errores en la comunicación con servicios**:
    - Error: Cuando un servicio falla, la aplicación se rompe.

    - Solución: Usa catchError() de RxJS para capturar errores y mostrar mensajes de error al usuario sin afectar la aplicación.

3. Optimización del flujo de datos

Optimizar el flujo de datos es esencial para mejorar la eficiencia de la aplicación. Aquí algunas prácticas recomendadas:

- **Uso de OnPush Change Detection**:
    - Establece la detección de cambios en el modo OnPush para que Angular solo actualice los componentes cuando cambien sus entradas (@Input()).
    - Esto reduce las verificaciones innecesarias y mejora el rendimiento de la aplicación.
- **Evita la mutación de objetos y arreglos**:
    - En lugar de modificar directamente un arreglo u objeto, crea una nueva instancia con los cambios aplicados.
    - Esto facilita la detección de cambios cuando se usa OnPush.
- **Uso de Observables y Subject()**:
    - Usa RxJS para manejar flujos de datos reactivos. Los Observables permiten emitir múltiples valores a lo largo del tiempo.
    - Usa BehaviorSubject o ReplaySubject para almacenar y proporcionar el último valor emitido a los nuevos suscriptores.
- **Evitar el uso excesivo de ngIf y ngFor**:
    - Estas directivas pueden afectar el rendimiento si no se usan correctamente.
    - Usa trackBy en ngFor para optimizar el rendimiento y reducir la re-renderización innecesaria.
- **Desacopla la lógica de negocio de la lógica de la vista**:
    - Mueve la lógica de negocio a servicios. Esto hace que los componentes sean más simples, fáciles de mantener y probar.
    - Usa Subject() y BehaviorSubject() en servicios para emitir y sincronizar el estado de la aplicación.

Conclusión

Manejar la comunicación en proyectos complejos de Angular requiere estrategias claras para la interacción entre componentes y la optimización del flujo de datos. La implementación de @Input(), @Output(), servicios compartidos y la correcta liberación de suscripciones son prácticas clave para evitar errores comunes. Además, la optimización del flujo de datos mediante el uso de OnPush, trackBy, y el control eficiente de cambios contribuye a la creación de aplicaciones más rápidas y sostenibles.

Resoluciones para las actividades de la Clase Práctica en Vivo - Unidad 3

1.Configuración de Bootstrap

Consigna: Configura la librería de Bootstrap en tu proyecto Angular y verifica su correcto funcionamiento.

Posible resolución:

1. Abre el archivo `index.html` del proyecto Angular.

Agrega la referencia de Bootstrap dentro de la etiqueta `<head>`, de la siguiente forma:

```
<link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.1/dist/css/bootstrap.
min.css" rel="stylesheet"
integrity="sha384-F3w7mX95PdgyTmZZMECAngseQB83DfGTowi0iMjiWaeVhAn4FJkq
JByhZMI3AhiU" crossorigin="anonymous">
```

2.
3. Guarda los cambios en el archivo `index.html`.

Verifica el correcto funcionamiento de Bootstrap utilizando una clase distintiva en la vista, por ejemplo, agregando un botón con la clase `btn btn-primary` en el archivo `app.component.html`:

```
<button class="btn btn-primary">Botón de Prueba</button>
```

4.

2.Creación de Componentes Anidados

Consigna: Crea dos componentes anidados y vincúlalos en la vista principal.

Posible resolución:

Abre la terminal y ejecuta los siguientes comandos para generar los componentes:

```
ng generate component nestedcomponent1
```

```
ng generate component nestedcomponent2
```

1.

Modifica el archivo `app.module.ts` para declarar e importar los nuevos componentes:

```
import { Nestedcomponent1Component } from
'./nestedcomponent1/nestedcomponent1.component';

import { Nestedcomponent2Component } from
'./nestedcomponent2/nestedcomponent2.component';


@NgModule({
  declarations: [
    AppComponent,
    Nestedcomponent1Component,
    Nestedcomponent2Component
  ],
  imports: [BrowserModule, FormsModule],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

2. Inserta los selectores de los nuevos componentes dentro del archivo `app.component.html`:

```
<app-nestedcomponent1></app-nestedcomponent1>

<app-nestedcomponent2></app-nestedcomponent2>
```

3.

---

3. Comunicación entre componentes con @Input

Consigna: Usa **@Input** para enviar datos desde un componente padre a un componente hijo.

Posible resolución:

Crea un componente hijo usando Angular CLI:

```
ng generate component employeelist
```

1.

En el archivo `employeelist.component.ts`, importa **@Input** y declara la variable que recibirá los datos:

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


@Component({
  selector: 'app-employeelist',
  templateUrl: './employeelist.component.html',
  styleUrls: ['./employeelist.component.css']
})
export class EmployeelistComponent {
  @Input() employeelist: string[];
}
```

2.

