

# Angular

## *M-dulos*

CertiDevs

#  ndice de contenidos

1. M�dulos .....	1
2. Estructura b�sica de un m�dulo .....	1
2.1. Declaraciones .....	1
2.2. Importaciones .....	2
2.3. Exportaciones .....	2
2.4. Proveedores .....	2
2.5. Bootstrap .....	2
2.6. M�dulo ra�z y m�dulos de caracter�sticas .....	2
3. Ejemplo 1: Creaci�n de un m�dulo b�sico .....	2
4. Ejemplo 2: Creaci�n de un m�dulo con componentes y servicios .....	3
5. Ejemplo 3: Utilizar un m�dulo en otro m�dulo .....	4
6. Ejemplo 4: M�dulo de usuarios con varios componentes .....	5
7. Ejemplo 5: App gesti�n de tareas .....	6
7.1. AppModule (m�dulo ra�z) .....	6
7.2. SharedModule .....	7
7.3. TodoModule .....	8
8. Ejemplo 5: Aplicaci�n de Notas y etiquetas .....	9
8.1. Paso 1: Crear una nueva aplicaci�n Angular .....	9
8.2. Paso 2: Generar m�dulos .....	9
8.3. Paso 3: Crear componentes y servicios .....	9
8.4. Paso 4: Configurar m�dulos .....	10
8.5. Paso 5: Importar m�dulos en AppModule .....	11
8.6. Paso 6: Utilizar componentes de los m�dulos .....	11
8.7. Paso 7: Implementar la l�gica de los servicios y componentes .....	11
8.8. Paso 8: Crear las plantillas HTML de los componentes .....	14
9. Ejemplo @Input y @Output .....	15
10. Ejemplo ecommerce .....	18

# 1. Módulos

Los módulos son la base de la arquitectura de Angular.

Un módulo es una colección de componentes, directivas y pipes que se agrupan para formar una unidad funcional.

Cada aplicación Angular tiene al menos un módulo raíz, que se crea automáticamente durante la creación de la aplicación. Puede crear más módulos para agrupar funcionalidades relacionadas.

Un módulo en Angular es una unidad de organización que agrupa componentes, directivas, servicios y otros elementos relacionados. Los módulos facilitan la organización y la reutilización del código, lo que mejora la escalabilidad y la mantenibilidad del código.

Un módulo típico en Angular se define utilizando la clase `@NgModule`.

## 2. Estructura básica de un módulo

Un módulo en Angular se define utilizando la clase `@NgModule`.

Esta clase es un decorador que acepta un objeto de metadatos como argumento.

El objeto de metadatos define las propiedades clave del módulo, como declaraciones, importaciones, exportaciones y proveedores.

A continuación se muestra un ejemplo básico de un módulo en Angular:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';

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

### 2.1. Declaraciones

La propiedad `declarations` es una lista de componentes, directivas y pipes que pertenecen a este módulo.

Los elementos declarados en un módulo están disponibles para su uso en las plantillas (html) y typescript de los componentes que pertenecen a ese módulo.

## 2.2. Importaciones

La propiedad `imports` es una lista de otros módulos cuyos elementos exportados son necesarios en las plantillas de este módulo.

Al importar otros módulos, sus componentes, directivas y pipes exportados están disponibles para su uso en este módulo.

## 2.3. Exportaciones

La propiedad `exports` es una lista de componentes, directivas y pipes que deben ser accesibles desde otros módulos. Estos elementos exportados pueden ser utilizados por otros módulos cuando importan este módulo.

## 2.4. Proveedores

La propiedad `providers` es una lista de servicios y otros elementos que deben ser accesibles globalmente en toda la aplicación.

Los servicios declarados como proveedores en un módulo estarán disponibles en cualquier parte de la aplicación a través de la inyección de dependencias.

## 2.5. Bootstrap

La propiedad `bootstrap` es una lista de componentes que se deben cargar automáticamente cuando se inicia la aplicación.

Por lo general, solo hay un componente de inicio en la aplicación principal, que actúa como la raíz de la jerarquía de componentes.

## 2.6. Módulo raíz y módulos de características

En una aplicación Angular, suele haber un módulo raíz, que es el punto de entrada de la aplicación y contiene el componente de inicio.

