

## Requisitos técnicos

Este capítulo contiene varios ejemplos de código que te guiarán en el enruteamiento en el framework Angular. Puedes encontrar el código fuente relacionado en la carpeta ch09 del siguiente repositorio de GitHub:

<https://www.github.com/PacktPublishing/Aprendizaje-Angular-Quinta-Edición>

## Presentamos el enruteador Angular

En las aplicaciones web tradicionales, al cambiar de una vista a otra, necesitábamos solicitar una nueva página al servidor. El navegador creaba una URL para la vista y la enviaba al servidor. El navegador recargaba la página en cuanto el cliente recibía una respuesta. Este proceso generaba retrasos en el proceso de ida y vuelta y una mala experiencia de usuario para nuestras aplicaciones:

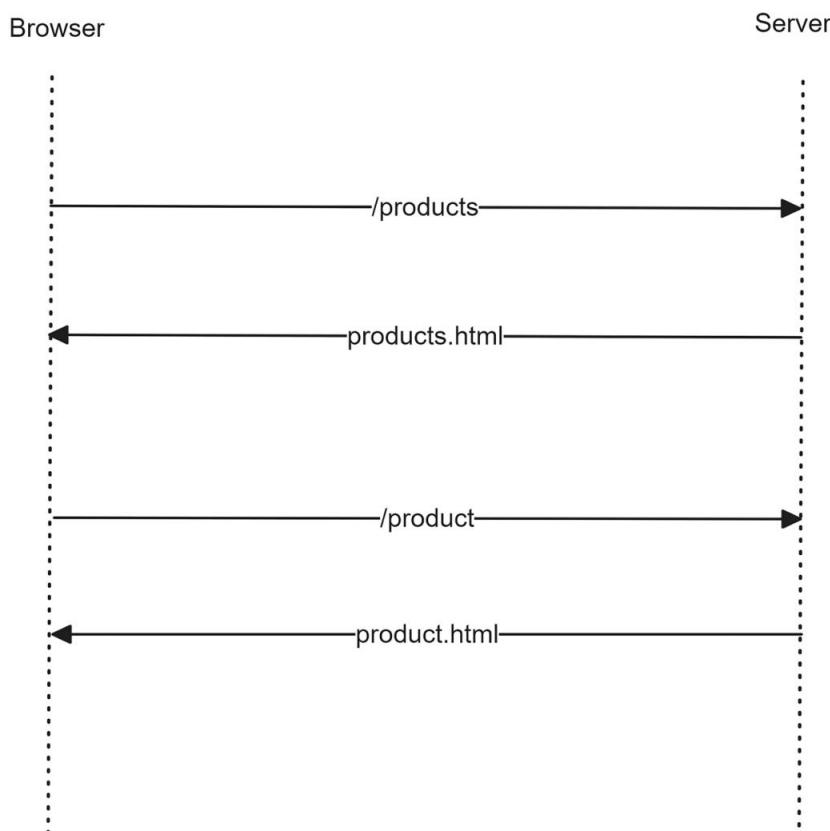


Figura 9.1: Enrutamiento en aplicaciones web tradicionales

Las aplicaciones web modernas que utilizan marcos de JavaScript como Angular siguen un enfoque diferente. Gestionan cambios entre vistas o componentes en el lado del cliente sin afectar al servidor. Se comunican con el servidor una vez durante el arranque para obtener el archivo HTML principal. El enrutador del cliente intercepta y gestiona cualquier cambio de URL posterior. Estas aplicaciones se denominan Aplicaciones de Página Única (SPA) porque no provocan la recarga completa de la página.

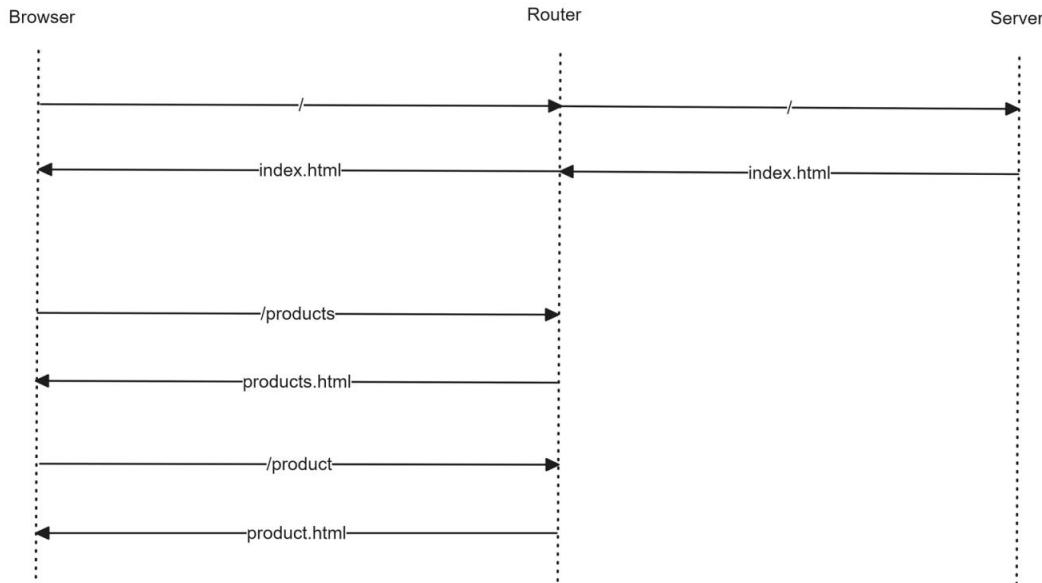


Figura 9.2: Arquitectura de SPA

El marco Angular proporciona el paquete npm `@angular/router`, que podemos usar para navegar entre diferentes componentes en una aplicación Angular.

Agregar enrutamiento en una aplicación Angular implica los siguientes pasos:

1. Especificación de la ruta base para la aplicación Angular
2. Usando una directiva o servicio apropiado del paquete npm `@angular/router`
3. Configurar diferentes rutas para la aplicación Angular
4. Decidir dónde representar los componentes durante la navegación

En las siguientes secciones, aprenderemos los conceptos básicos del enrutamiento angular antes de profundizar en ejemplos prácticos.

## Especificación de una ruta base

Como ya hemos visto, las aplicaciones web modernas y tradicionales reaccionan de forma diferente cuando una URL cambia dentro de la aplicación. La arquitectura de cada navegador juega un papel esencial en este comportamiento. Los navegadores más antiguos inician una nueva solicitud al servidor cuando cambia la URL. Los navegadores modernos, también conocidos como navegadores evergreen , pueden cambiar la URL y el historial del navegador al navegar en diferentes vistas sin enviar una solicitud al servidor mediante una técnica llamada pushState.



HTML5 pushState permite la navegación dentro de la aplicación sin provocar una recarga completa de una página y es compatible con todos los navegadores modernos.

Una aplicación Angular debe configurar la etiqueta HTML <base> en el archivo index.html para habilitar el enrutamiento pushState:

```
<!doctype html>
<html lang="es">
<cabeza>
  <meta charset="utf-8">
  <title>Mi aplicación</title>
  <base href="/">
  <meta name="viewport" content="ancho=ancho-del-dispositivo, escala-inicial=1">
  <link rel="icon" tipo="imagen/x-icono" href="favicon.ico">
</cabeza>
<cuerpo>
  <app-root></app-root>
</cuerpo>
</html>
```

El atributo href informa al navegador la ruta que debe seguir al cargar los recursos de la aplicación. La CLI de Angular agrega automáticamente la etiqueta al crear una nueva aplicación y establece el valor href en la raíz de la aplicación, /. Si su aplicación reside en una carpeta distinta a la raíz, debe asignarle el mismo nombre.

## Habilitación del enrutamiento en aplicaciones Angular

El enrutador Angular está habilitado de forma predeterminada en las nuevas aplicaciones Angular, como lo indica el método provideRouter en el archivo app.config.ts :

```

importar { ApplicationConfig, provideZoneChangeDetection } desde '@angular/
centro';
importar { provideRouter } desde '@angular/router';

importar { rutas } desde './app.routes';

exportar const appConfig: ApplicationConfig = {
  proveedores: [
    proporcionarZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(rutas)
  ]
};

```



En aplicaciones creadas con versiones anteriores del marco Angular, el enrutador se habilita importando la clase RouterModule en el módulo de la aplicación principal y usando su método forRoot para definir la configuración de enrutamiento.

El método provideRouter nos permite utilizar un conjunto de artefactos angulares relacionados con el enrutamiento:

- Servicios para realizar tareas de enrutamiento comunes, como navegación.
- Directivas que podemos utilizar en nuestros componentes para enriquecerlos con lógica de navegación

Acepta un único parámetro, que es la configuración de enrutamiento de la aplicación, y se define por defecto en el archivo app.routes.ts .

## Configurando el enrutador

El archivo app.routes.ts contiene una lista de objetos Routes que especifican qué rutas existen en la aplicación y qué componentes deben responder a cada una. Su aspecto es el siguiente:

```

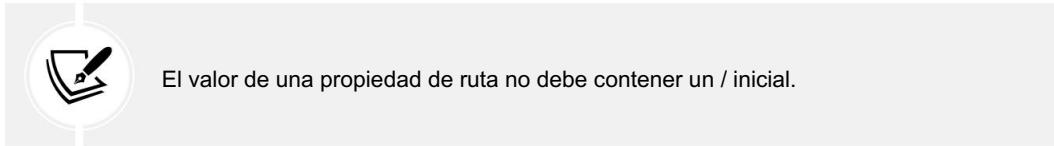
const rutas: Rutas = [
  { ruta: 'productos', componente: ProductListComponent },
  { ruta: '**', componente: PageNotFoundComponent }
];

```



En aplicaciones creadas con versiones anteriores del marco Angular, es posible que notes que la configuración de la ruta está definida en un archivo app-routing.module.ts dedicado .

Cada objeto de definición de ruta contiene una propiedad de ruta , que es la ruta URL de la ruta, y una propiedad de componente que define qué componente se cargará cuando la aplicación navegue a esa ruta.



El valor de una propiedad de ruta no debe contener un / inicial.

La navegación en una aplicación Angular puede realizarse manualmente modificando la URL del navegador o usando enlaces dentro de la aplicación. En el primer caso, el navegador recargará la aplicación, mientras que en el segundo, indicará al enrutador que navegue en tiempo de ejecución. En nuestro caso, cuando la URL del navegador contiene la ruta de los productos , el enrutador muestra el componente de lista de productos en la página. Por el contrario, cuando la aplicación navega a los productos mediante código, el enrutador sigue el mismo procedimiento y actualiza la URL del navegador.

Si el usuario intenta navegar a una URL que no coincide con ninguna ruta, Angular activa un tipo de ruta personalizado llamado comodín o de respaldo. La ruta comodín tiene una propiedad de ruta con dos asteriscos y coincide con cualquier URL. La propiedad del componente para esto suele ser un componente PageNotFoundComponent específico de la aplicación o el componente principal de la aplicación.

## Componentes de renderizado

La plantilla del componente principal de la aplicación contiene el elemento <router-outlet> , una de las directivas principales del enrutador Angular. Se encuentra dentro de app.component.html. Este archivo se utiliza como marcador de posición para los componentes activados con el enrutamiento. Estos componentes se representan como un elemento hermano del elemento <router-outlet> .

Hemos cubierto los conceptos básicos y proporcionado una configuración mínima de enrutador. En la siguiente sección, analizaremos un ejemplo más realista y ampliaremos nuestros conocimientos sobre enrutamiento.

## Configurando las rutas principales

Al comenzar a diseñar la arquitectura de una aplicación Angular con enrutamiento, lo más fácil es pensar en sus características principales, como los enlaces de menú a los que los usuarios pueden acceder haciendo clic. Los productos y los carritos de compra son características básicas de la aplicación de tienda online que estamos desarrollando. Añadir enlaces y configurarlos para activar ciertas funciones de una aplicación Angular forma parte de la configuración de enrutamiento de la aplicación.



Necesitará el código fuente de la aplicación Angular que creamos en el Capítulo 8, "Comunicación con Servicios de Datos a través de HTTP", para continuar con el resto del capítulo. Después de obtener el código, le sugerimos que realice las siguientes acciones para simplificar:

- Elimine auth.interceptor.ts y su archivo de prueba unitaria. Llamadas reales en el La API de tienda falsa no necesita autenticación.
- Modifique el archivo app.config.ts para que el método provideHttpClient no utilice el interceptor.

Para configurar la ruta de nuestra aplicación debemos seguir los siguientes pasos:

1. Ejecute el siguiente comando para crear un nuevo componente Angular para el carrito de compras:

```
ng generar componente carrito
```

2. Abra el archivo app.routes.ts y agregue las siguientes declaraciones de importación :

```
importar { CartComponent } desde './cart/cart.component';
importar { ProductListComponent } desde './product-list/product-list.component';
```

3. Agregue dos objetos de definición de ruta en la variable rutas :

```
exportar const rutas: Rutas = [ { ruta:
  'productos', componente: ProductListComponent }, { ruta: 'carrito',
  componente: CartComponent } ];
```

En el fragmento anterior, la ruta de productos activará ProductListComponent y la ruta de carrito activará CartComponent.

4. Abra el archivo app.component.html y modifique el elemento HTML <header> de la siguiente manera:

```
<encabezado>
<h2>{{configuración.título}}</h2> <span
class="spacer">
<div class="enlaces-de-menú">
  <a routerLink="/products"> Productos</a>
  <a routerLink="/cart"> Mi carrito</a>
</div>
```

```
<app-auth></app-auth> </header>
```

En la plantilla anterior, aplicamos la directiva routerLink para anclar elementos HTML y asignar la ruta que queremos navegar. Observe que la ruta debe comenzar con /, a diferencia de la propiedad path en el objeto de definición de ruta.

Cómo comienza la ruta depende de si queremos utilizar enruteamiento absoluto o relativo en nuestra aplicación, como aprenderemos más adelante en el capítulo.

- Mueva el elemento HTML <router-outlet> dentro del elemento <div> con el contenido selector de clase y eliminar el componente <app-product-list> :

```
<main class="principal">
  <div class="contenido">
    <enrutador-de-salida />
  </div>
</principal>
```

- Abra el archivo app.component.ts , elimine cualquier referencia a la clase ProductListComponent e importe la clase RouterLink :

```
importar { Componente, inyectar } desde '@angular/core'; importar { RouterLink,
RouterOutlet } desde '@angular/router';
importar { CopyrightDirective } desde './copyright.directive'; importar { APP_SETTINGS } desde
'./app.settings';
importar { AuthComponent } desde './auth/auth.component';

@Component({ selector: 'app-root',
importaciones: [
  Salida de enrutador,
  Enlace de enrutador,
  Directiva sobre derechos de autor,
  Componente de
  autenticación ],
URL de plantilla: './app.component.html', URL de estilo:
'./app.component.css' })
```

7. Abra el archivo app.component.css y reemplace todos los estilos CSS relacionados con .social-links Selector con los siguientes estilos:

```

encabezado {
    pantalla: flex; flex-
    direccion: fila;
    espacio: 0,73rem;
    justificar-contenido: fin;
    margen superior: 1.5rem;
}

.menu-links {
    pantalla: flex;
    alinear-elementos: centro;
    brecha: 0,73rem;
}

.menu-links a {
    transición: relleno 0,3s facilidad;
    color: var(--gris-400);
}

.menu-links a:hover {
    color: var(--gris-900);
}

```

8. Finalmente, abra el archivo global style.css y agregue los siguientes estilos CSS:

```

a {
    decoración de texto: ninguna;

} .spacer
    { flex: 1 1 auto;
}

```

Ahora estamos listos para obtener una vista previa de nuestra aplicación Angular:

- Ejecute el comando ng serve y navegue a <http://localhost:4200>. Inicialmente, la página de la aplicación muestra únicamente el encabezado de la aplicación y la información de derechos de autor.

2. Haga clic en el enlace Productos . La aplicación debería mostrar la lista de productos y actualizarla.  
la URL del navegador para que coincida con la ruta /productos .
3. Ahora navegue a la ruta raíz en <http://localhost:4200> y añada la ruta /cart al final de la URL del navegador. La aplicación debería reemplazar la vista de lista de productos con el componente del carrito:

¡El carrito funciona!



El enruteamiento en Angular funciona bidireccionalmente. Nos permite acceder a un componente Angular mediante los enlaces de la aplicación o la barra de direcciones del navegador.

¡Felicitaciones! Tu aplicación Angular ahora admite navegación dentro de la aplicación.

Apenas hemos explorado el enruteamiento de Angular. Hay muchas características que podemos investigar en las siguientes secciones. Por ahora, intentemos dividir nuestros componentes en más rutas para facilitar su gestión.

## Organización de rutas de aplicaciones

Nuestra aplicación muestra la lista de productos junto con sus detalles y los componentes que los crean. Necesitamos organizar la configuración de enruteamiento para que diferentes rutas activen cada componente.

En esta sección, agregaremos una nueva ruta para el componente de creación de producto. Más adelante, en la sección "Pasar parámetros a las rutas" , agregaremos una ruta independiente para el componente de detalles del producto.