En el archivo `employeelist.component.html`, muestra la lista de empleados:

```
<ul>
  <li *ngFor="let item of employeelist">{{ item }}</li>
</ul>
```

3.

En el archivo `app.component.ts`, define la lista de empleados que se enviará al componente hijo:

```
export class AppComponent {
  employeelist = ['Empleado 1', 'Empleado 2', 'Empleado 3'];
```

```
}
```

4.

En el archivo `app.component.html`, utiliza el componente hijo e inyecta los datos de la lista:

```
<app-employeelist [employeelist]="employeelist"></app-employeelist>
```

5.

---

4. Comunicación entre componentes con @Output

Consigna: Usa **@Output** para enviar eventos desde un componente hijo al componente padre.

Posible resolución:

Genera el componente hijo con el siguiente comando:

```
ng generate component child-output
```

1.

En el archivo `child-output.component.ts`, configura **@Output** y **EventEmitter**:

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


@Component({
  selector: 'app-child-output',
  template: `<button (click)="emitirEvento()">Enviar al
padre</button>`,
  styleUrls: ['./child-output.component.css']
})
export class ChildOutputComponent {
  @Output() eventoHijo = new EventEmitter<string>();
```

```
  emitirEvento() {
    this.eventoHijo.emit('Mensaje desde el componente hijo');
  }
}
```

2.

En el archivo `app.component.html`, captura el evento emitido por el hijo:

```
<app-child-output
(eventoHijo)="recibirEvento($event)"></app-child-output>
```

3.

En el archivo `app.component.ts`, define la función que recibe el evento y muestra el mensaje en la consola:

```
export class AppComponent {
  recibirEvento(mensaje: string) {
    console.log('Mensaje recibido del hijo:', mensaje);
  }
}
```

5.Formulario Reactivo con Validaciones

Consigna: Crea un formulario reactivo que valide los campos de nombre y correo electrónico.

Posible resolución:

Asegúrate de importar **ReactiveFormsModule** en `app.module.ts`:

```
import { ReactiveFormsModule } from '@angular/forms';


@NgModule({
```

```
  imports: [
    BrowserModule,
    ReactiveFormsModule
  ],
  declarations: [AppComponent],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

1.

En el archivo `app.component.ts`, crea el formulario con **FormBuilder**:

```
import { Component } from '@angular/core';
import { FormBuilder, Validators } from '@angular/forms';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  formulario = this.fb.group({
    nombre: ['', [Validators.required, Validators.minLength(3)]],
    correo: ['', [Validators.required, Validators.email]]
  });
```

```
  constructor(private fb: FormBuilder) {}


  enviarFormulario() {
    if (this.formulario.valid) {
      console.log('Formulario enviado:', this.formulario.value);
    } else {
      console.log('Formulario inválido');
    }
  }
}
```

2.

En el archivo `app.component.html`, crea el formulario con validaciones y muestra los errores correspondientes:

html

Copiar código

```
<form [formGroup]="formulario" (ngSubmit)="enviarFormulario()">

  <label for="nombre">Nombre</label>

  <input id="nombre" formControlName="nombre">

  <div *ngIf="formulario.get('nombre')?.errors?.required">El nombre es requerido</div>

  <div *ngIf="formulario.get('nombre')?.errors?.minlength">El nombre debe tener al menos 3 caracteres</div>


  <label for="correo">Correo</label>

  <input id="correo" formControlName="correo">

  <div *ngIf="formulario.get('correo')?.errors?.required">El correo es requerido</div>
```

```
  <div *ngIf="formulario.get('correo')?.errors?.email">El formato del
correo es incorrecto</div>


  <button type="submit">Enviar</button>

</form>
```

---

6. Find the Bug - Comunicación entre componentes

Consigna: Encuentra y corrige los errores en la comunicación entre un componente padre y su hijo.

Posible resolución:

Error 1: Falta de comillas en la asignación de la variable `parentName`.
Corrección:

```
parentName: string = 'Juan Perez';
```

Error 2: Falta de tipo en la función `handleChildClick`.
Corrección:

```
handleChildClick(event: string) {

  console.log('El componente hijo envió:', event);

}
```