Además del módulo raíz, es posible tener módulos de características que agrupan funcionalidades relacionadas. Los módulos de características pueden ser cargados de forma temprana (`eager`) o perezosa (`lazy`) para mejorar el rendimiento y la organización del código.

En resumen, los módulos en Angular proporcionan una estructura organizativa para agrupar y aislar partes de la aplicación. Ayudan a dividir el código en unidades más pequeñas y reutilizables, lo que facilita la escalabilidad y la mantenibilidad del proyecto.

## 3. Ejemplo 1: Creación de un módulo básico

En este ejemplo, se muestra el módulo llamado `AppModule`. Este módulo ya existe cuando se crea una aplicación de angular, se genera automáticamente en la creación del proyecto.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';

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

## 4. Ejemplo 2: Creación de un módulo con componentes y servicios

En este ejemplo, se crea un módulo llamado `UserModule` que incluye un componente `UserDetailComponent` y un servicio `UserService`.

Crear el componente `UserDetailComponent`.

Utiliza el comando:

```
ng generate component UserDetail
```

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

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

}
```

Crear el servicio `UserService`:

```
ng generate service User
```

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

@Injectable({
  providedIn: 'root'
})
export class UserService {

  constructor() { }
}
```

Crear el m-**dulo** `UserModule` e importar el componente `UserComponent` y el servicio `UserService`:

Utiliza el comando:

```
ng generate module User
```

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { UserDetailsComponent } from './user-detail.component';
import { UserService } from './user.service';

@NgModule({
  declarations: [UserDetailsComponent],
  imports: [CommonModule],
  providers: [UserService],
  exports: [UserDetailsComponent]
})
export class UserModule { }
```

## 5. Ejemplo 3: Utilizar un m-**dulo** en otro m-**dulo**

En este ejemplo, se muestra c-**mo** utilizar el m-**dulo** `UserModule` creado en el Ejemplo 2 dentro del m-**dulo** principal `AppModule`.

Importar el m-**dulo** `UserModule` en `AppModule` donde pone `imports`:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { UserModule } from './user/user.module';

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

```

    ],
    imports: [
        BrowserModule,
        UserModule
    ],
    providers: [],
    bootstrap: [AppComponent]
  })
  export class AppModule { }

```

Utilizar el componente `UserComponent` en la plantilla de `AppComponent`:

```

<!-- app.component.html -->
<h1>App Component</h1>
<app-user></app-user>

```

En este ejemplo, el módulo `UserModule` se importa en el módulo `AppModule`, lo que permite utilizar el componente `UserComponent` en la plantilla de `AppComponent`.

## 6. Ejemplo 4: Módulo de usuarios con varios componentes

Similar al ejemplo anterior, vamos a crear una estructura de módulo de usuarios con varios componentes.

El resultado será similar a este:

```

src/
|-- app/
    |-- user/
        |-- user-detail/
            |-- user-detail.component.html
            |-- user-detail.component.scss (o .css)
            |-- user-detail.component.ts
            |-- user-detail.component.spec.ts
        |-- user-list/
            |-- user-list.component.html
            |-- user-list.component.scss (o .css)
            |-- user-list.component.ts
            |-- user-list.component.spec.ts
        |-- user.module.ts
        |-- user-routing.module.ts (opcional, si necesitas rutas específicas para el
        módulo)
        |-- user.service.ts
        |-- user.service.spec.ts

```

En un proyecto Angular recién generado, una forma común y recomendada de estructurar los módulos, componentes y servicios es agruparlos por funcionalidad. Para lograr el ejemplo anterior utilizamos los siguientes comandos

Abrir una terminal dentro del proyecto angular y ejecutar los siguientes comandos:

```
ng generate module user

ng generate component user/user-detail

ng generate component user/user-list

ng generate service user/user
```

## 7. Ejemplo 5: App gestión de tareas

En este ejemplo, crearemos una aplicación que gestiona tareas (ToDo).

La aplicación incluirá tres módulos: **AppModule**, **TodoModule** y **SharedModule**.

**AppModule** es el módulo raíz, **TodoModule** manejará las tareas y **SharedModule** contendrá componentes y servicios compartidos entre diferentes módulos.

### 7.1. AppModule (módulo raíz)

El componente principal **AppComponent** ya viene creado al crear un nuevo proyecto angular:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  title = 'todo-app';
}
```