Comencemos con el componente de creación de producto:

1. Abra el archivo app.routes.ts y agregue la siguiente declaración de importación :

```
importar { ProductCreateComponent } desde './product-create/product-create.component';
```

2. Agregue el siguiente objeto de definición de ruta en la variable rutas :

```
{ ruta: 'productos/nuevo', componente: ProductCreateComponent }
```

3. Abra el archivo product-list.component.ts y elimine cualquier referencia a Clase ProductCreateComponent .
4. Abra el archivo product-list.component.html y elimine la etiqueta <app-product-create> elemento.

- Ejecute el comando `ng serve` para iniciar la aplicación, haga clic en el enlace Productos y verifique que no se muestra el formulario de creación de producto.

Actualmente, solo se puede acceder al componente de creación de producto mediante la URL del navegador y no podemos acceder a él mediante la interfaz de usuario de la aplicación. En la siguiente sección, aprenderemos cómo realizar esta tarea y navegar imperativamente a una ruta.

## Navegar imperativamente hacia una ruta

El componente de creación de producto solo se puede activar ingresando la dirección `http://localhost:4200/products/new` en la barra de direcciones del navegador. Agreguemos un botón en la lista de productos que nos permitirá navegar también desde la interfaz de usuario:

- Abra el archivo `product-list.component.html` y modifique el segundo bloque `@if` de la siguiente manera:



El elemento `<path>` a continuación puede ser difícil de escribir manualmente. Como alternativa, puede encontrar el código en la carpeta ch09 del repositorio de GitHub del libro y copiarlo desde allí.

```
@if (productos) {  
  <div class="caption">  
    Productos ({products.length})  
    <a routerLink="nuevo">  
      <svg  
        ancho="24"  
        altura="24"  
        xmlns="http://www.w3.org/2000/svg"  
        regla de relleno="evenodd"  
        regla de clip="evenodd">  
        <path d="M11.5 0c6.347 0 11.5 5.153 11.5 11.5s-5.153 11.5-11.5 11.5-5.153-11.5 5.153-11.5c11.5-11.5zm0 1c5.795 0 10.5 4.705 10.5 10.5s-4.705 10.5 10.5-10.5s-10.5 4.705-10.5 10.5c4.705-10.5 10.5-10.5zm.5 10h6v1h-6v6h-1v-6h-6v-1h6v-6h1v6z"/>  
      </svg>  
    </a>  
  </div>  
}
```

En el fragmento anterior, agregamos un elemento de anclaje que nos llevará al componente de creación del producto, como lo indica el valor de la directiva `routerLink`.



El valor de la directiva routerLink es nuevo y no /products/new, como cabría esperar. El comportamiento anterior se debe a que el botón reside en el componente de lista de productos, que ya está activado por los productos. parte de la ruta.

El enrutador Angular puede sintetizar la ruta de destino por todas las rutas activadas, pero si no desea comenzar desde la raíz, puede agregar un / antes la ruta.

2. Abra el archivo product-list.component.css y agregue los siguientes estilos CSS:

```
.subtítulo {
    pantalla: flexible;
    alinear-elementos: centro;
    brecha: 1,25rem;
}

camino {
    transición: relleno 0,3s facilidad;
    relleno: var(--gray-400);
}

a:hover ruta de SVG {
    relleno: var(--gray-900);
}
```

3. Abra el archivo product-list.component.ts y agregue la siguiente declaración de importación :

```
importar { RouterLink } desde '@angular/router';
```

4. Agregue la clase RouterLink en la matriz de importaciones del decorador @Component :

```
@Component({
    selector: 'lista-de-productos-de-aplicaciones',
    importaciones: [
        ComponenteDetalleDeProducto,
        SortPipe,
        Tubería asíncrona,
        Enlace de enrutador
```

```
    ],
    URL de plantilla: './lista-de-productos.componente.html',
    styleUrl: './product-list.component.css' })
```

5. Abra el archivo product-create.component.css y agregue el siguiente estilo CSS:

```
:host
{ ancho: 400px;
}
```

En el estilo anterior, el selector :host apunta al elemento host del producto creado. componente.

6. Ejecute el comando ng serve para iniciar la aplicación y navegue a `http://localhost:4200/productos`:

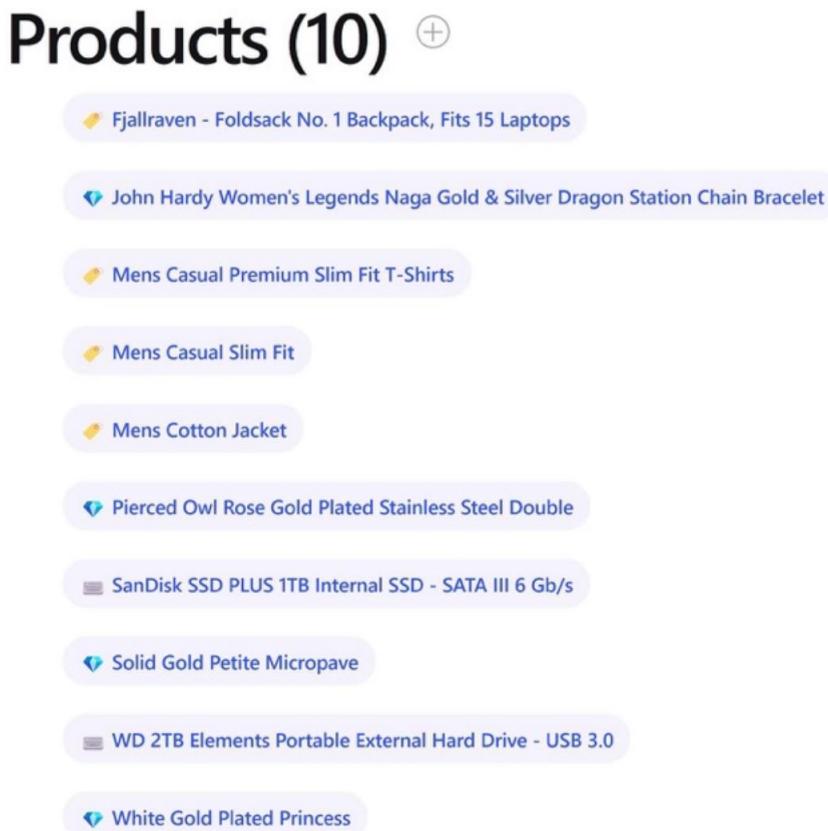


Figura 9.3: Lista de productos

7. Haga clic en el botón con el signo más. La aplicación le redirige a /productos/nuevo ruta y activa el componente de creación de producto:

## Add new product

Title

Price

Category

Select a category

**Create**

Figura 9.4: Formulario de creación de producto

Aunque el componente de creación de productos sigue funcionando, nuestro cambio introdujo una falla en la experiencia de usuario (UX) de la aplicación. El usuario no recibe una indicación visual cuando se crea un nuevo producto, ya que la lista de productos pertenece a una ruta diferente. Debemos modificar la lógica del botón "Crear" para que redirija al usuario a la lista de productos tras la creación correcta de un producto.

1. Abra el archivo product-create.component.ts y agregue la siguiente declaración de importación :

```
importar { Router } desde '@angular/router';
```

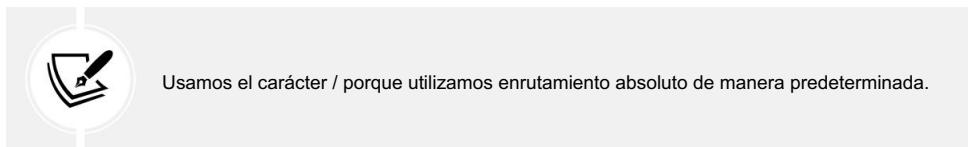
2. Inyecte el servicio Router en el constructor de la clase ProductCreateComponent :

```
constructor( productosServicioprivado: ProductosServicio, enrutador privado:  
Enrutador ) {
```

3. Modifique el método createProduct de la siguiente manera:

```
createProduct(título: cadena, precio: cadena, categoría: cadena) {  
    este.productsService.addProduct({  
        título,  
        precio: Número(precio),  
        categoría  
    }).subscribe(() => this.router.navigate(['/productos']));  
}
```

En el método anterior, llamamos al método de navegación del servicio Router para navegar a la ruta / products de la aplicación.



Usamos el carácter / porque utilizamos enrutamiento absoluto de manera predeterminada.

Acepta una matriz de parámetros de enlace que contiene la ruta de destino que queremos navegar.

4. Abra el archivo products.service.ts y modifique el método getProducts para que utilice

La API de tienda falsa cuando no hay datos de productos locales:

```
obtenerProductos(): Observable<Producto[]> {
  si (este.producto.longitud === 0) {
    const opciones = new HttpParams().set('limit', 10);
    devuelve este.http.get<Product[]>(este.productsUrl, {
      parámetros: opciones
    }).pipe(map(productos => {
      este.productos = productos;
      devolver productos;
    }));
  }
  retorno de(este.productos);
}
```

Si no realizamos el cambio anterior, el componente de lista de productos siempre retornará datos de la API de Fake Store.

Nuestra aplicación ahora redirige a los usuarios a la lista de productos cada vez que crean un nuevo producto para que puedan verlo en la lista.

Hasta ahora, hemos configurado el enrutamiento de la aplicación para activar los componentes según una ruta determinada. Sin embargo, nuestra aplicación no muestra ningún componente en las siguientes situaciones:

- Cuando navegamos a la ruta raíz de la aplicación
- Cuando intentamos navegar hacia una ruta inexistente

En la siguiente sección, aprenderemos cómo utilizar las rutas integradas que proporciona Angular Router y mejorar la experiencia de usuario (UX) de la aplicación.

## Uso de rutas integradas

Para definir un componente que se cargará al navegar a la ruta raíz, creamos un objeto de definición de ruta y asignamos la propiedad `path` a una cadena vacía. Una ruta con `path` vacía se denomina ruta predeterminada de la aplicación Angular.

En nuestro caso, queremos que la ruta predeterminada muestre el componente de lista de productos. Abra app.routes. archivo ts y agregue la siguiente ruta al final de la variable rutas :

```
{ ruta: "", redirección a: 'productos', coincidencia de ruta: 'completo' }
```

En el fragmento anterior, indicamos al enrutador que redirija a la ruta de los productos cuando la aplicación navegue a la ruta predeterminada. La propiedad pathMatch indica al enrutador cómo hacer coincidir la URL con la propiedad de la ruta raíz. En este caso, el enrutador redirige a la ruta de los productos solo cuando la URL coincide con la ruta raíz, que es la cadena vacía.

Si ejecutamos la aplicación, notaremos que cuando la URL del navegador apunta a la ruta raíz de nuestra aplicación, somos redirigidos a la ruta de los productos y se muestra la lista de productos en la pantalla.



Agregamos la ruta predeterminada después de todas las demás rutas porque el orden de las rutas es

Importante. El enrutador selecciona rutas con una estrategia de "primera coincidencia gana". Se deben definir rutas más específicas antes que las menos específicas.

Ya vimos el concepto de rutas desconocidas en la sección "Introducción al enrutador Angular" . Vimos brevemente cómo configurar una ruta comodín para mostrar un componente PageNotFoundComponent cuando nuestra aplicación intenta navegar a una ruta inexistente. En aplicaciones reales, es común crear este tipo de componente, especialmente si se desea mostrar información adicional al usuario, como los pasos a seguir. En nuestro caso, que es más sencillo, redirigiremos a la ruta de productos .

Abra el archivo app.routes.ts y agregue la siguiente ruta al final de la variable rutas :

```
{ ruta: "**", redirección a: 'productos' }
```



La ruta comodín debe ser la última entrada en la lista de rutas porque la aplicación solo debe acceder a ella si no hay rutas coincidentes.

Si ejecutamos nuestra aplicación usando el comando ng serve y navegamos a una ruta desconocida, nuestra aplicación mostrará la lista de productos.

Hasta ahora, nos hemos basado en la barra de direcciones del navegador para indicar qué ruta está activa en cada momento. Como veremos en la siguiente sección, podemos mejorar la experiencia del usuario mediante estilos CSS.

### Estilizar enlaces de enrutador

El encabezado de la aplicación contiene los enlaces Productos y Mi carrito . Al navegar a cada uno, no queda claro qué ruta se ha activado. El enrutador Angular exporta el enlace routerLinkActive. directiva, que podemos utilizar para cambiar el estilo de un enlace cuando la ruta correspondiente está activa. Funciona de forma similar al enlace de clases que vimos en el Capítulo 3, "Estructura de interfaces de usuario con componentes". Acepta una lista de nombres de clases o una clase que se añade cuando el enlace está activo. y se elimina cuando se vuelve inactivo.

Veamos cómo usarlo en nuestra aplicación:

1. Abra el archivo app.component.css y agregue el siguiente estilo CSS:

```
.menu-links a.active {  
    color: var(--violeta eléctrico);  
}
```

2. Abra el archivo app.component.ts e importe la clase RouterLinkActive desde el

Paquete npm @angular/router :

```
importar { RouterLink, RouterLinkActive, RouterOutlet } desde '@  
angular/enrutador';
```

3. Agregue la clase RouterLinkActive en la matriz de importaciones del decorador @Component :

```
@Componente({  
    selector: 'app-root',  
    importaciones: [  
        Salida de enrutador,  
        Enlace de enrutador,  
        Enlace de enrutador activo,  
        Directiva sobre derechos de autor,  
        Componente de autenticación  
    ],  
    URL de plantilla: './app.component.html',  
    URL de estilo: './app.component.css'  
})
```

4. Abra el archivo app.component.html y agregue la directiva routerLinkActive a ambos enlaces:

```
<div class="enlaces-de-menú">  
  <a routerLink="/products" routerLinkActive="active"> Productos</a>  
  <a routerLink="/cart" routerLinkActive="active"> Mi carrito</a>  
</div>
```

Ahora, cuando hacemos clic en un enlace de aplicación en el encabezado, su color cambia para indicar que el enlace está activo.

Hemos aprendido a utilizar el enrutamiento y a activar componentes que no necesitan ningún parámetro.

Sin embargo, el componente de detalles del producto acepta el ID del producto como parámetro. En la siguiente sección, aprenderemos a activar el componente mediante parámetros de ruta dinámicos.

## Pasando parámetros a rutas

Un escenario común en las aplicaciones web empresariales es tener una lista de elementos y, al hacer clic en uno, la página cambia la vista actual y muestra los detalles del elemento seleccionado. El enfoque anterior se asemeja a una función de navegación maestro-detalle, donde cada URL generada en la página maestra contiene los identificadores necesarios para cargar cada elemento en la página de detalles.

Podemos representar el escenario anterior con dos rutas que navegan a diferentes componentes. Un componente es la lista de elementos y el otro, sus detalles. Por lo tanto, necesitamos encontrar una manera de crear y transferir datos dinámicos específicos de cada elemento de una ruta a la otra.

Nos enfrentamos a un doble problema: crear URL con parámetros dinámicos en tiempo de ejecución y analizar el valor de estos parámetros. No hay problema: el enrutador Angular nos respalda, y veremos cómo con un ejemplo real.

### Creación de una página de detalles utilizando parámetros de ruta

La lista de productos de nuestra aplicación muestra actualmente una lista de productos. Al hacer clic en un producto, sus detalles aparecen debajo. Necesitamos refactorizar el flujo de trabajo anterior para que el componente responsable de mostrar los detalles del producto se renderice en una página diferente de la lista. Usaremos el enrutador Angular para redirigir al usuario a la nueva página al hacer clic en un producto de la lista.

El componente de lista de productos pasa actualmente el ID del producto seleccionado mediante el enlace de entrada. En su lugar, usaremos el enrutador Angular para pasar el ID del producto como parámetro de ruta:

1. Abra el archivo app.routes.ts y agregue la siguiente declaración de importación :

```
importar { ProductDetailComponent } desde './product-detail/product-detail.component';
```

2. Agregue la siguiente definición de ruta en la variable rutas después de la ruta productos/nueva :

```
{ ruta: 'productos/:id', componente: ProductDetailComponent }
```

El carácter de dos puntos denota id como un parámetro de ruta en el nuevo objeto de definición de ruta.

Si una ruta tiene varios parámetros, los separamos con /. Como veremos más adelante, el nombre del parámetro es importante cuando queremos consumir su valor en nuestros componentes.

3. Abra el archivo product-list.component.html y agregue un elemento de anclaje para el producto título para que utilice la nueva definición de ruta:

```
<ul class="grupo de píldoras">
  @for (producto de productos | ordenar; rastrear producto.id) {
    <li class="pill" (clic)="productoSeleccionado = producto">
      @switch (producto.categoría) {
        @case ('electrónica') {  }
        @case ('joyería') {  }
        { @default { }  }
      }
      <a [routerLink]=[product.id]>{{producto.título}}</a>
    </li>
  } @vacío {
    <p>¡No se encontraron productos!</p>
  }
</ul>
```

En el fragmento anterior, la directiva routerLink utiliza la vinculación de propiedades para establecer su valor en una matriz de parámetros de enlace. Pasamos el ID de la variable de referencia de la plantilla de producto como parámetro en la matriz.



No necesitamos anteponer /products al valor de la matriz de parámetros de enlace porque esa ruta ya activa la lista de productos.

4. Elimine el componente <app-product-detail> y el enlace del evento de clic de la

Etiqueta <li> .



Podemos refactorizar el archivo product-list.component.ts y eliminar cualquier código que use la propiedad selectedProduct y ProductDetailComponent Clase. La lista de productos no necesita mantener el producto seleccionado en su estado local, ya que nos alejamos de la lista al elegir un producto.

Ahora podemos proceder a modificar el componente de detalle del producto para que funcione con el enruteamiento:

1. Abra el archivo product-detail.component.css y agregue un estilo CSS para establecer el ancho de el elemento anfitrión:

```
:anfitrión {  
    ancho: 450px;  
}
```

2. Abra el archivo product-detail.component.ts y modifique las declaraciones de importación de la siguiente manera:

mínimos:

```
importar { CommonModule } desde '@angular/common';  
importar { Componente, entrada, OnInit } desde '@angular/core';  
importar { ActivatedRoute, Router } desde '@angular/router';  
importar { Producto } de './producto';  
importar { Observable, switchMap } de 'rxjs';  
importar { ProductsService } desde './products.service';  
importar { AuthService } desde './auth.service';
```

El enrutador Angular exporta el servicio ActivatedRoute , que podemos usar para recuperar información sobre la ruta actualmente activada, incluidos los parámetros.

3. Modifique el constructor del componente para inyectar los servicios ActivatedRoute y Router :

```
constructor(  
    productoServicio privado : ProductosServicio,  
    servicio de autenticación público : AuthService,  
    ruta privada: ActivatedRoute,  
    enrutador privado: enrutador  
) {}
```

4. Modifique la lista de interfaces implementadas de la clase ProductDetailComponent :

La clase de exportación ProductDetailComponent implementa OnInit

5. Cree el siguiente método ngOnInit :

```
ngOnInit(): vacío {  
    este.producto$ = esta.ruta.paramMap.pipe(  
        switchMap(parámetros => {  
            devuelve este.productService.getProduct(Number(params.  
                obtener('id')));  
        })  
    );  
}
```

El servicio ActivatedRoute contiene el observable paramMap , que podemos usar para suscribirnos y obtener los valores de los parámetros de ruta. El operador switchMap de RxJS se usa cuando queremos obtener un valor de un observable, completarlo y pasarlo a otro observable. En este caso, lo usamos para canalizar el parámetro id del observable paramMap al método getProduct de la clase ProductsService .

6. Modifique los métodos changePrice y remove para que la aplicación redirija a la

Lista de productos al finalizar cada acción:

```
changePrice(producto: Producto, precio: cadena) {  
    este.productService.updateProduct(producto.id, Número(precio)).  
    suscribirse() => {  
        este.router.navigate(['/productos']);  
    });  
}  
  
eliminar(producto: Producto) {  
    este.productService.deleteProduct(producto.id).subscribe(() => {  
        este.router.navigate(['/productos']);  
    });  
}
```

7. Elimine el método ngOnChanges porque el componente y sus enlaces se inicializan cada vez que se activa la ruta.

8. Elimine los emisores de eventos de salida, ya que el componente de lista de productos ya no es un componente principal. Deje la propiedad de entrada id tal como está, ya que la usaremos más adelante en este capítulo.
9. Deje el método addToCart vacío por ahora. Lo usaremos más adelante en el Capítulo 10, Recopilación. Datos de usuario con formularios.

También vale la pena señalar lo siguiente:

1. El observable paramMap devuelve un objeto del tipo ParamMap . Podemos usar la función get. método del objeto ParamMap para pasar el nombre del parámetro que definimos en la configuración de la ruta y acceder a su valor.
2. Convertimos el valor del parámetro id en un número porque los valores del parámetro de ruta siempre son cadenas.

Si ejecutamos la aplicación usando el comando ng serve y hacemos clic en un producto de la lista, la aplicación nos lleva al componente de detalles del producto:

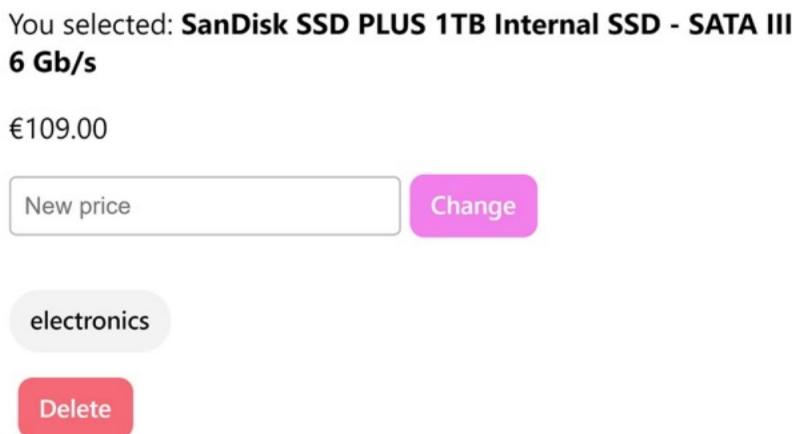


Figura 9.5: Página de detalles del producto

📝

Si actualiza el navegador, la aplicación no mostrará el producto porque el método getProduct de la clase ProductsService solo funciona con la versión en caché de los datos del producto. Debe volver a la lista de productos y seleccionar uno.

porque se ha restablecido la caché local. Tenga en cuenta que este comportamiento se basa en la corriente implementación de alquiler de la aplicación de la tienda electrónica y no está vinculada al enruteador Angular arquitectura.

En el ejemplo anterior, utilizamos la propiedad `paramMap` para obtener los parámetros de ruta como un observable. Idealmente, nuestro componente podría recibir notificaciones de nuevos valores durante su ciclo de vida. Sin embargo, el componente se destruye cada vez que queremos seleccionar un producto diferente de la lista, al igual que la suscripción al observable `paramMap`.

Como alternativa, podemos evitar el uso de observables reutilizando la instancia de un componente en cuanto permanezca renderizado en pantalla durante navegaciones consecutivas. Podemos lograr este comportamiento mediante rutas secundarias, como veremos en la siguiente sección.

### Reutilización de componentes mediante rutas secundarias

Las rutas secundarias son una solución perfecta cuando queremos un componente de página de destino que proporcione enruteamiento a otros componentes. El componente debe contener un elemento `<router-outlet>` en las rutas secundarias que cargarán.

Supongamos que queremos definir el diseño de nuestra aplicación Angular de esta manera:

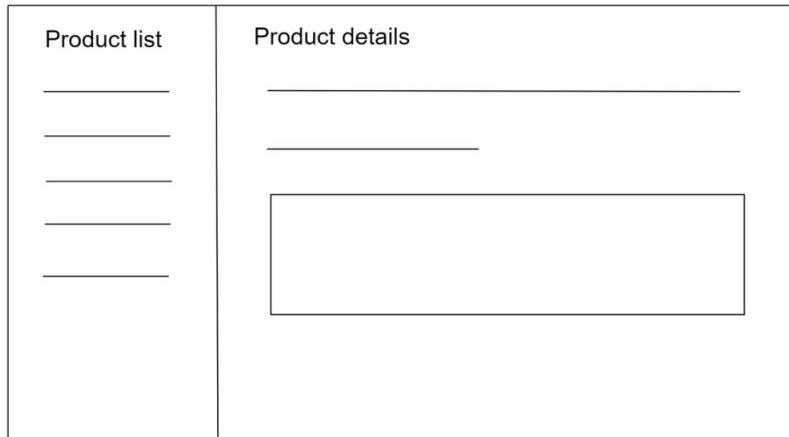


Figura 9.6: Diseño maestro-detalle

El escenario del diagrama anterior requiere que el componente de lista de productos contenga un elemento `<router-outlet>` para representar el componente de detalles del producto cuando se activa la ruta relacionada.

El componente de detalles del producto se representará en el `<router-outlet>` del componente de lista de productos y no en el `<router-outlet>` del componente de aplicación principal.

El componente de detalles del producto no se destruye cuando navegamos de un producto a otro. En cambio, permanece en el árbol DOM y su método `ngOnInit` se llama una vez, la primera vez que seleccionamos un producto. Al elegir un nuevo producto de la lista, el observable `paramMap` emite su `id`. El nuevo producto se obtiene mediante la clase `ProductsService` y la plantilla del componente se actualiza para reflejar los cambios.

La configuración de ruta de la aplicación, en este caso, sería la siguiente:

```
exportar const rutas: Rutas = [
  {
    ruta: 'productos',
    componente: ProductListComponent,
    niños: [
      { ruta: 'nuevo', componente: ProductCreateComponent },
      { ruta: ':id', componente: ProductDetailComponent },
    ],
  },
  { ruta: 'carrito', componente: CartComponent },
  { ruta: '^', redirección a: 'productos', coincidencia de ruta: 'completo' },
  { ruta: '**', redirección a: 'productos' }
];
```

En el fragmento anterior, utilizamos la propiedad `children` del objeto de definición de ruta para definir rutas secundarias que contienen una lista de objetos de definición de ruta.



Tenga en cuenta también que eliminamos la palabra `productos` de la propiedad de ruta de las rutas secundarias porque la ruta principal la agregará.

Una ruta principal también puede proporcionar servicios a sus hijas mediante la propiedad "proveedores" del objeto de definición de ruta. Proporcionar servicios en una ruta es muy útil cuando queremos limitar el acceso a un subconjunto de la configuración de enruteamiento. Si quisieramos restringir la clase `ProductsService` solo a los componentes relacionados con el producto, deberíamos hacer lo siguiente:

```
{
  ruta: 'productos',
  componente: ProductListComponent,
  niños: [
    { ruta: 'nuevo', componente: ProductCreateComponent },
```

```
{ ruta: ':id', componente: ProductDetailComponent },  
],  
proveedores: [ProductosServicio]  
}
```

Angular crea un inyector independiente al proporcionar servicios en objetos de definición de ruta, que es un elemento secundario inmediato del inyector raíz. Supongamos que el servicio también se proporciona en el inyector raíz y que el componente del carrito lo utiliza. En ese caso, la instancia creada por uno de los componentes relacionados con el producto será diferente a la del componente del carrito.

Hemos aprendido a usar el observable paramMap en el enrutamiento de Angular. En la siguiente sección, analizaremos un enfoque alternativo mediante instantáneas.

### Tomar una instantánea de los parámetros de la ruta

Al seleccionar un producto de la lista, el componente de lista de productos se elimina del árbol DOM y se añade el componente de detalles del producto. Para seleccionar otro producto, debemos hacer clic en el enlace "Productos" o en el botón "Atrás" del navegador. En consecuencia, el componente de detalles del producto se reemplaza por el componente de lista de productos en el DOM. Por lo tanto, solo se muestra un componente en pantalla a la vez.

Cuando se destruye el componente de detalles del producto, también se destruye su método ngOnInit y la suscripción al observable paramMap . Por lo tanto, no nos beneficiamos del uso de observables en este momento. Como alternativa, podríamos usar la propiedad instantánea del servicio ActivatedRoute para obtener los valores de los parámetros de ruta, como se indica a continuación:

```
ngOnInit(): vacío {  
  const id = this.route.snapshot.params['id'];  
  este.producto$ = este.productoServicio.getProduct(id);  
}
```

La propiedad instantánea representa el valor actual de un parámetro de ruta, que también es el valor inicial. Contiene la propiedad params , un objeto de pares clave-valor de parámetros de ruta a los que podemos acceder.



Si está seguro de que su componente no se reutilizará, utilice el método de instantánea.

Hasta ahora, hemos trabajado con parámetros de enruteamiento en forma de products/:id. Usamos estos parámetros para navegar a un componente que requiere dicho parámetro. En nuestro caso, el componente de detalles del producto requiere el parámetro id para obtener detalles específicos del producto. Sin embargo, existe otro tipo de parámetro de ruta que necesitamos que sea opcional, como veremos en la siguiente sección.

## Filtrado de datos mediante parámetros de consulta

En el Capítulo 8, Comunicación con Servicios de Datos a través de HTTP, aprendimos a pasar parámetros de consulta a una solicitud mediante la clase HttpParams . El enruteador Angular también permite pasar parámetros de consulta a través de la URL de la aplicación.

El método getProducts en el archivo products.service.ts usa parámetros de consulta HTTP para limitar los resultados de productos devueltos desde la API de Fake Store:

```
obtenerProductos(): Observable<Producto[]> {
  si (este.producto.longitud === 0) {
    const opciones = new HttpParams().set('limit', 10);
    devuelve este.http.get<Product[]>(este.productsUrl, {
      parámetros: opciones
    }).pipe(map(productos => {
      este.productos = productos;
      devolver productos;
    }));
  }
  retorno de(este.productos);
}
```

Utiliza un valor codificado para establecer el parámetro de consulta de límite . Modificaremos la aplicación para que el componente de lista de productos pase el valor de límite dinámicamente:

1. Abra el archivo products.service.ts y modifique el método getProducts para que

El límite se pasa como parámetro:

```
obtenerProductos(límite?: número): Observable<Producto[]> {
  si (este.producto.longitud === 0) {
    const opciones = new HttpParams().set('limit', limit || 10);
    devuelve este.http.get<Product[]>(este.productsUrl, {
      parámetros: opciones
    }).pipe(map(productos => {
```

```
        este.productos = productos;
        devolver productos;
    });
}
retorno de(este.productos);
}
```

En el método anterior, si el valor límite es falso, pasamos un valor predeterminado de 10 al parámetro de consulta.



Un valor falso se evalúa como Falso en un contexto booleano y puede ser nulo, indefinido, 0 o Falso. Puede leer más en <https://developer.mozilla.org/docs/Glosario/Falsy>.

2. Abra el archivo product-list.component.ts e importe el servicio ActivatedRoute y El operador switchMap RxJS:

```
importar { RouterLink, ActivatedRoute } desde '@angular/router';
importar { Observable, switchMap } de 'rxjs';
```

3. Inyecte el servicio ActivatedRoute en el constructor de la clase ProductListComponent :

```
constructor( productoServicio privado : ProductosServicio, ruta privada:
Ruta activada) {}
```

4. El servicio ActivatedRoute contiene un observable queryParamMap al que podemos suscribirnos para obtener los valores de los parámetros de consulta. Devuelve un objeto ParamMap , similar al observable paramMap que vimos anteriormente, al que podemos consultar para obtener los valores de los parámetros. Modifique getProducts. Método para utilizar el observable queryParamMap :

```
privado obtenerProductos() {
    este.productos$ = esta.ruta.queryParamMap.pipe(
        switchMap(parámetros => {
            devuelve este.productService.getProducts(Number(params.
obtener('limite')));
        })
    );
}
```

En el fragmento anterior, usamos el operador switchMap RxJS para canalizar el parámetro de límite desde el observable queryParamMap al método getProducts de ProductsService. clase como un número.

## 5. Ejecute el comando ng serve para iniciar la aplicación y navegue a http://

localhost:4200?limit=5. Debería ver una lista de 5 productos:

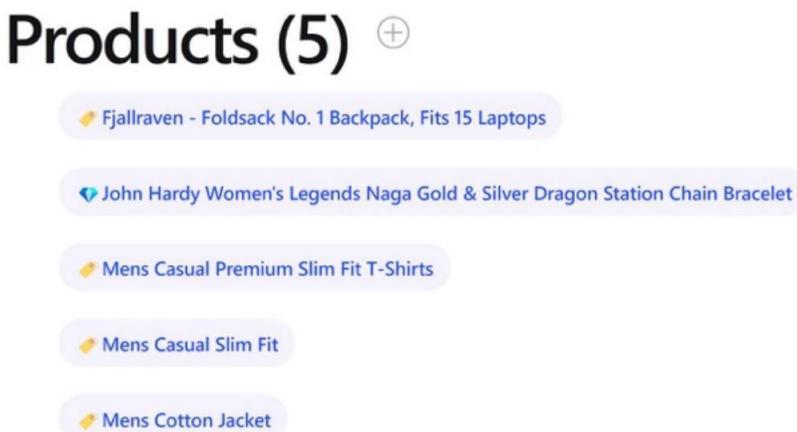


Figura 9.7: Lista de productos filtrada

Intente experimentar con diferentes valores para el parámetro límite y observe el resultado.

Los parámetros de consulta en el enruteamiento son potentes y pueden utilizarse para diversos casos de uso, como el filtrado y la ordenación de datos. También se pueden usar al trabajar con enruteamiento basado en instantáneas.

En la siguiente sección, exploraremos una nueva forma innovadora de pasar parámetros de ruta utilizando propiedades de entrada del componente.