Crear la plantilla para **AppComponent**:

```
<!-- app.component.html -->
<h1>{{ title }}</h1>
<app-todo-list></app-todo-list>
```

Crear el módulo principal **AppModule** e importar **BrowserModule** y **TodoModule**:



```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import { TodoModule } from './todo/todo.module';
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    TodoModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

## 7.2. SharedModule

Crear un componente compartido `InputWithButtonComponent`:

```

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

@Component({
  selector: 'app-input-with-button',
  templateUrl: './input-with-button.component.html',
  styleUrls: ['./input-with-button.component.css']
})
export class InputWithButtonComponent {
  inputValue = '';

  @Output() submitValue = new EventEmitter<string>();
  onSubmit() {
    this.submitValue.emit(this.inputValue);
    this.inputValue = '';
  }
}

```

Crear la plantilla para `InputWithButtonComponent`:

```

<!-- input-with-button.component.html -->
<input [(ngModel)]="inputValue" placeholder="Enter task" />
<button (click)="onSubmit()">Add Task</button>

```

Crear el módulo `SharedModule` e importar `InputWithButtonComponent` y `FormsModule`:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { InputWithButtonComponent } from './input-with-button/input-with-button.component';
@NgModule({
  declarations: [InputWithButtonComponent],
  imports: [CommonModule, FormsModule],
  exports: [InputWithButtonComponent]
})
export class SharedModule { }
```

## 7.3. TodoModule

Crear el componente `TodoListComponent`:

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

@Component({
  selector: 'app-todo-list',
  templateUrl: './todo-list.component.html',
  styleUrls: ['./todo-list.component.css']
})
export class TodoListComponent {
  tasks: string[] = [];

  onTaskAdded(task: string) {
    this.tasks.push(task);
  }
}
```

Crear la plantilla para `TodoListComponent`:

```
<!-- todo-list.component.html -->

<app-input-with-button (submitValue)="onTaskAdded($event)"></app-input-with-button>
<ul>
<li *ngFor="let task of tasks">{{ task }}</li>
</ul>
```

Crear el módulo `TodoModule` e importar `TodoListComponent` y `SharedModule`:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { TodoListComponent } from './todo-list/todo-list.component';
import { SharedModule } from '../shared/shared.module';
```

```
@NgModule({
  declarations: [TodoListComponent],
  imports: [CommonModule, SharedModule],
  exports: [TodoListComponent]
})
export class TodoModule { }
```

En este ejemplo, hemos creado una aplicación para gestionar tareas con tres módulos:

- ¥ AppModule: el módulo raíz de la aplicación, que importa y utiliza el módulo TodoModule.
- ¥ SharedModule: un módulo que contiene un componente compartido InputWithButtonComponent. Este módulo exporta el componente para que otros módulos puedan utilizarlo.
- ¥ TodoModule: un módulo que maneja las tareas, que importa y utiliza el componente compartido InputWithButtonComponent desde el módulo SharedModule.

Estos módulos organizan el código de la aplicación en unidades lógicas y facilitan la reutilización de componentes y servicios entre diferentes partes de la aplicación.

## 8. Ejemplo 5: Aplicación de Notas y etiquetas

En este ejercicio, se creará una aplicación de notas simple utilizando módulos en Angular. La aplicación contendrá un módulo para gestionar las notas y otro para gestionar las etiquetas.

### 8.1. Paso 1: Crear una nueva aplicación Angular

Utiliza Angular CLI para crear una nueva aplicación llamada notes-app:

```
ng new notes-app
```

### 8.2. Paso 2: Generar módulos

Genera dos módulos, uno para notas y otro para etiquetas:

```
ng generate module notes
ng generate module tags
```

Estos comandos crearán los archivos `notes.module.ts` y `tags.module.ts` en las carpetas `src/app/notes` y `src/app/tags`, respectivamente.

### 8.3. Paso 3: Crear componentes y servicios

Genera componentes y servicios necesarios para los módulos de notas y etiquetas:

```
ng generate component notes/note-list
ng generate component notes/note-editor
ng generate service notes/note

ng generate component tags/tag-list
ng generate service tags/tag
```

Estos comandos crearán componentes y servicios en las carpetas correspondientes dentro de `src/app/notes` y `src/app/tags`.