## Vinculación de propiedades de entrada a rutas

Ya aprendimos, en el Capítulo 3, "Estructura de Interfaces de Usuario con Componentes", que utilizamos enlaces de entrada y salida para la comunicación entre componentes. Un enlace de entrada también puede pasar parámetros de ruta al navegar a un componente. Veremos un ejemplo con el componente de detalle del producto:

1. La vinculación de entrada con parámetros de ruta no está habilitada de forma predeterminada en el enruteador Angular.

Debemos activarlo desde el archivo de configuración de la aplicación. Abra el archivo app.config.ts e importe la función withComponentInputBinding del paquete npm @angular/router :

```
importar { provideRouter, withComponentInputBinding } desde '@angular/  
enrutador';
```

2. Pase la función anterior como segundo parámetro en el método provideRouter :

```
provideRouter(rutas, conComponentInputBinding()),
```

3. Ahora, abra el archivo product-detail.component.ts y cambie el tipo de id propiedad componente de una cadena:

```
id = entrada<cadena>();
```

Debemos cambiar el tipo de propiedad porque los parámetros de enrutamiento se pasan como cadenas.

4. Modifique el método ngOnInit para usar el parámetro id para obtener un producto:

```
ngOnInit(): vacío {  
    este.producto$ = este.productoServicio.obtenerProducto(Número(este.  
    identificación()));  
}
```

5. Ejecute el comando ng serve y verifique que los detalles del producto se muestren al seleccionar.

Seleccionar un producto de la lista.

Vincular parámetros de ruta a las propiedades de entrada del componente tiene las siguientes ventajas:

- La clase de componente TypeScript es más simple porque no tenemos llamadas asíncronas con observables
- Podemos acceder a componentes existentes que funcionan con enlaces de entrada y salida usando una ruta



La vinculación de entrada funciona con componentes que se activan mediante enrutamiento. Si queremos acceder a cualquier parámetro de ruta desde otro componente, debemos usar ActivatedRoute servicio.

Ahora que hemos aprendido todas las diferentes formas de pasar parámetros durante la navegación, hemos cubierto toda la información esencial que necesitamos para comenzar a construir aplicaciones Angular con enrutamiento. En las siguientes secciones, nos centraremos en prácticas avanzadas que mejoran la experiencia del usuario al utilizar la navegación en la aplicación en aplicaciones Angular.

## Mejorar la navegación con funciones avanzadas

Hasta ahora, hemos cubierto el enrutamiento básico con parámetros de ruta y consulta. Sin embargo, el enrutador Angular es bastante capaz y permite mucho más, como por ejemplo:

- Controlar el acceso a una ruta
- Evitar la navegación fuera de una ruta
- Precarga de datos para mejorar la experiencia de usuario de la aplicación
- Carga diferida de rutas para acelerar el tiempo de respuesta

En los siguientes apartados aprenderemos todas estas técnicas con más detalle.

### Controlar el acceso a las rutas

Cuando queremos controlar el acceso a una ruta específica, usamos un guard. Para crearlo, usamos el comando ng generate de la CLI de Angular, pasando la palabra guard y su nombre como parámetros:

```
ng genera autenticación de guardia
```

Al ejecutar el comando anterior, la CLI de Angular pregunta qué tipo de protección queremos crear. Existen varios tipos de protección que podemos crear según su funcionalidad:

- CanActivate: controla si se puede activar una ruta
- CanActivateChild: controla si se pueden activar las rutas secundarias
- CanDeactivate: controla si se puede desactivar una ruta



La desactivación ocurre cuando nos desviamos de una ruta.

- CanMatch: controla si se puede acceder a una ruta.

Seleccione CanActivate y presione Enter. La CLI de Angular crea el siguiente archivo auth.guard.ts :

```
importar { CanActivateFn } desde '@angular/router';

exportar const authGuard: CanActivateFn = (ruta, estado) => {
  devuelve verdadero;
};
```

El protector que hemos creado es una función de tipo CanActivateFn, que acepta dos parámetros:

- ruta: Indica la ruta que se activará
- estado: contiene el estado del enrutador tras una navegación exitosa



La función CanActivateFn puede devolver un valor booleano, ya sea de forma síncrona o asíncrona. En este último caso, el enrutador esperará a que se resuelva el observable o la promesa antes de continuar. Si el evento asíncrono no se completa, la navegación no continuará. También puede devolver un objeto UrlTree , que generará una nueva navegación a una ruta definida.

Nuestro guardián devuelve "true" inmediatamente, lo que permite el libre acceso a la ruta. Agreguemos lógica personalizada para controlar el acceso según si el usuario ha iniciado sesión:

1. Modifique las declaraciones de importación de la siguiente manera:

```
importar { inyectar } desde '@angular/core';
importar { CanActivateFn, Router } desde '@angular/router';
importar { AuthService } desde './auth.service';
```

2. Reemplace el cuerpo de la función de flecha con el siguiente fragmento:

```
const authService = inyectar(AuthService);
const router = inyectar(Enrutador);

si (authService.isLoggedIn()) {
    devuelve verdadero;
}
devolver router.parseUrl('/');
```

En el fragmento anterior, usamos el método de inyección para inyectar AuthService y Router servicios en la función. Luego, verificamos el valor de la señal isLoggedIn . Si es verdadero, permitimos que la aplicación navegue a la ruta solicitada. De lo contrario, usamos parseUrl. Método del servicio Router para navegar a la ruta raíz de la aplicación Angular.



El método parseUrl devuelve un objeto UrlTree , que cancela la navegación anterior y redirige al usuario a la URL introducida en el parámetro. Se recomienda usarlo en lugar del método browse , ya que podría generar un comportamiento inesperado y problemas de navegación complejos.

3. Abra el archivo app.routes.ts y agregue la siguiente declaración de importación :

```
importar { authGuard } desde './auth.guard';
```

4. Agregue la función authGuard en la matriz canActivate de la ruta del carrito :

```
{
  ruta: 'carrito',
  componente: CartComponent,
  canActivate: [authGuard]
}
```



La propiedad canActivate es un array porque varios guardias pueden controlar la activación de la ruta. El orden de los guardias en el array es importante. Si uno de los guardias no cumple con los requisitos, Angular impedirá el acceso a la ruta.

Ahora solo los usuarios autenticados pueden acceder al carrito de compras. Si ejecuta la aplicación con el comando ng serve y hace clic en el enlace "Mi carrito" , notará que no ocurre nada.



Al intentar acceder al carrito de compras desde la lista de productos, siempre se permanece en la misma página. Esto se debe a que la redirección que se produce debido a la protección de autenticación no tiene ningún efecto cuando ya se está en la ruta redirigida.

Otro tipo de protección relacionada con la activación de una ruta es la protección CanDeactivate . En la siguiente sección, aprenderemos a usarla para evitar que un usuario abandone una ruta.

## Evitar la navegación fuera de una ruta

Una función de tipo CanDeactivateFn controla si una ruta puede desactivarse . Aprenderemos a usarla implementando una función que notifique al usuario sobre productos pendientes en el carrito cuando abandone el componente del carrito:

1. Ejecute el siguiente comando para generar un nuevo protector:

```
ng generate guard checkout
```

2. Seleccione el tipo CanDeactivate de la lista y presione Entrar.

3. Abra el archivo checkout.guard.ts y agregue la siguiente declaración de importación :

```
importar { CartComponent } desde './cart/cart.component';
```

4. Cambie el genérico de CanDeactivateFn a CartComponent y elimine los parámetros de la función de flecha.



En un escenario del mundo real, probablemente necesitaremos agregar más componentes en los genéricos para crear una protección genérica.

5. Reemplace el cuerpo de la función de flecha con el siguiente fragmento:

```
const confirmación = confirmar(  
    Tienes artículos pendientes en tu carrito. ¿Quieres continuar?  
);  
confirmación de devolución ;
```

En el fragmento anterior, usamos el método de confirmación del objeto de ventana global para mostrar un cuadro de diálogo de confirmación antes de abandonar el componente del carrito. La aplicación esperará hasta que el cuadro de diálogo de confirmación se descarte como interacción del usuario.

6. Abra el archivo app.routes.ts y agregue la siguiente declaración de importación :

```
importar { checkoutGuard } desde './checkout.guard';
```

7. Un objeto de definición de ruta contiene una matriz canDeactivate similar a canActivate. Agregue el Función checkoutGuard para la matriz canDeactivate de la ruta del carrito :

```
{  
    ruta: 'carrito',  
    componente: CartComponent,  
    puedeActivar: [authGuard],  
    canDeactivate: [checkoutGuard]  
}
```



La propiedad canDeactivate es un array porque varios guardias pueden controlar la desactivación de la ruta. El orden de los guardias en el array es importante. Si uno de los guardias no cumple con los requisitos, Angular impedirá que el usuario abandone la ruta.

Para un escenario tan simple, podríamos haber escrito la lógica de la función checkoutGuard en línea para evitar la creación del archivo checkout.guard.ts :

```
{
```

```

    ruta: 'carrito',
    componente: CartComponent,
    puedeActivar: [authGuard],
    canDeactivate: [() => confirm('Tienes artículos pendientes en tu carrito. Hazlo
    ¿Quieres continuar?')]
}

```

Ejecute la aplicación con el comando ng serve y haga clic en el enlace "Mi carrito" después de iniciar sesión . Si hace clic en el enlace "Productos" o presiona el botón "Atrás" del navegador, debería ver un cuadro de diálogo con el siguiente mensaje:

Tienes artículos pendientes en tu carrito ¿Quieres continuar?

Si hace clic en el botón Cancelar , se cancela la navegación y la aplicación permanece en su estado actual. Si hace clic en el botón Aceptar , se le redirigirá a la lista de productos.

## Precarga de datos de ruta

Quizás hayas notado que, al acceder a la ruta raíz de la aplicación por primera vez, se produce un retraso en la visualización de la lista de productos. Esto es lógico, ya que estamos realizando una solicitud HTTP a la API del backend. Sin embargo, el componente de lista de productos ya estaba inicializado en ese momento.

El comportamiento anterior puede generar efectos no deseados si el componente contiene lógica que interactúa con los datos durante la inicialización. Para solucionar este problema, podemos usar un solucionador para precargar la lista de productos y cargar el componente cuando los datos estén disponibles.



Un solucionador puede ser útil para gestionar posibles errores antes de activar una ruta. Sería más apropiado navegar a una página de error si la solicitud a la API no se realiza correctamente en lugar de mostrar una página en blanco.

Para crear un resolver, utilizamos el comando ng generate de la CLI de Angular, pasando la palabra resolver y su nombre como parámetros:

```
ng genera productos de resolución
```

El comando anterior crea el siguiente archivo products.resolver.ts :

```

importar { ResolveFn } desde '@angular/router';

exportar const productsResolver: ResolveFn<boolean> = (ruta, estado) => {
  devuelve verdadero;
};

```

El resolver que creamos es una función de tipo ResolveFn, que acepta dos parámetros:

- ruta: Indica la ruta que se activará
- estado: Contiene el estado de la ruta activada



Una función ResolveFn puede devolver un observable o una promesa. El enrutador esperará a que el observable o la promesa se resuelvan antes de continuar. Si el evento asíncrono no se completa, la navegación no continuará.

Actualmente, nuestro solucionador devuelve un valor booleano. Agreguemos lógica personalizada para que devuelva una matriz de productos:

1. Agregue las siguientes declaraciones de importación :

```
importar { inyectar } desde '@angular/core';
importar { Producto } de './producto'; importar
{ ProductosServicio } de './productos.servicio';
```

2. Modifique la función productsResolver para que devuelva una matriz de productos:

```
exportar const productsResolver: ResolveFn<Product[]> = (ruta, estado) => {
};
```

3. Utilice el método de inyección para inyectar ProductsService en el cuerpo de la función:

```
const productoServicio = inyectar(ProductosServicio);
```

4. Utilice la propiedad queryParamMap para obtener el valor del parámetro límite de la ruta actual:

```
const limit = Number(route.queryParamMap.get('limit'));
```

5. Reemplace la declaración de retorno con lo siguiente:

```
devolver productoServicio.getProducts(limit);
```

6. La función resultante debería verse así:

```
exportar const productsResolver: ResolveFn<Product[]> = (ruta, estado) => {
  const productService = inject(ProductsService); const limit =
  Number(route.queryParamMap.get('limit'));
```

```
    devolver productoServicio.getProducts(limit); };
```

Ahora que hemos creado el solucionador, podemos conectarlo con el componente de lista de productos:

1. Abra el archivo app.routes.ts y agregue la siguiente declaración de importación :

```
importar { productsResolver } desde './products.resolver';
```

2. Agregue la siguiente propiedad de resolución a la ruta de productos :

```
{
  ruta: 'productos',
  componente: ProductListComponent,
  resolución:
    { productos: productsResolver
    }
}
```

La propiedad resolve es un objeto que contiene un nombre único como clave y la función resolvadora como valor. El nombre de la clave es importante porque lo usaremos en nuestros componentes para acceder a los datos resueltos.

3. Abra el archivo product-list.component.ts e importe el operador of desde rxjs

paquete npm:

```
importar { Observable, switchMap, de } de 'rxjs';
```

4. Modifique el método getProducts para que se suscriba a la propiedad de datos del Servicio ActivatedRoute :

```
privado getProducts()
{
  este.productos$ = este.ruta.datos.pipe(
    switchMap(datos => of(datos['productos']));
}
```

En el fragmento anterior, el observable "data" emite un objeto cuyo valor existe en la clave "products" . Observe que usamos el operador "switchMap" para devolver "products" en un nuevo observable.



En este punto, también podemos eliminar cualquier referencia al Servicio de Productos. clase porque ya no es necesaria.

- Ejecute el comando `ng serve` para iniciar la aplicación y verificar que la lista de productos se muestre al navegar a `http://localhost:4200`.

Los resolvers angulares mejoran el rendimiento de la aplicación cuando existe una lógica de inicialización compleja en los componentes enrutados. Otra forma de mejorar el rendimiento de la aplicación es cargar componentes o rutas secundarias bajo demanda, como veremos en la siguiente sección.

## Carga diferida de partes de la aplicación

Nuestra aplicación puede crecer en algún momento y la cantidad de datos que introducimos en ella también puede aumentar. La aplicación puede tardar bastante tiempo en iniciarse inicialmente, o ciertas partes pueden tardar mucho tiempo en cargarse. Para superar estos problemas, podemos utilizar una técnica llamada carga diferida.

La carga diferida implica que no cargamos inicialmente ciertas partes de la aplicación, como componentes o rutas de Angular. La carga diferida en una aplicación Angular ofrece muchas ventajas:

- Los componentes y rutas se pueden cargar a solicitud del usuario.
- Los usuarios que visitan determinadas áreas de su aplicación pueden beneficiarse significativamente de esta tecnología.
- Podemos agregar más funciones en un área de carga diferida sin afectar la aplicación general

tamaño del paquete

Para comprender cómo funciona la carga diferida en Angular, crearemos un nuevo componente que muestre el perfil del usuario actual.



Una buena práctica es cargar de forma diferida partes de la aplicación que no se utilizan con frecuencia, como el perfil del usuario que ha iniciado sesión actualmente.

Empecemos:

- Ejecute el siguiente comando para crear un componente Angular:

```
ng genera el componente usuario
```

2. Cree un archivo llamado user.routes.ts en la carpeta src\app y agregue el siguiente contenido:

```
importar { UserComponent } desde './user/user.component';

exportación predeterminada [
  { ruta: "", componente: UserComponent }
];
```

En el fragmento anterior, configuramos la propiedad path con una cadena vacía para activar la ruta por defecto.

También usamos la palabra clave "default" para aprovechar la función de exportación predeterminada en la carga diferida.

3. Abra el archivo app.routes.ts y agregue la siguiente definición de ruta en la variable rutas :

```
{ ruta: 'usuario', loadChildren: () => import('./usuario.rutas') }
```

La propiedad loadChildren de un objeto de definición de ruta se utiliza para cargar de forma diferida rutas angulares.

Devuelve una función de flecha que utiliza una instrucción de importación dinámica para cargar el archivo de rutas de forma diferida. La función de importación acepta la ruta relativa del archivo de rutas que queremos importar.

4. Agregue un nuevo elemento de anclaje al elemento <header> del archivo app.component.html que

Enlaces a la ruta recién creada:

```
<div class="enlaces-de-menú">
  <a routerLink="/products" routerLinkActive="active"> Productos</a>
  <a routerLink="/cart" routerLinkActive="active"> Mi carrito</a>
  <a routerLink="/user" routerLinkActive="active">Mi perfil</a>
</div>
```

5. Ejecute el comando ng serve y observe el resultado en la ventana de la consola. Debería...

parecer similares a lo siguiente:

Archivos de fragmentos iniciales	Tamaño bruto
polyfills.js	Nombres
main.js	polyfills   47,22 kB     1,14
estilos.css	principal   estilos kB
Total inicial   131,07 kB	

Archivos de fragmentos diferidos | Nombres | Tamaño sin procesar

chunk-D3RURZVV.js | rutas de usuario | 1,26 kB |

Generación del paquete de aplicaciones completada. [1,234 segundos]

En la salida anterior, podemos ver que la CLI de Angular ha creado un archivo de fragmento diferido llamado user-routes además de los archivos de fragmentos iniciales de la aplicación.

6. Navegue con su navegador a <http://localhost:4200> y abra las herramientas para desarrolladores.

7. Haga clic en el enlace Mi perfil e inspeccione la pestaña Solicitudes de red :

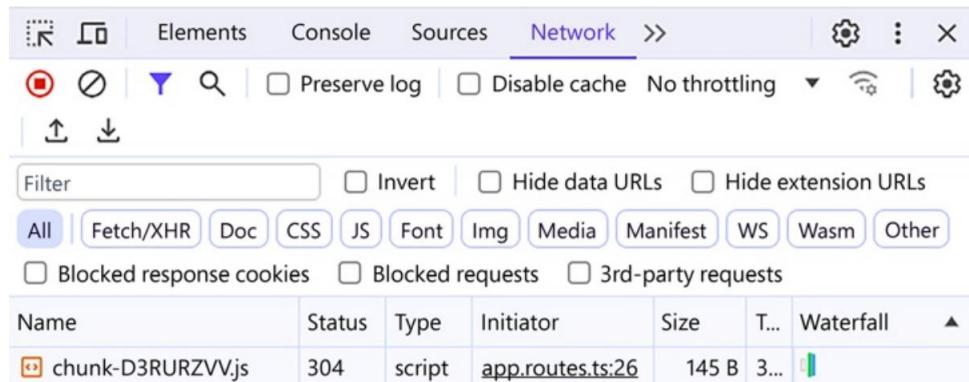


Figura 9.8: Ruta de carga diferida

La aplicación inicia una nueva solicitud al archivo de fragmento, que es el paquete de la ruta del usuario. El framework Angular crea un nuevo paquete para cada artefacto cargado diferidamente y no lo incluye en el paquete principal de la aplicación.

Si sale y vuelve a hacer clic en el enlace "Mi perfil" , observará que la aplicación no realiza una nueva solicitud para cargar el archivo del paquete. En cuanto se solicita una ruta con carga diferida, esta se guarda en memoria y puede utilizarse para solicitudes posteriores.

La carga diferida funciona no solo con rutas, sino también con componentes. Podríamos haber cargado diferidamente el componente de usuario en lugar de toda la ruta, modificando la ruta de usuario de la siguiente manera:

```
{
  ruta: 'usuario',
  cargarComponente: () => import('./usuario/usuario.componente').luego(c =>
  c.UserComponent),
}
```

En el fragmento anterior, usamos la propiedad `loadComponent` para importar `user.component.ts` Archivo dinámicamente. La función de importación devuelve una promesa que encadenamos con el método "then" para cargar la clase `UserComponent` .

La ruta de usuario está actualmente accesible para todos los usuarios, incluso si no están autenticados. En la siguiente sección, aprenderemos a protegerlos mediante guardias.

Protección de una ruta de carga diferida: Podemos

controlar el acceso no autorizado a una ruta de carga diferida de forma similar a como lo hacemos con las rutas normales.

Sin embargo, nuestros protectores deben ser compatibles con un tipo de función llamado CanMatchFn.

Ampliaremos nuestra protección de autenticación para su uso con rutas de carga diferida:

1. Abra el archivo auth.guard.ts e importe el tipo CanMatchFn desde @angular/router paquete npm:

```
importar { CanActivateFn, CanMatchFn, Router } desde '@angular/router';
```

2. Modifique la firma de la función authGuard de la siguiente manera:

```
exportar const authGuard: CanActivateFn | CanMatchFn = () => {
  const authService = inyectar(AuthService); const router
  = inyectar(Router);

  si (authService.isLoggedIn()) { devolver
    verdadero;

  } devolver router.parseUrl('/');
};
```

3. Abra el archivo app.routes.ts y agregue la función authGuard en la matriz canMatch de la ruta del usuario :

```
{
  ruta: 'usuario',
  loadChildren: () => import('./usuario.routes'),
  canMatch: [authGuard]
}
```



La propiedad canMatch es un array porque varios guardias pueden controlar la coincidencia de rutas . El orden de los guardias en el array es importante. Si uno de los guardias no coincide con una ruta, Angular impedirá el acceso a ella.

Si ahora ejecutamos la aplicación y hacemos clic en el enlace `Mi perfil`, notaremos que no podemos navegar al componente respectivo a menos que estemos autenticados.

La carga diferida es una técnica preferida cuando el rendimiento de la aplicación es crítico. Angular también ha introducido una función de mayor rendimiento para retrasar la carga de partes de una aplicación Angular, denominada vistas diferibles. Estas vistas ofrecen a los desarrolladores un control más preciso sobre las condiciones de carga de una parte de la aplicación. Analizaremos las vistas diferibles en el capítulo 15, Optimización del rendimiento de la aplicación.

## Resumen

Hemos descubierto el potencial del enrutador Angular y esperamos que hayan disfrutado de este recorrido por las complejidades de esta biblioteca. Una de las características más destacadas del enrutador Angular es la gran cantidad de opciones y escenarios que podemos cubrir con una implementación tan simple pero potente.

Hemos aprendido los conceptos básicos de la configuración del enrutamiento y la gestión de diferentes tipos de parámetros. También hemos aprendido funciones más avanzadas, como el enrutamiento secundario. Además, hemos aprendido a proteger nuestras rutas del acceso no autorizado. Finalmente, hemos mostrado todo el potencial del enrutamiento y cómo se puede mejorar el tiempo de respuesta con la carga diferida y la precarga.

En el próximo capítulo, reforzaremos los componentes de nuestra aplicación para mostrar los mecanismos subyacentes a los formularios web en Angular y las mejores estrategias para captar la entrada del usuario con controles de formulario.



# 10

## Recopilación de datos de usuario con formularios

Las aplicaciones web utilizan formularios para recopilar datos de entrada de los usuarios. Los casos de uso varían, desde permitir a los usuarios iniciar sesión, completar información de pago, reservar un vuelo o incluso realizar una búsqueda. Los datos del formulario pueden almacenarse posteriormente en el almacenamiento local o enviarse a un servidor mediante una API de backend.

En este capítulo cubriremos los siguientes temas sobre formularios:

- Introducción a los formularios web •

Creación de formularios basados en plantillas

- Construcción de formas reactivas
- Usar un generador de formularios
- Validación de entradas en formularios
- Manipular el estado del formulario

## Requisitos técnicos

Este capítulo contiene varios ejemplos de código que te guiarán en la creación y gestión de formularios en Angular. Puedes encontrar el código fuente relacionado en la carpeta ch10 del siguiente repositorio de GitHub:

<https://www.github.com/PacktPublishing/Aprendizaje-Angular-Quinta-Edición>

## Presentación de formularios web

Un formulario normalmente tiene las siguientes características que mejoran la experiencia del usuario de una aplicación web:

- Define diferentes tipos de campos de entrada

- Configura diferentes tipos de validaciones y muestra errores de validación al usuario.
- Admite diferentes estrategias para manejar datos si el formulario está en un estado de error

El framework Angular ofrece dos enfoques para gestionar formularios: basado en plantillas y reactivo .

Ninguno se considera mejor; debe elegir el que mejor se adapte a su situación.

La principal diferencia entre ambos enfoques es cómo gestionan los datos:

Formularios basados en plantillas: Son fáciles de configurar y añadir a una aplicación Angular. Operan únicamente en la plantilla del componente para crear elementos y configurar reglas de validación; por lo tanto, no son fáciles de probar. También dependen del mecanismo de detección de cambios del framework.

Formularios reactivos: Son más robustos al escalar y realizar pruebas. Operan en la clase del componente para gestionar los controles de entrada y establecer reglas de validación. También manipulan datos mediante un modelo de formulario intermedio, manteniendo su naturaleza immutable. Esta técnica es ideal si utilizas técnicas de programación reactiva con frecuencia o si tu aplicación Angular incluye varios formularios.

Un formulario en una aplicación web consta de un elemento HTML <form> que contiene elementos HTML para introducir datos, como los elementos <input> y <select> , y elementos <button> para interactuar con los datos. El formulario puede recuperar y guardar datos localmente o enviarlos a un servidor para su posterior manipulación. A continuación, se muestra un ejemplo de un formulario simple utilizado para iniciar sesión en una aplicación web:

```
<formulario>
  <div>
    <input type="text" name="nombre de usuario" placeholder="Nombre de usuario" />
  </div>
  <div>
    <input type="contraseña" name="contraseña" placeholder="Contraseña" />
  </div>
  Iniciar sesión
</form>
```

El formulario anterior tiene dos elementos <input> : uno para introducir el nombre de usuario y otro para la contraseña. El campo de contraseña está configurado como "password" para que el contenido del control de entrada no sea visible al escribir. El elemento <button> está configurado como " submit" para que el formulario pueda recopilar datos cuando el usuario haga clic en el botón o presione Intro en cualquier control de entrada.



Podríamos agregar otro botón con el tipo de reinicio si quisieramos restablecer los datos del formulario.

Tenga en cuenta que un elemento HTML debe residir dentro del elemento `<form>` para formar parte de él. La siguiente captura de pantalla muestra el aspecto del formulario al visualizarse en una página:

The screenshot shows a simple login form. It consists of three rectangular input fields stacked vertically. The top field is labeled "Username", the middle field is labeled "Password", and the bottom field is labeled "Login". Below the form is a caption.

Username

Password

Login

Figura 10.1: Formulario de inicio de sesión

Las aplicaciones web pueden mejorar significativamente la experiencia del usuario al utilizar formularios que brindan funciones como autocompletar en controles de entrada o solicitar al usuario que guarde datos confidenciales. Ahora que hemos entendido cómo se ve un formulario web, aprendamos cómo encaja todo eso en el marco Angular.

## Creación de formularios basados en plantillas

Los formularios basados en plantillas son una de las dos maneras de integrar formularios con Angular. Pueden ser muy útiles cuando queremos crear formularios pequeños y sencillos para nuestra aplicación Angular.

Aprendimos sobre el enlace de datos en el Capítulo 3, "Estructura de interfaces de usuario con componentes", y cómo podemos usar diferentes tipos para leer y escribir datos de un componente Angular. En ese caso, el enlace es unidireccional. En formularios basados en plantillas, podemos combinar ambos métodos y crear un enlace bidireccional que permite leer y escribir datos simultáneamente.

Los formularios basados en plantillas proporcionan la directiva `ngModel`, que podemos usar en nuestros componentes para obtener este comportamiento. Para obtener más información sobre los formularios basados en plantillas, adaptaremos la función de cambio de precio de nuestro componente de detalle de producto para que funcione con formularios Angular.



Necesitará el código fuente de la aplicación Angular que creamos en el Capítulo 9, Navegación por aplicaciones con enrutamiento, para seguir con el resto del capítulo.

Empecemos:

1. Abra el archivo product-detail.component.ts y agregue la siguiente declaración de importación :

```
importar { FormsModule } desde '@angular/forms';
```

Agregamos formularios basados en plantillas a una aplicación Angular usando la clase FormsModule del paquete npm @angular/forms .

2. Agregue FormsModule en la matriz de importaciones del decorador @Component :

```
@Component({
  selector: 'aplicación-producto-detalle',
  importaciones: [CommonModule, FormsModule],
  URL de plantilla: './producto-detalle.componente.html',
  styleUrls: ['./producto-detalle.componente.css'
})
```

3. Abra el archivo product-detail.component.html y modifique el elemento <input> de la siguiente manera:

mínimos:

```
<input placeholder="Nuevo precio" type="number" name="price"
[(ngModel)]="producto.precio" />
```

En el fragmento anterior, vinculamos la propiedad price de la variable de plantilla de producto a la directiva ngModel del elemento <input> . El atributo name es obligatorio para que Angular pueda crear internamente un control de formulario único que lo distinga.



La sintaxis de la directiva ngModel se conoce como un plátano en una caja, y la creamos en dos pasos. Primero, creamos el plátano rodeando ngModel. entre paréntesis (). Luego, lo encerramos en un recuadro entre corchetes [()].

4. Modifique el elemento <button> de la siguiente manera:

```
<button class="secondary" type="submit">Cambiar</button>
```

En el fragmento anterior, eliminamos el evento de clic del elemento <button> porque al enviar el formulario se actualiza el precio. También añadimos el tipo de envío para indicar que el formulario puede enviarse cuando el usuario hace clic en el botón.

5. Rodee los elementos <input> y <button> con el siguiente elemento <form> :

```
<form(ngSubmit)="cambiarPrecio(producto)">
  <input placeholder="Nuevo precio" type="number" name="price"
    [(ngModel)]="producto.precio" />
  <button class="secondary" type="submit">Cambiar</button>
</form>
```

En el fragmento anterior, vinculamos el método changePrice al evento ngSubmit del formulario. Esta vinculación activará la ejecución del método si presionamos Enter dentro del cuadro de entrada o hacemos clic en el botón. El evento ngSubmit forma parte del módulo FormsModule de Angular y se conecta al evento de envío nativo de un formulario HTML.

6. Abra el archivo product-detail.component.ts y modifique el método changePrice como sigue:

```
cambiarPrecio(producto: Producto) {
  este.productService.updateProduct(
    producto.id,
    precio del producto
  ).subscribe(() => this.router.navigate(['/productos']));
}
```

7. Ejecute la aplicación utilizando el comando ng serve y seleccione un producto de la lista.
8. Notarás que el precio actual del producto ya se muestra dentro del cuadro de entrada.

Intenta cambiar el precio y notarás que el precio actual del producto también cambia mientras escribes:

You selected: **SanDisk SSD PLUS 1TB Internal SSD - SATA III  
6 Gb/s**

€79.00

79

Change

electronics

Delete

Figura 10.2: Enlace bidireccional

El comportamiento de nuestra aplicación representado en la imagen anterior es la magia detrás del enlace bidireccional y ngModel.



El enlace bidireccional fue el mayor atractivo cuando AngularJS salió en 2010. Era complejo lograr ese comportamiento en aquellos días con JavaScript y jQuery.

Mientras escribimos dentro del cuadro de entrada, la directiva ngModel actualiza el valor del precio del producto.

El nuevo precio se refleja directamente en la plantilla porque utilizamos la sintaxis de interpolación Angular para mostrar su valor.

En nuestro caso, actualizar el precio actual del producto mientras se ingresa uno nuevo es una mala experiencia de usuario.

El usuario debe poder ver el precio actual del producto en todo momento. Modificaremos el componente de detalle del producto para que el precio se muestre correctamente:

1. Abra el archivo product-detail.component.ts y cree una propiedad de precio dentro del

Clase ProductDetailComponent :

```
precio: numero | undefined;
```

2. Modifique el método changePrice para utilizar la propiedad del componente de precio :

```
cambiarPrecio(producto: Producto) {
    este.productService.updateProduct(
        producto.id,
        ¡este.precio!
    ).subscribe(() => this.router.navigate(['/productos']));
}
```

3. Abra el archivo product-detail.component.html y reemplace el enlace en <input>

elemento para utilizar la nueva propiedad del componente:

```
<input placeholder="Nuevo precio" type="number" name="price"
[(ngModel)]="precio" />
```

Si ejecutamos la aplicación e intentamos introducir un nuevo precio en el cuadro de entrada "Nuevo precio" , observaremos que el precio actual mostrado no cambia. La función de cambiar el precio también funciona correctamente.

Hemos visto cómo los formularios basados en plantillas pueden ser útiles al crear formularios pequeños y simples.

En la siguiente sección, profundizamos en el enfoque alternativo que ofrece el marco Angular.

Trabajo: formas reactivas.

## Construyendo formas reactivas

Los formularios reactivos, como su nombre indica, proporcionan acceso reactivo a formularios web. Se crean con la reactividad en mente, donde los controles de entrada y sus valores se pueden manipular mediante flujos observables. Además, mantienen un estado inmutable de los datos del formulario, lo que facilita su prueba, ya que podemos estar seguros de que el estado del formulario se puede modificar de forma explícita y consistente.

Los formularios reactivos utilizan un enfoque programático para crear elementos de formulario y configurar reglas de validación mediante la configuración de todo en la clase del componente. Las clases clave de Angular involucradas en este enfoque son las siguientes:

- FormControl: representa un control de formulario individual, como un elemento <input> .
- FormGroup: Representa una colección de controles de formulario. El elemento <form> es el superior.  
FormGroup en la jerarquía de un formulario reactivo.

FormArray : Representa una colección de controles de formulario, al igual que FormGroup, pero se puede modificar en tiempo de ejecución. Por ejemplo, podemos añadir o eliminar objetos FormControl dinámicamente.  
según sea necesario.

Las clases anteriores están disponibles en el paquete npm @angular/forms y contienen propiedades que se pueden usar en los siguientes escenarios:

- Para representar la interfaz de usuario de forma diferente según el estado de un formulario o control
- Para comprobar si hemos interactuado con un formulario o control

Exploraremos cada clase de formulario mediante un ejemplo en nuestra aplicación Angular. En la siguiente sección, presentaremos formularios reactivos en nuestra aplicación mediante el componente de creación de producto.