## 8.4. Paso 4: Configurar módulos

Abre los archivos `notes.module.ts` y `tags.module.ts` y configura sus decoradores `@NgModule`:

Archivo `src/app/notes/notes.module.ts`

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { NoteListComponent } from '../note-list/note-list.component';
import { NoteEditorComponent } from '../note-editor/note-editor.component';
import { NoteService } from '../note.service';

@NgModule({
  declarations: [NoteListComponent, NoteEditorComponent],
  imports: [CommonModule],
  providers: [NoteService],
  exports: [NoteListComponent, NoteEditorComponent]
})
export class NotesModule { }
```

Archivo `src/app/tags/tags.module.ts`

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { TagListComponent } from '../tag-list/tag-list.component';
import { TagService } from '../tag.service';

@NgModule({
  declarations: [TagListComponent],
  imports: [CommonModule],
  providers: [TagService],
  exports: [TagListComponent]
})

export class TagsModule { }
```

## 8.5. Paso 5: Importar m-dulos en AppModule

Abre el archivo `src/app/app.module.ts` e importa los m-dulos `NotesModule` y `TagsModule`:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { NotesModule } from './notes/notes.module';
import { TagsModule } from './tags/tags.module';

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

## 8.6. Paso 6: Utilizar componentes de los m-dulos

Abre el archivo `src/app/app.component.html` y utiliza los selectores de los componentes de los m-dulos de notas y etiquetas:

```
<h1>Aplicaci3n de Notas</h1>
<h2>Notas</h2>
<app-note-list></app-note-list>
<app-note-editor></app-note-editor>
```

## 8.7. Paso 7: Implementar la l-gica de los servicios y componentes

A continuaci3n, implementaremos la l-gica b1sica para gestionar notas y etiquetas en sus respectivos servicios y componentes.

Archivo `src/app/notes/note.service.ts`

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

export interface Note {
  id: number;
  title: string;
  content: string;
}
```

```

@Injectable({
  providedIn: 'root',
})
export class NoteService {
  private notes: Note[] = [];

  getNotes(): Note[] {
    return this.notes;
  }

  addNote(note: Note): void {
    this.notes.push(note);
  }

  deleteNote(id: number): void {
    this.notes = this.notes.filter((note) => note.id !== id);
  }
}

```

Archivo src/app/tags/tag.service.ts

```

import { Injectable } from '@angular/core';

export interface Tag {
  id: number;
  name: string;
}

@Injectable({
  providedIn: 'root',
})
export class TagService {
  private tags: Tag[] = [];

  getTags(): Tag[] {
    return this.tags;
  }

  addTag(tag: Tag): void {
    this.tags.push(tag);
  }

  deleteTag(id: number): void {
    this.tags = this.tags.filter((tag) => tag.id !== id);
  }
}

```

Archivo src/app/notes/note-list/note-list.component.ts

```

import { Component, OnInit } from '@angular/core';
import { Note, NoteService } from '../note.service';

```

```

@Component({
  selector: 'app-note-list',
  templateUrl: './note-list.component.html',
  styleUrls: ['./note-list.component.css'],
})
export class NoteListComponent implements OnInit {
  notes: Note[] = [];

  constructor(private noteService: NoteService) {}

  ngOnInit(): void {
    this.notes = this.noteService.getNotes();
  }

  deleteNote(id: number): void {
    this.noteService.deleteNote(id);
    this.notes = this.noteService.getNotes();
  }
}

```

Archivo src/app/notes/note-editor/note-editor.component.ts

```

import { Component } from '@angular/core';
import { NoteService } from '../note.service';

@Component({
  selector: 'app-note-editor',
  templateUrl: './note-editor.component.html',
  styleUrls: ['./note-editor.component.css'],
})
export class NoteEditorComponent {
  title = '';
  content = '';

  constructor(private noteService: NoteService) {}

  addNote(): void {
    if (this.title.trim() && this.content.trim()) {
      const newNote = {
        id: Date.now(),
        title: this.title,
        content: this.content,
      };
      this.noteService.addNote(newNote);
      this.title = '';
      this.content = '';
    }
  }
}

```

src/app/tags/tag-list/tag-list.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Tag, TagService } from '../tag.service';

@Component({
  selector: 'app-tag-list',
  templateUrl: './tag-list.component.html',
  styleUrls: ['./tag-list.component.css'],
})
export class TagListComponent implements OnInit {
  tags: Tag[] = [];

  constructor(private tagService: TagService) {}

  ngOnInit(): void {
    this.tags = this.tagService.getTags();
  }
  deleteTag(id: number): void {
    this.tagService.deleteTag(id);
    this.tags = this.tagService.getTags();
  }
}
```