## Interactuando con formas reactivas

La aplicación Angular que creamos contiene un componente para añadir nuevos productos. Este componente utiliza variables de referencia de plantilla para recopilar datos de entrada. Usaremos la API de formularios Angular para realizar la misma tarea con formularios reactivos:

1. Abra el archivo product-create.component.ts y agregue la siguiente declaración de importación :

```
importar { FormControl, FormGroup, ReactiveFormsModule } desde '@  
angular/forms';
```

2. Agregue la clase ReactiveFormsModule en la matriz de importaciones del decorador @Component :

```
@Componente({  
  selector: 'app-product-create',
```

```

importaciones: [ReactiveFormsModule],
URL de plantilla: './product-create.component.html',
styleUrl: './product-create.component.css'
})

```

La biblioteca de formularios Angular proporciona la clase ReactiveFormsModule para crear formularios reactivos en una aplicación Angular.

3. Defina la siguiente propiedad productForm en la clase ProductCreateComponent :

```

productForm = nuevo FormGroup({
  título: nuevo FormControl("", { nonNullable: true }),
  precio: nuevo FormControl<número | indefinido>(indefinido, { no nulo:
veradero }),
  categoría: nuevo FormControl("", { nonNullable: true })
});

```

El constructor FormGroup acepta un objeto que contiene pares clave-valor de controles de formulario . La clave es un nombre de control único y el valor es una instancia de FormControl . El constructor FormControl acepta el valor predeterminado del control en el primer parámetro. Para los controles de título y categoría , pasamos una cadena vacía para no establecer ningún valor inicialmente. Para el control de precio , que debe aceptar números como valores, lo establecemos inicialmente como indefinido. El segundo parámetro pasado en FormControl es un objeto que establece la propiedad nonNullable para indicar que el control no acepta valores nulos.

4. Tras crear el grupo de formularios y sus controles, debemos asociarlos con los elementos HTML correspondientes en la plantilla. Abra el componente product-create. archivo html y rodee los elementos HTML <input>, <select> y <button> con el siguiente elemento <form> :

```

<form [formGroup]="FormularioProducto">
  <div>
    <label for="title">Título</label>
    <input id="título" #título />
  </div>
  <div>
    <label for="price">Precio</label>
    <input id="precio" #precio tipo="número" />
  </div>
  <div>

```

```
<label for="category">Categoría</label> <select  
id="category" #category>  
    <option>Seleccionar una categoría</option>  
    <option value="electronics">Electrónica</option>  
    <option value="jewelery">Joyas</option> <option>Otros</  
        option>  
    </seleccionar>  
</div>  
<div>  
    <button (click)="createProduct(title.value, price.value,  
category.value)">Crear</button>  
</div>  
</form>
```

En la plantilla anterior, utilizamos la directiva `formGroup`, exportada desde la clase `ReactiveFormsModule`, para conectar una instancia de `FormGroup` a un elemento `<form>`.

5. La clase `ReactiveFormsModule` también exporta la directiva `formControlName`, que usamos para conectar una instancia de `FormControl` a un elemento HTML. Modificar el HTML del formulario. elementos como sigue:

```
<div>  
    <label for="title">Título</label>  
    <input id="título" formControlName="título" />  
</div>  
<div>  
    <label for="price">Precio</label> <input  
        id="price" formControlName="price" type="number" />  
</div>  
<div>  
    <label for="category">Categoría</label> <select  
        id="category" formControlName="category">  
        <option>Seleccionar una categoría</option>  
        <option value="electronics">Electrónica</option>  
        <option value="jewelery">Joyas</option> <option>Otros</  
            option>  
        </seleccionar>  
</div>
```

En el fragmento anterior, asignamos el valor de la directiva `formControlName` al nombre de la instancia de `FormControl` correspondiente. También eliminamos las variables de referencia de plantilla, ya que podemos obtener sus valores directamente de la instancia de `FormGroup`.

6. Modifique el método `createProduct` en el archivo `product-create.component.ts` según corresponda:

```
crearProducto() {
  este.productService.addProduct(este.productForm.value);
  suscribirse(() => {
    este.router.navigate(['/productos']);
  });
}
```

En el método anterior, utilizamos la propiedad de valor de la clase `FormGroup` para obtener el valor. valor del formulario.



Tenga en cuenta que la propiedad `value` no incluye valores de los campos deshabilitados de un formulario. En su lugar, podemos usar el método `getRawValue` para devolver valores. de todos los campos.

En este caso, podemos utilizar el valor del formulario porque el modelo del formulario es idéntico al Producto. interfaz.

Si fuera diferente, podríamos usar la propiedad de controles de la clase `FormGroup` para obtener los valores de los controles de formulario individualmente de la siguiente manera:

```
crearProducto() {
  este.productsService.addProduct({
    título: este.productForm.controla.título.valor,
    precio: este.productForm.controla.precio.valor,
    categoría: este.productForm.controla.categoría.valor
  }).subscribe(() => {
    este.router.navigate(['/productos']);
  });
}
```

La clase `FormControl` contiene una propiedad de valor que devuelve el valor de un control de formulario.

7. Modifique el elemento <form> en el archivo product-create.component.html para que creamos un nuevo producto al enviar el formulario:

```
<form [formGroup]="FormularioProducto" (ngSubmit)="crearProducto()">
  <div>
    <label for="title">Título</label>
    <input id="título" formControlName="título" />
  </div>
  <div>
    <label for="price">Precio</label>
    <input id="precio" formControlName="precio" type="número" />
  </div>
  <div>
    <label for="category">Categoría</label>
    <select id="categoría" formControlName="categoría">
      <option>Seleccionar una categoría</option>
      Electrónica

      <option>Otro</option>
    </seleccionar>
  </div>
  <div>
    <button type="submit">Crear</button>
  </div>
</form>
```

8. Abra el archivo global style.css y agregue el siguiente estilo CSS:

```
etiqueta {
  margen inferior: 4px;
  pantalla: bloque;
}
```

Queremos que los estilos anteriores estén disponibles globalmente porque los usaremos en el componente de carrito más adelante en el capítulo.

9. Abra el archivo product-create.component.css y elimine el estilo de la etiqueta <label> .

Si ejecutamos la aplicación, veremos que la funcionalidad de agregar un nuevo producto sigue funcionando como se esperaba.

Aprendimos que la clase FormGroup agrupa una colección de controles de formulario. Un control de formulario puede ser un solo control u otro grupo de formularios, como veremos en la siguiente sección.

### Creación de jerarquías de formularios anidados

El componente de creación de productos consta de un único grupo de formularios con tres controles. Algunos casos de uso en aplicaciones empresariales requieren formularios más avanzados que implican la creación de jerarquías anidadas de grupos de formularios. Considere el siguiente formulario, que se utiliza para agregar un nuevo producto junto con detalles adicionales:

## Add new product

Title

Price

Category

Select a category



## Additional details

Description

Photo URL

Create

Figura 10.3: Formulario de nuevo producto con información adicional

El formulario anterior puede parecer un solo grupo de formularios, pero si observamos más a fondo la clase de componente, veremos que productForm consta de dos instancias de FormGroup , una anidada dentro del otro:

```
productForm = nuevo FormGroup({
```

```
título: nuevo FormControl("", { nonNullable: true }), precio:  
nuevo FormControl<número | indefinido>(undefined, { nonNullable: true }), categoría:  
nuevo  
FormControl("", { nonNullable: true }), extra: nuevo  
FormGroup({ imagen:  
    nuevo FormControl("", descripción:  
        nuevo FormControl("") } );
```

La propiedad productForm es el grupo de formularios principal, mientras que extra es su grupo secundario. Un grupo de formularios principal puede tener tantos grupos secundarios como necesite. Si observamos la plantilla del componente, veremos que el grupo de formularios secundario se define de forma diferente al principal:

```
<form [formGroup]="FormularioProducto" (ngSubmit)="crearProducto()">  
  <div>  
    <label for="title">Título</label>  
    <input id="título" formControlName="título" />  
  </div>  
  <div>  
    <label for="price">Precio</label>  
    <input id="precio" formControlName="precio" type="número" />  
  </div>  
  <div>  
    <label for="category">Categoría</label>  
    <select id="categoría" formControlName="categoría">  
      <option>Seleccionar una categoría</option>  
      Electrónica  
  
      <option>Otro</option>  
    </seleccionar>  
  </div>  
  <h2>Detalles adicionales</h2>  
  <form formGroupName="extra">  
    <div>  
      <label for="descr">Descripción</label> <input  
        id="descr" formControlName="description" />  
    </div>
```

```

<div>
    <label for="photo">URL de la foto</label>
    <input id="foto" formControlName="imagen" />
</div>
</form>
<div>
    <button type="submit">Crear</button>
</div>
</form>

```

En la plantilla HTML anterior, utilizamos la directiva `formGroupName` para vincular el elemento de formulario interno a la propiedad adicional .



Quizás esperaba vincularlo directamente a la propiedad `productForm.extra` , pero Angular es bastante inteligente porque entiende que `extra` es un grupo de formularios hijo de `productForm`. Puede deducir esta información porque el elemento de formulario relacionado con `extra` está dentro del elemento de formulario que se vincula a la propiedad `productForm` .

El valor de un grupo de formularios secundarios se comparte con su formulario principal en una jerarquía de formularios anidada. En nuestro caso, el valor del grupo de formularios adicional se incluirá en el grupo `productForm` , manteniendo así un modelo de forma consistente.

Ya hemos visto las clases `FormGroup` y `FormControl` . En la siguiente sección, aprenderemos a usar la clase `FormArray` para interactuar con formularios dinámicos.

## Modificar formularios dinámicamente

Consideremos el escenario en el que hemos agregado algunos productos al carrito de compras de nuestra aplicación de tienda electrónica y queremos actualizar sus cantidades antes de realizar el pedido.

Actualmente, nuestra aplicación no tiene ninguna funcionalidad para un carrito de compras, por lo que ahora añade uno:

1. Ejecute el siguiente comando para crear una interfaz de carrito :

```
ng generar interfaz Carrito
```

2. Abra el archivo `cart.ts` y modifique la interfaz del carrito de la siguiente manera:

```
Interfaz de exportación Carrito {
```

```
    id: numero;
```

```
    productos: { productId :número }[];  
}
```

En el fragmento anterior, la propiedad de productos contendrá los ID de productos que pertenecen al carrito actual.

3. Cree un nuevo servicio para administrar el carrito de compras ejecutando la siguiente CLI de Angular dominio:

```
ng generar carrito de servicio
```

4. Abra el archivo cart.service.ts y modifique las declaraciones de importación de la siguiente manera:

```
importar { Inyectable, inyectar } desde '@angular/core'; importar  
{ HttpClient } desde '@angular/common/http'; importar { Observable,  
aplazar, mapear } desde 'rxjs'; importar { Carrito } desde './  
cart'; importar { APP_SETTINGS }  
desde './app.settings';
```

5. Cree las siguientes propiedades en la clase CartService :

```
carrito: Carrito | indefinido;  
private cartUrl = inject(APP_SETTINGS).apiUrl + '/carts';
```

La propiedad cartUrl se utiliza para el punto final del carrito de la API de Fake Store y la propiedad cart para mantener un caché local del carrito del usuario.

6. Inyecte el servicio HttpClient en el constructor:

```
constructor(privado http: HttpClient) { }
```

7. Agregue el siguiente método para agregar un producto al carrito:

```
addProduct(id: número): Observable<Cart> {  
  const cartProduct = { productId: id, cantidad: 1 };  
  
  devolver aplazar(() =>  
    este.carrito  
    ? este.http.post<Cart>(este.cartUrl, { productos: [cartProduct]  
  })  
    : este.http.put<Cart>(`${este.cartUrl}/${este.cart.id}`, {  
      productos:  
      [ ...este.carrito.productos,
```

```

    carritoProducto
  ]
})
).pipe(map(cart => this.cart = cart));
}

```

En el método anterior, usamos un nuevo operador RxJS llamado defer. Este operador funciona como una sentencia if/else para observables.

Si la propiedad del carrito no se ha inicializado, lo que significa que nuestro carrito está actualmente vacío, iniciamos una solicitud POST a la API pasando la variable cartProduct como parámetro.

De lo contrario, iniciamos una solicitud PATCH pasando el cartProduct junto con los productos existentes del carrito.

Hemos completado la configuración de nuestro servicio para que pueda comunicarse con la API de Fake Store. Ahora necesitamos conectar el servicio con el componente respectivo:

1. Abra el archivo product-detail.component.ts y agregue la siguiente declaración de importación :

```
importar { CartService } desde './cart.service';
```

2. Inyecte CartService en la clase ProductDetailComponent :

```

constructor(
  productoServicio privado : ProductosServicio,
  servicio de autenticación público : AuthService,
  ruta privada : ActivatedRoute,
  enrutador privado : enrutador,
  servicio de carrito privado: CartService
) {}

```

3. Modifique el método addToCart para que llame al método addProduct de CartService clase:

```

addToCart(id: número) {
  este.cartService.addProduct(id).subscribe();
}

```

4. Finalmente, abra el archivo product-detail.component.html y modifique el evento de clic de El botón Añadir al carrito :

```
Añadir al carrito
```

Hemos implementado la funcionalidad básica para almacenar los productos seleccionados que los usuarios desean comprar. Ahora, debemos modificar el componente del carrito para mostrar los artículos del carrito:

1. Abra el archivo cart.component.ts y modifique las declaraciones de importación de la siguiente manera:

```
importar { Componente, OnInit } desde '@angular/core'; importar {  
      
    FormArray,  
    Control de formulario,  
    Grupo de formularios,  
    Módulo de formularios reactivos  
} de '@angular/forms'; importar  
{ Producto } de './producto';  
importar { CartService } desde '../cart.service'; importar  
{ ProductsService } desde '../products.service';
```

2. Agregue la clase ReactiveFormsModule en la matriz de importaciones del decorador @Component :

```
@Componente({  
    selector: 'app-cart',  
    importaciones: [ReactiveFormsModule],  
    templateUrl: './cart.component.html',  
    estiloUrl: './cart.component.css' })
```

3. Agregue la interfaz OnInit a la lista de interfaces implementadas de la clase CartComponent :

La clase de exportación CartComponent implementa OnInit

4. Cree las siguientes propiedades en la clase TypeScript:

```
cartForm = nuevo FormGroup({  
    productos: new FormArray<FormControl<number>>([]));  
  
    productos: Producto[] = [];
```

En el fragmento anterior, creamos un objeto FormGroup que contiene una propiedad de productos . Establecemos el valor de la propiedad "products" en una instancia de la clase FormArray . El constructor de la clase FormArray acepta una lista de instancias de FormControl con el tipo number como parámetro. La lista está vacía por ahora, ya que el carrito no contiene productos. La propiedad "products" , fuera de la instancia FormGroup , se usará para fines de búsqueda para mostrar el título de cada producto del carrito.

5. Agregue un constructor para injectar los siguientes servicios:

```
constructor(  
    servicio de carrito privado : CartService,  
    servicio de productos privado : ProductsService ) {}
```

6. Crea el siguiente método para obtener productos del carrito:

```
privado obtenerProductos() {  
    este.productsService.getProducts().subscribe(productos =>  
        { este.cartService.cart?.products.forEach(artículo => {  
            const producto = productos.find(p => p.id === item.productId); if (producto) {  
  
                este.productos.push(producto);  
  
            } });});  
}
```

En el método anterior, nos suscribimos inicialmente al método getProducts de la clase ProductsService para obtener los productos disponibles. Luego, para cada producto del carrito, extraemos la propiedad productId y comprobamos si existe. Si se encuentra el producto, lo añadimos a la propiedad del componente products .

7. Crea otro método para construir nuestro formulario:

```
privado buildForm() {  
    este.productos.paraCada(() => {  
        este.cartForm.controla.productos.push(  
            nuevo FormControl(1, { no nulo: verdadero })  
  
        );});  
}
```

En el método anterior, iteramos sobre la propiedad products y añadimos una instancia de FormControl para cada una dentro de la matriz de formularios products . Establecemos el valor de cada control de formulario en 1 para indicar que el carrito contiene un artículo de cada producto por defecto.

8. Cree el siguiente método ngOnInit que combina ambos métodos de los pasos 6 y 7:

```
ngOnInit(): vacío {  
  esto.getProducts();  
  este.buildForm();  
}
```

9. Abra el archivo cart.component.html y reemplace su plantilla HTML con la siguiente contenido:

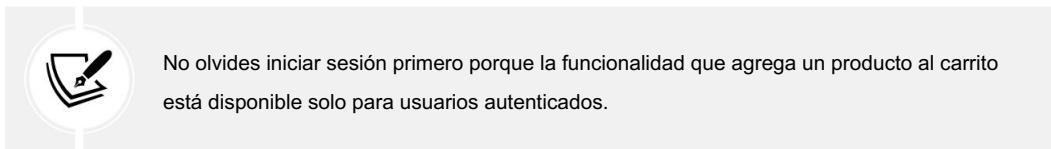
```
<div [formGroup]="Formulario de carrito">  
  <div formArrayName="productos">  
    @for(producto de cartForm.controls.products.controls; pista $index) {  
  
      <label>{{productos[$índice].título}}</label>  
      <input [formControlName]="$índice" tipo="número" />  
    }  
  </div>  
</div>
```

En la plantilla anterior, usamos un bloque @for para iterar sobre la propiedad de controles del array de formularios de productos y crear un elemento <input> para cada uno. Usamos el \$index. La palabra clave del bloque @for asigna un nombre dinámico a cada control de formulario mediante la vinculación formControlName . También hemos añadido una etiqueta <label> que muestra el título del producto desde la propiedad del componente products . El título del producto se obtiene mediante el \$index. del producto actual en la matriz.