## 8.8. Paso 8: Crear las plantillas HTML de los componentes

Implementa las plantillas HTML de los componentes `NoteListComponent` y `NoteEditorComponent` y `TagListComponent`.

src/app/notes/note-list/note-list.component.html

```
<ul>
  <li *ngFor="let note of notes">
    {{ note.title }} - {{ note.content }}
    <button (click)="deleteNote(note.id)">Eliminar</button>
  </li>
</ul>
```

src/app/notes/note-editor/note-editor.component.html

```
<div>
  <label for="title">Titulo: </label>
  <input [(ngModel)]="title" id="title" type="text">
</div>
<div>
  <label for="content">Contenido: </label>
```



```

    <textarea [(ngModel)]="content" id="content"></textarea>
  </div>
  <button (click)="addNote()">Agregar nota</button>

```

src/app/tags/tag-list/tag-list.component.html

```

<ul>
  <li *ngFor="let tag of tags">
    {{ tag.name }}
    <button (click)="deleteTag(tag.id)">Eliminar</button>
  </li>
</ul>

```

### Paso 9: Importar FormsModule y HttpClientModule

Para utilizar `[(ngModel)]` en `NoteEditorComponent` y `TagListComponent`, necesitamos importar el módulo `FormsModule`.

Además, si deseas utilizar servicios HTTP en tu aplicación, también debes importar el módulo `HttpClientModule`.

Abre el archivo `src/app/app.module.ts` e importa ambos módulos:

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/http';

import { AppComponent } from './app.component';
import { NotesModule } from './notes/notes.module';
import { TagsModule } from './tags/tags.module';
@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, FormsModule, HttpClientModule, NotesModule, TagsModule],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Con estos pasos, hemos creado una aplicación de notas básica utilizando módulos en Angular. Se puede ejecutar la aplicación con `ng serve` y verla en acción en el navegador.

## 9. Ejemplo @Input y @Output

`@Input` y `@Output` son decoradores en Angular que se utilizan para la comunicación entre componentes.

`@Input` se utiliza para pasar datos de un componente padre a un componente hijo, mientras que `@Output` se utiliza para emitir eventos desde un componente hijo al componente padre.

Aquí tienes un ejemplo detallado que ilustra el uso de `@Input` y `@Output` en Angular:

Crea un componente `ParentComponent` que incluya el componente hijo `ChildComponent`:

```
// src/app/parent/parent.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-parent',
  template: `
    <div>
      <h1>Componente Padre</h1>
      <p>Mensaje del componente padre: {{ parentMessage }}</p>
      <app-child [childInput]="parentMessage"
        (childOutput)="handleChildOutput($event)"></app-child>
    </div>
  `,
})
export class ParentComponent {
  parentMessage = 'Mensaje enviado desde el componente padre';

  handleChildOutput(childMessage: string): void {
    console.log('Mensaje del componente hijo:', childMessage);
  }
}
```

En este ejemplo, la variable `parentMessage` se pasa al componente hijo como un `@Input`. También tenemos un método `handleChildOutput` que manejará el evento emitido por el componente hijo.

Crea un componente `ChildComponent` que recibirá datos del componente padre y emitirá un evento al componente padre:

```
// src/app/child/child.component.ts
import { Component, EventEmitter, Input, Output } from '@angular/core';

@Component({
  selector: 'app-child',
  template: `
    <div>
      <h2>Componente Hijo</h2>
      <p>Mensaje recibido desde el componente padre: {{ childInput }}</p>
      <button (click)="emitChildMessage()">Enviar mensaje al componente
        padre</button>
    </div>
  `,
})
```

```

export class ChildComponent {
  @Input() childInput: string;
  @Output() childOutput = new EventEmitter<string>();

  emitChildMessage(): void {
    const message = 'Mensaje enviado desde el componente hijo';
    this.childOutput.emit(message);
  }
}

```

En `ChildComponent`, utilizamos el decorador `@Input` para recibir datos del componente padre y el decorador `@Output` para emitir eventos al componente padre.

Cuando se hace clic en el botón, se llama al método `emitChildMessage`, que emite un evento al componente padre con un mensaje.