10. Finalmente, abra los archivos cart.component.css y agregue los siguientes estilos CSS:

```
:anfitrión {  
  ancho: 500px;  
}  
  
aporte {  
  ancho: 50px;  
}
```

Para ver el componente del carrito en acción, ejecute la aplicación usando el comando ng serve y agregue algunos productos al carrito.



Después de añadir algunos productos al carrito, haga clic en el enlace "Mi carrito" para ver su carrito de compras. Debería verse así:

SanDisk SSD PLUS 1TB Internal SSD - SATA III 6 Gb/s

1

Mens Cotton Jacket

1

WD 2TB Elements Portable External Hard Drive - USB 3.0

1

Figura 10.4: Carrito de compras

Una vez que hemos establecido la lógica empresarial para administrar un carrito de compras, también podemos actualizar la protección de pago que creamos en el capítulo anterior:

1. Abra el archivo checkout.guard.ts y agregue las siguientes declaraciones de importación :

```
importar { inyectar } desde '@angular/core';
importar { CartService } desde './cart.service';
```

2. Inyecte la clase CartService en la función checkoutGuard usando la siguiente declaración:

```
const cartService = inyectar(CartService);
```

3. Modifique el cuerpo restante de la función de flecha de checkoutGuard para que mostremos el

Diálogo de confirmación solo cuando el carrito no está vacío:

```
si (cartService.cart) {
  const confirmación = confirmar(
    Tienes artículos pendientes en tu carrito. ¿Quieres continuar?
  );
  confirmación de devolución ;
}
devuelve verdadero;
```

Con FormArray, hemos completado nuestra exploración de los componentes básicos de un formulario Angular. Aprendimos a usar las clases de formularios Angular para crear formularios web estructurados y recopilar la información del usuario. En la siguiente sección, aprenderemos a crear formularios Angular usando Servicio FormBuilder .

### Usando un generador de formularios

Usar clases de formulario para crear formularios Angular puede resultar repetitivo y tedioso en escenarios complejos . El framework Angular proporciona FormBuilder, un servicio integrado para formularios Angular que contiene métodos auxiliares para crear formularios. Veamos cómo podríamos usarlo para crear un formulario para crear nuevos productos:

1. Abra el archivo product-create.component.ts e importe OnInit y FormBuilder artefactos:

```
importar { Componente, OnInit } desde '@angular/core';
importar { FormControl, FormGroup, ReactiveFormsModule, FormBuilder } desde
'@angular/forms';
```

2. Agregue OnInit a la lista de interfaces implementadas en la clase ProductCreateComponent :

La clase de exportación `ProductCreateComponent` implementa OnInit

3. Inyecte la clase FormBuilder en el constructor:

```
constructor(
  Productos privadosServicio : ProductosServicio,
  enrutador privado : enrutador,
  constructor privado: FormBuilder
){}
```

4. Modifique la propiedad productForm de la siguiente manera:

```
FormularioProducto: FormGroup<
  título: FormControl<string>,
  precio: FormControl<número | indefinido>,
  categoría: FormControl<string>
  > | indefinido;
```

En el fragmento anterior, definimos solo la estructura del formulario porque ahora se creará utilizando el servicio FormBuilder .

5. Cree el siguiente método para crear el formulario:

```
privado buildForm() {  
    este.productForm = este.builder.nonNullable.group({  
        título: [""],  
        precio: this.builder.nonNullable.control<número |  
undefined>(undefined),  
        categoría: [""]  
    });  
}
```

En el método anterior, usamos la propiedad nonNullable de la clase FormBuilder para crear un grupo de formularios que no puede ser nulo. El método "group" se utiliza para agrupar controles de formulario. Los controles de formulario de título y categoría se crean con una cadena vacía como valor predeterminado. El control de formulario de precio sigue un enfoque diferente, ya que no podemos asignar un valor predeterminado indefinido debido a las limitaciones del lenguaje TypeScript. En este caso, utilizamos el método de control de la propiedad nonNullable para definir el control de formulario.

6. Agregue el gancho del ciclo de vida ngOnInit para ejecutar el método buildForm :

```
ngOnInit(): vacío {  
    este.buildForm();  
}
```

7. Agregue el operador de afirmación no nulo al acceder a la propiedad productForm en el

Método createProduct :

```
crearProducto() {  
    este.productService.addProduct(este.productForm!.value).  
suscribirse(() => {  
    este.router.navigate(['/productos']);  
});  
}
```

8. Abra el archivo product-create.component.html y agregue el operador de afirmación no nulo en el elemento HTML <form> también:

```
<form [formGroup]="FormularioProducto!" (ngSubmit)="crearProducto()">  
<div>  
    <label for="title">Título</label>
```

```
<input id="título" formControlName="título" />
</div>
<div>
    <label for="price">Precio</label>
    <input id="precio" formControlName="precio" type="número" />
</div>
<div>
    <label for="category">Categoría</label>
    <select id="categoría" formControlName="categoría">
        <option>Seleccionar una categoría</option>
        <option value="electronics">Electrónica</option>
        <option value="jewelery">Joyas</option>
        <option>Otro</option>
        <option></seleccionar>
    </div>
    <div>
        <button type="submit">Crear</button>
    </div>
</form>
```

Al usar el servicio FormBuilder para crear formularios Angular, no tenemos que lidiar con FormGroup y los tipos de datos FormControl explícitamente, aunque eso es lo que se está creando bajo el capó.

Ejecute la aplicación con el comando ng serve y verifique que el proceso de creación del nuevo producto funcione correctamente.

Intente hacer clic en el botón Crear sin introducir ningún valor en los controles del formulario y observe qué sucede en la lista de productos. La aplicación crea un producto con un título vacío. Esta es una situación que debemos evitar en situaciones reales. Debemos tener en cuenta...

estado de un control de formulario y tomar medidas en consecuencia.



El código de ejemplo en el resto del capítulo no utiliza el servicio FormBuilder cuando se trabaja con formularios reactivos.

En la siguiente sección, investigaremos diferentes propiedades que podemos verificar para obtener el estado del formulario y proporcionar comentarios al usuario.

## Validación de entradas en formularios

Un formulario Angular debe validar la entrada y proporcionar retroalimentación visual para mejorar la experiencia de usuario y guiar a los usuarios para que completen el formulario correctamente. Investigaremos las siguientes maneras de validar formularios en aplicaciones Angular:

- Validación global con CSS
- Validación en la clase de componente
- Validación en la plantilla del componente
- Creación de validadores personalizados

En la siguiente sección, aprenderemos cómo aplicar reglas de validación de forma global en una aplicación Angular utilizando estilos CSS.

## Validación global con CSS

El marco Angular establece las siguientes clases CSS automáticamente en un formulario, basado en plantillas o reactivo, que podemos usar para proporcionar comentarios de los usuarios:

- ng-untouched: Indica que aún no hemos interactuado con un formulario
- ng-touched: Indica que hemos interactuado con un formulario
- ng-dirty: Indica que hemos establecido un valor en un formulario
- ng-pristine: Indica que aún no hemos modificado ningún formulario

Además, Angular agrega las siguientes clases al elemento HTML de un control de formulario:

- ng-valid: Indica que el valor de un formulario es válido
- ng-invalid: Indica que el valor de un formulario no es válido

Angular asigna las clases CSS anteriores al formulario y sus controles según su estado. El estado del formulario se evalúa según el estado de sus controles. Por ejemplo, si al menos un control de formulario no es válido, Angular asignará la clase CSS ng-invalid al formulario y a su correspondiente... control.



En el caso de jerarquías de formularios anidados, el estado de un grupo de formularios secundarios se transmite a la jerarquía y se comparte con su formulario principal.

Podemos usar las clases CSS integradas y aplicar estilo a los formularios de Angular usando solo CSS. Por ejemplo, para mostrar un borde resaltado en azul claro en un control de entrada al interactuar con él por primera vez, debemos agregar el siguiente estilo:

```
input.ng-touched { border:  
    3px azul claro sólido;  
}
```

También podemos combinar clases CSS según las necesidades de nuestra aplicación:

1. Abra el archivo global style.css y modifique el estilo input.valid de la siguiente manera:

```
entrada.válida, entrada.ng-sucia.ng-válida {  
    border: verde sólido;  
}
```

El estilo anterior mostrará un borde verde cuando un control de entrada tenga un valor válido ingresado por el usuario.

2. Modifique el estilo input.invalid según corresponda:

```
input.invalid, input.ng-dirty.ng-invalid { border: rojo sólido;  
}
```

El estilo anterior mostrará un borde rojo cuando un control de entrada tenga un valor no válido ingresado por el usuario.

3. Abra el archivo product-create.component.html y agregue el atributo requerido en el

Controles del formulario <input> :

```
<div>  
    <label for="title">Título</label>  
    <input id="título" formControlName="título" requerido />  
</div>  
  
    <div>  
        <label for="price">Precio</label> <input  
            id="price" formControlName="price" type="number" required />  
    </div>
```

4. Ejecute la aplicación usando el comando `ng serve` y navegue a `http://localhost:4200/` productos/huevos.

5. Ingrese texto en el campo Título y haga clic fuera del control de entrada. Observe que tiene un borde verde.

6. Elimine el texto del campo Título y haga clic fuera del control de entrada. El borde Ahora debería volverse rojo.

Aprendimos a definir reglas de validación en la plantilla mediante estilos CSS. En la siguiente sección, aprenderemos a definirlas en formularios basados en plantillas y a proporcionar retroalimentación visual mediante mensajes adecuados.

## Validación en formularios basados en plantillas

En la sección anterior, aprendimos que Angular agrega una colección de clases CSS integradas al validar formularios Angular. Cada clase tiene una propiedad booleana correspondiente en el modelo de formulario correspondiente, tanto en formularios basados en plantillas como en formularios reactivos:

- `intacto`: Indica que aún no hemos interactuado con un formulario
- `tocado`: Indica que hemos interactuado con un formulario
- `sucio`: Indica que hemos establecido un valor en un formulario
- `pristino`: Indica que aún no hemos modificado ningún formulario
- `válido`: Indica que el valor de un formulario es válido
- `inválido`: Indica que el valor de un formulario no es válido

Podemos aprovechar las clases anteriores para informar al usuario sobre el estado actual del formulario. Primero, investiguemos el comportamiento del proceso de cambio de precio en el componente de detalles del producto:

1. Ejecute el comando `ng serve` para iniciar la aplicación y navegue a `http://localhost:4200`.
2. Seleccione un producto de la lista.
3. Agregue un valor de 0 en el cuadro de entrada Nuevo precio y haga clic en el botón Cambiar .
4. Seleccione el mismo producto de la lista y observe el resultado:

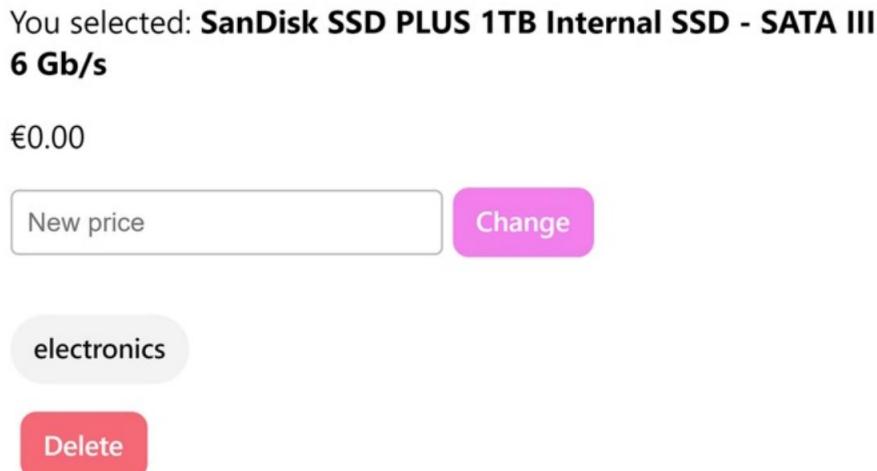
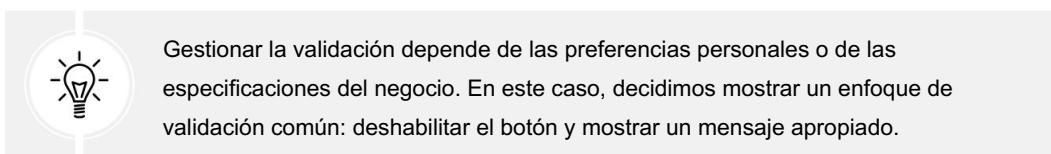


Figura 10.5: Detalles del producto

La lógica de presentación del componente no detecta que el usuario puede introducir 0 como precio del producto. Un producto siempre debe tener un precio.

El componente de detalles del producto debe validar la entrada del valor del precio y, si se determina que la entrada no es válida, deshabilitar el botón Cambiar y mostrar un mensaje informativo al usuario.



La validación basada en plantillas se realiza en la plantilla del componente. Abra el detalle del producto archivo component.html y ejecute los siguientes pasos:

1. Cree la variable de referencia de plantilla priceCtrl y vincúlela a la propiedad ngModel :

```
<entrada  
    placeholder="Nuevo precio"  
    tipo="número"  
    nombre="precio"  
    #priceCtrl="ngModel"  
    [(ngModel)]="precio" />
```

La propiedad ngModel nos da acceso al modelo de control de formulario subyacente.

2. Agregue los atributos de validación requeridos y mínimos al elemento HTML:

```
<entrada
placeholder="Nuevo precio"
tipo="número"
nombre="precio"
requerido min="1"
#priceCtrl="ngModel"
[(ngModel)]="precio" />
```

El atributo de validación mínima solo se puede utilizar con elementos HTML <input> del tipo número tipo. Se utiliza para definir el valor mínimo al utilizar las flechas del control numérico.

3. Agregue el siguiente elemento HTML <span> debajo del elemento <button> del formulario:

```
@if (priceCtrl.dirty && (priceCtrl.invalid || priceCtrl.
hasError('min'))) {
<span class="help-text">Por favor, introduzca un precio válido
}
```

El elemento HTML anterior se mostrará cuando ingresemos un valor de precio y luego lo dejemos en blanco o ingresemos un cero. Usamos el método hasError del modelo de control de formulario para... comprobar si la validación mínima genera un error.



Todos los atributos de validación se pueden comprobar mediante el método hasError . El estado de validez de un control se evalúa en función del estado de todas las validaciones. atributos que asignamos al elemento HTML.

4. Agregue una variable de referencia de plantilla priceForm en el elemento HTML <form> y vincúlela a la propiedad ngForm :

```
<form (ngSubmit)="cambiarPrecio(producto)" #priceForm="ngForm">
<entrada
placeholder="Nuevo precio"
tipo="número"
nombre="precio"
requerido min="1"
#priceCtrl="ngModel"
[(ngModel)]="precio" />
```

```
<button class="secondary" type="submit">Cambiar</button> @if
(priceCtrl.dirty && (priceCtrl.invalid || priceCtrl.
hasError('min'))) {
    <span class="help-text">Por favor, introduzca un precio válido
}
</form>
```

La propiedad ngForm nos da acceso al modelo de formulario subyacente.

#### 5. Vincula la propiedad deshabilitada del elemento HTML <button> al estado no válido del

modelo de formulario:

```
<botón
    clase="secundaria"
    tipo="enviar"
    [deshabilitado]="priceForm.invalid">
    Cambiar
</botón>
```



En la plantilla anterior, podríamos vincular directamente el estado priceCtrl.invalid , ya que el formulario solo tiene un control. Elegimos el formulario para fines de demostración.

#### 6. Abra el archivo style.css y agregue los siguientes estilos CSS para la etiqueta <span> y el

botón deshabilitado :

```
.help-text
{
    display: flex;
    color: var(--hot-red); tamaño
    de fuente: 0.875rem;

} botón:deshabilitado {
    color de fondo: gris claro; cursor: no
    permitido;
}
```

Para verificar que la validación funciona según lo previsto, ejecute los siguientes pasos:

1. Ejecute el comando ng serve para iniciar la aplicación y seleccione un producto de la lista.

2. Ingrese 0 en el cuadro de entrada Nuevo precio y observe el resultado:

You selected: **SanDisk SSD PLUS 1TB Internal SSD - SATA III 6 Gb/s**

€109.00

Change

Please enter a valid price

electronics

Delete

Figura 10.6: Error de validación

3. Ingrese un valor válido y verifique que el mensaje de error haya desaparecido y que el botón Cambiar esté activado.

4. Deje en blanco el cuadro de entrada Nuevo precio y verifique que el mensaje de error se muestre nuevamente.

Ahora que hemos aprendido cómo realizar la validación en formularios basados en plantillas, veamos cómo validar los datos de entrada en formularios reactivos.