Asegúrate de que `ParentComponent` y `ChildComponent` estén declarados y exportados en el módulo correspondiente.

Por ejemplo, en `app.module.ts`:

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { ParentComponent } from './parent/parent.component';
import { ChildComponent } from './child/child.component';

@NgModule({
  declarations: [
    AppComponent,
    ParentComponent,
    ChildComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Ahora, puedes utilizar el componente padre en `app.component.html`:

```
<app-parent></app-parent>
```

Este ejemplo muestra cómo utilizar `@Input` y `@Output` en Angular para comunicarse entre componentes. El componente padre pasa datos al componente hijo utilizando un `@Input` y recibe

eventos del componente hijo utilizando un @Output.

## 10. Ejemplo ecommerce

En este ejemplo de una aplicación de comercio electrónico, utilizaremos @Input y @Output para comunicarnos entre componentes para mostrar detalles del producto y permitir que los usuarios agreguen productos al carrito de compras.

Crea un componente ProductListComponent que mostrará una lista de productos y permitirá al usuario seleccionar un producto para ver más detalles:

```
// src/app/product-list/product-list.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-product-list',
  template: `
    <div>
      <h1>Lista de productos</h1>
      <ul>
        <li *ngFor="let product of products" (click)="selectProduct(product)">
          {{ product.name }}
        </li>
      </ul>
      <app-product-detail *ngIf="selectedProduct" [product]="selectedProduct"
      (addToCart)="addToCart($event)"></app-product-detail>
    </div>
  `,
})
export class ProductListComponent {

  products = [
    { id: 1, name: 'Producto 1', description: 'Descripción del Producto 1', price: 100 },
    { id: 2, name: 'Producto 2', description: 'Descripción del Producto 2', price: 150 },
  ];

  selectedProduct = null;

  selectProduct(product): void {
    this.selectedProduct = product;
  }

  addToCart(product): void {
    console.log('Producto añadido al carrito:', product);
  }
}
```

Aquí, el componente `ProductListComponent` muestra una lista de productos y permite al usuario seleccionar un producto.

Cuando se selecciona un producto, se muestra el componente `ProductDetailComponent` con información detallada del producto.

Crea un componente `ProductDetailComponent` que mostrará los detalles del producto y permitirá al usuario agregar el producto al carrito de compras:

```
// src/app/product-detail/product-detail.component.ts
import { Component, EventEmitter, Input, Output } from '@angular/core';

@Component({
  selector: 'app-product-detail',
  template: `
    <div>
      <h2>Detalles del producto</h2>
      <p>Nombre: {{ product.name }}</p>
      <p>Descripción: {{ product.description }}</p>
      <p>Precio: {{ product.price }}</p>
      <button (click)="addToCart.emit(product)">Agregar al carrito</button>
    </div>
  `,
})
export class ProductDetailComponent {

  @Input() product: any;
  @Output() addToCart = new EventEmitter<any>();
}
```

En `ProductDetailComponent`, utilizamos el decorador `@Input` para recibir el producto seleccionado del componente padre (`ProductListComponent`) y el decorador `@Output` para emitir un evento cuando el usuario agrega un producto al carrito de compras.

Asegúrate de que `ProductListComponent` y `ProductDetailComponent` estén declarados y exportados en el módulo correspondiente.

Por ejemplo, en `app.module.ts`:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { ProductListComponent } from './product-list/product-list.component';
import { ProductDetailComponent } from './product-detail/product-detail.component';

@NgModule({
  declarations: [
    AppComponent,
```

```

    ProductListComponent,
    ProductDetailComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Ahora, puedes utilizar el componente `ProductListComponent` en `app.component.html`:

```
<app-product-list></app-product-list>
```

Con este ejemplo, hemos creado una aplicación de comercio electrónico simple que muestra una lista de productos y permite al usuario seleccionar un producto para ver más detalles. Cuando se selecciona un producto, se muestra el componente `ProductDetailComponent` con información detallada del producto.

La comunicación entre el componente `ProductListComponent` (padre) y el componente `ProductDetailComponent` (hijo) se realiza utilizando `@Input` y `@Output`.

El componente padre pasa el producto seleccionado al componente hijo utilizando un `@Input` y recibe eventos del componente hijo cuando el usuario agrega un producto al carrito utilizando un `@Output`.