## Validación en formas reactivas

Los formularios basados en plantillas se basan únicamente en la plantilla del componente para realizar validaciones. En los formularios reactivos, la fuente de información es nuestro modelo de formulario, que reside en la clase TypeScript del componente. Definimos las reglas de validación en los formularios reactivos al crear la instancia FormGroup mediante programación.

Para demostrar la validación en formularios reactivos, agregaremos reglas de validación en la creación del producto.

componente:

1. Abra el archivo product-create.component.ts e importe la clase Validators desde el

Paquete npm @angular/forms :

```
importar {  
    Control de formulario,  
    Grupo de formularios,  
    Módulo de formularios reactivos,  
    Validadores  
} de '@angular/forms';
```

2. Modifique la declaración de la propiedad productForm para que el título y el precio formen

Los controles pasan una propiedad de validadores en la instancia de FormControl :

```
productForm = nuevo FormGroup({  
    título: nuevo FormControl("", {  
        no nulo: verdadero,  
        validadores: Validadores.requeridos  
    }),  
    precio: nuevo FormControl<número | indefinido>(indefinido, {  
        no nulo: verdadero,  
        validadores: [Validadores.obligatorio, Validadores.min(1)]  
    }),  
    categoría: nuevo FormControl("", { nonNullable: true })  
});
```

La clase Validadores contiene un campo estático para cada regla de validación disponible. Contiene Prácticamente las mismas reglas de validación disponibles para formularios basados en plantillas. Podemos combinar varios validadores agregándolos a una matriz, como lo indican los validadores. propiedad en la forma de precio control.



Al agregar un validador mediante la clase FormControl , podemos eliminar el atributo HTML correspondiente de la plantilla HTML. Sin embargo, se recomienda conservarlo por motivos de accesibilidad para que las aplicaciones de lectura de pantalla... ciones pueden usarlo.

3. Abra el archivo product-create.component.html y use la propiedad no válida del

Propiedad productForm para deshabilitar el botón Crear :

```
<button type="submit" [disabled]="productForm.invalid">Crear</  
botón>
```

4. Agregue un elemento HTML <span> en cada control de formulario <input> para mostrar un mensaje de error cuando se ha tocado el control y la validación requerida arroja un error:

```
<div>
  <label for="title">Título</label>
  <input id="title" formControlName="title" required /> @if
    (productForm.controls.title.touched && productForm.controls.
    título.invalido) { <span
      class="help-text">El título es obligatorio
    }
</div>
<div>
  <label for="price">Precio</label>
  <input id="precio" formControlName="precio" type="número" requerido />

  @if (productForm.controls.price.touched && productForm.controls.
  precio.no válido) {
    <span class="help-text">Se requiere el precio
  }
</div>
```

En el fragmento anterior, utilizamos la propiedad de controles de la propiedad productForm para obtener acceso a los modelos de control de formulario individuales y obtener sus estados.

5. Sería útil mostrar diferentes mensajes según la regla de validación. Podríamos mostrar un mensaje más específico cuando la validación mínima del control de precio genere un error, por ejemplo. Podemos usar el método hasError que vimos en la sección anterior para mostrar dicho mensaje:

```
<div>
  <label for="price">Precio</label>
  <input id="precio" formControlName="precio" type="número" requerido />

  @if (productForm.controls.price.touched && productForm.controls.
  precio.hasError('requerido')) {
    <span class="help-text">Se requiere el precio
  }
  } @if (productForm.controls.price.touched && productForm.controls.price.hasError('min')) {
    <span class="help-text">El precio debe ser mayor que 0
```

```
}
```

```
</div>
```

El marco Angular proporciona un conjunto de validadores integrados que aprendimos a usar en nuestros formularios.

En la siguiente sección, aprenderemos cómo crear un validador personalizado para formularios reactivos y basados en plantillas para satisfacer necesidades comerciales particulares.

## Creación de validadores personalizados

Los validadores integrados no cubren todos los escenarios que podríamos encontrar en una aplicación Angular; sin embargo, escribir un validador personalizado y usarlo en Angular es sencillo. En nuestro caso, crearemos un validador para comprobar que el precio de un producto no supere un límite especificado.



Podríamos usar el validador Max integrado para realizar la misma tarea. Sin embargo, desarrollaremos la función de validación con fines de aprendizaje.

Los validadores personalizados se utilizan cuando queremos validar un formulario o un control con datos personalizados.

Código. Por ejemplo, para comunicarse con una API para validar un valor o para realizar un cálculo complejo para validar un valor.

1. Cree un archivo llamado price-maximum.validator.ts en la carpeta src\app y agregue el siguientes contenidos:

```
importar { ValidatorFn, AbstractControl, ValidationErrors } desde '@  
angular/forms';  
  
función de exportación priceMaximumValidator(precio: número): ValidatorFn {  
    devolver (control: AbstractControl): ValidationErrors | null => {  
        const isMax = control.value <= precio;  
        ¿ El retorno es Max? nulo : { precioMáximo: verdadero };  
    };  
}
```

Un validador de formulario es una función que devuelve un objeto ValidationErrors con el error especificado o un valor nulo . Acepta como parámetro el control de formulario al que se aplicará. En el fragmento anterior, si el valor del control supera un umbral específico establecido en el parámetro de precio de la función exportada, devuelve un objeto de error de validación.

De lo contrario, devuelve nulo.

La clave del objeto de error de validación especifica un nombre descriptivo para el error del validador.

Es un nombre que podemos verificar más tarde con el método `hasError` del control para saber si

Contiene algún error. El valor del objeto de error de validación puede ser cualquier valor arbitrario que podamos pasar en el mensaje de error.

2. Abra el archivo `product-create.component.ts` y agregue la siguiente declaración de importación :

```
importar {priceMaximumValidator} desde '../price-maximum.validator';
```

3. Agregue el validador en la matriz de validadores del control de formulario de precio y establezca el umbral hasta 1000:

```
precio: nuevo FormControl<número | indefinido>(indefinido, {
    no nulo: verdadero,
    validadores:
        [ Validadores.obligatorio,
            Validadores.min(1),
            priceMaximumValidator(1000)
        ]
})
```

4. Agregue un nuevo elemento HTML `<span>` para el control del formulario de precio en la creación del producto.

archivo `component.html` :

```
<div>
    <label for="price">Precio</label> <input
        id="price" formControlName="price" type="number" required />

    @if (productForm.controls.price.touched && productForm.controls.
        precio.hasError('requerido')) {
        <span class="help-text"> Se requiere el precio

    } @if (productForm.controls.price.touched && productForm.controls.price.hasError('min')) {

        <span class="help-text">El precio debe ser mayor que 0

    } @if (productForm.controls.price.touched && productForm.controls.price.hasError('precioMaximo'))
    {
        <span class="help-text">El precio debe ser menor o igual a 1000</
    }
</div>
```

5. Ejecute el comando ng serve para iniciar la aplicación y navegue a <http://localhost:4200/productos/nuevo>.
6. Ingrese un valor de 1200 en el campo Precio , haga clic fuera del cuadro de entrada y observe el resultado:

## Add new product

Title

Price

Price must be smaller or equal to 1000

Category

**Create**

Figura 10.7: Validación en formas reactivas

Para utilizar el validador de precio máximo en un formato basado en plantillas, debemos seguir un enfoque diferente que implica la creación de una directiva Angular:

1. Ejecute el siguiente comando para crear una directiva Angular:

```
ng generar directiva precio-máximo
```

La directiva anterior actuará como un envoltorio sobre la función priceMaximumValidator que ya hemos creado.

2. Abra el archivo price-maximum.directive.ts y modifique las declaraciones de importación de la siguiente manera:

```
importar { Directiva, entrada, atributoNumber } de '@angular/core'; importar
{ AbstractControl, NG_VALIDATORS, ValidationErrors, Validator } de '@angular/forms';
importar { priceMaximumValidator }
de './price-maximum.validator';
```

3. Agregue el proveedor NG\_VALIDATORS en el decorador @Directive :

```
@Directiva(
  selector: '[appPriceMaximum]',
```

```

proveedores: [
  {
    proporcionar: NG_VALIDATORS,
    utilizarExistente: PrecioMáximoDirective,
    multi: verdadero
  }
]
})

```

El token NG\_VALIDATORS es un token integrado en los formularios de Angular que nos permite registrar una directiva de Angular como validador de formulario. En el fragmento anterior, usamos la propiedad "multi" en la configuración del proveedor porque podemos registrar varias directivas con el token NG\_VALIDATORS .

4. Agregue la interfaz Validator en las interfaces implementadas de PriceMaximumDirective clase:

clase de exportación PriceMaximumDirective implementa Validator

5. Agregue la siguiente propiedad de entrada que se utilizará para pasar un valor para el máximo límite:

```

appPriceMaximum = entrada(indefinido, {
  alias: 'umbral',
  transformar: atributoNúmero
});

```

En la propiedad anterior, pasamos un objeto de configuración con dos propiedades como parámetro en la función de entrada . La propiedad alias define el nombre de la propiedad de entrada que usaremos para la vinculación. La propiedad transform se usa para convertir el valor de la propiedad de entrada a un tipo diferente. El atributo numberAttribute es una función integrada del framework Angular que convierte el valor de la propiedad de entrada a un número.



Angular también contiene la función booleanAttribute , que analiza un valor de propiedad de entrada como un booleano.

6. Implemente el método de validación de la interfaz Validator de la siguiente manera:

```
validar(control: AbstractControl): ErroresDeValidación | null {
```

```
    devuelve este.appPriceMaximum  
    & priceMaximumValidator(this.appPriceMaximum()!)(control) : null;  
}
```

La firma del método de validación coincide con la de la función `priceMaximumValidator`. Comprueba la propiedad de entrada `appPriceMaximum` y, en consecuencia, delega su valor a la función `priceMaximumValidator`.

Utilizaremos la nueva directiva que creamos en el componente de detalle del producto:

1. Abra el archivo `product-detail.component.ts` y agregue la siguiente declaración de importación :

```
importar { PriceMaximumDirective } desde './price-maximum.directive';
```

2. Agregue la clase `PriceMaximumDirective` en la matriz de importaciones del decorador `@Component` :

```
@Componente({  
  selector: 'app-product-detail', importa: [  
  
    Módulo común,  
    Módulo de formularios,  
    Directiva de precio máximo  
  ],  
  URL de plantilla: './product-detail.component.html', URL de estilo: './product-  
  detail.component.css' })
```

3. Abra el archivo `product-detail.component.html` y agregue el nuevo validador en `<input>` Elemento HTML:

```
<entrada  
  placeholder="Nuevo precio"  
  type="número"  
  nombre="precio"  
  requerido min="1"  
  Umbral de appPriceMaximum="500"  
  #priceCtrl="ngModel"  
  [(ngModel)]="precio" />
```

4. Agregue un nuevo elemento HTML <span> para mostrar un mensaje diferente cuando el validador lanza una excepción. un error:

```
@if (priceCtrl.dirty && priceCtrl.hasError('priceMaximum')) {  
    <span class="help-text">El precio debe ser menor o igual a 500</span>  
}
```

5. Ejecute el comando ng serve para iniciar la aplicación y seleccionar un producto de la lista.

6. Ingrese el valor 600 en el cuadro de entrada Nuevo precio y observe el resultado:

You selected: **SanDisk SSD PLUS 1TB Internal SSD - SATA III  
6 Gb/s**

€109.00

A screenshot of a web application interface. At the top, there is a message: "You selected: SanDisk SSD PLUS 1TB Internal SSD - SATA III 6 Gb/s". Below it is a price value of "€109.00". A text input field contains the value "600", which is highlighted with a red border. To the right of the input field is a "Change" button. Below the input field, two error messages are displayed in red: "Please enter a valid price" and "Price must be smaller or equal to 500". Further down, there is a category label "electronics" and a red "Delete" button.

Figura 10.8: Validación en formularios basados en plantillas

Las validaciones personalizadas de Angular pueden funcionar de forma síncrona o asíncrona. En esta sección, aprendimos a trabajar con las primeras. Las validaciones asíncronas son un tema avanzado que no abordaremos en este libro. Sin embargo, puede obtener más información en <https://angular.dev/guide/forms/form-validation#creando-validadores-asincrónicos>.

En la siguiente sección, exploraremos la manipulación del estado de un formulario angular.

## Manipulación del estado del formulario

El estado de un formulario Angular difiere entre los formularios basados en plantillas y los formularios reactivos. En los primeros, el estado es un objeto simple, mientras que en los segundos, se guarda en el modelo del formulario. En esta sección, aprenderemos los siguientes conceptos:

- Actualización del estado del formulario
- Reaccionar a los cambios de estado

Comenzaremos explorando cómo podemos cambiar el estado del formulario.

### Actualización del estado del formulario

Trabajar con el estado del formulario en formularios basados en plantillas es relativamente sencillo. Debemos interactuar con la propiedad del componente asociada a la directiva ngModel de un control de formulario.

En formularios reactivos, podemos usar la propiedad de valor de una instancia de FormControl o los siguientes métodos de la clase FormGroup para cambiar valores en todo el formulario:

- setValue: Reemplaza valores en todos los controles del formulario
- patchValue: Actualiza valores en controles específicos del formulario

El método setValue acepta un objeto como parámetro que contiene pares clave-valor para todos los controles de formulario. Si queremos completar los detalles de un producto en el componente de creación de productos mediante programación, el siguiente fragmento sirve como ejemplo:

```
este.productForm.setValue({  
    Título: 'Monitor de TV',  
    precio: 600,  
    categoría: 'electrónica'  
});
```

En el fragmento anterior, cada clave del objeto pasado en el método setValue debe coincidir con el nombre de cada control de formulario. Si omitimos una, Angular generará un error.

Si queremos completar algunos de los detalles de un producto, podemos utilizar el método patchValue :

```
este.productForm.patchValue({  
    Título: 'Monitor de TV',  
    categoría: 'electrónica'  
});
```

Los métodos `setValue` y `patchValue` de la clase `FormGroup` nos ayudan a establecer datos en un formulario.

Otro aspecto interesante de los formularios es que podemos ser notificados cuando estos valores cambien, como veremos en la siguiente sección.

## Reaccionando a los cambios de estado

Un escenario común al trabajar con formularios Angular es que queremos activar un efecto secundario cuando cambia el valor de un control de formulario. Un efecto secundario puede ser cualquiera de los siguientes:

- Para alterar el valor de un control de formulario
- Para iniciar una solicitud HTTP para filtrar el valor de un control de formulario
- Para habilitar o deshabilitar ciertas partes de la plantilla del componente

En formularios basados en plantillas, podemos usar una versión extendida de la directiva `ngModel` para recibir notificaciones cuando su valor cambia. La directiva `ngModel` contiene las siguientes propiedades enlazables:

- `ngModel`: una propiedad de entrada para pasar valores al control
- `ngModelChange`: una propiedad de salida para recibir notificaciones cuando cambia el valor del control

Podemos escribir el enlace `ngModel` en el elemento HTML `<input>` del componente de detalle del producto de la siguiente manera alternativa:

```
<entrada
  placeholder="Nuevo precio"
  tipo="número"
  nombre="precio"
  requerido min="1"
  Umbral de appPriceMaximum="500"
  #priceCtrl="ngModel"
  [ngModel]="precio"
  (ngModelChange)="precio = $evento" />
```

En el fragmento anterior, establecimos el valor de la propiedad de entrada `ngModel` mediante la vinculación de propiedades y el valor de la propiedad del componente de precio mediante la vinculación de eventos. Angular activa automáticamente el evento `ngModelChange` e incluye el nuevo valor del elemento HTML `<input>` en la propiedad `$event`. Podemos usar el evento `ngModelChange` para cualquier efecto secundario en nuestro componente cuando cambia el valor del control de formulario de precio.

En los formularios reactivos, utilizamos una API basada en observables para reaccionar a los cambios de estado. Las clases FormGroup y FormControl contienen el observable valueChanges , que podemos usar para suscribirnos y recibir notificaciones cuando el valor del formulario o control cambie.

Lo usaremos para restablecer el valor del control de formulario de precio en el componente de creación de producto cuando cambie la categoría:

1. Abra el archivo product-create.component.ts e importe el artefacto OnInit desde @

Paquete npm angular/core :

```
importar { Component, OnInit } desde '@angular/core';
```

2. Agregue la interfaz OnInit a la lista de la clase ProductCreateComponent implementada interfaces:

La clase de exportación ProductCreateComponent implementa OnInit

3. Cree el siguiente método ngOnInit para suscribirse a la propiedad valueChanges del control de formulario de categoría :

```
ngOnInit(): vacío {
  este.productForm.controla.category.valueChanges.subscribe(() => {
    este.productForm.controla.price.reset();
  });
}
```

En el método anterior, restablecemos el valor del control del formulario de precio mediante el uso del comando de restablecimiento. método de la clase FormControl .



La propiedad valueChanges de la clase FormControl es un flujo observable estándar. No olvide cancelar la suscripción cuando se destruya el componente.

Por supuesto, podemos hacer más con el observable valueChanges ; por ejemplo, podríamos comprobar si el título del producto ya está reservado enviándolo a una API de backend. Esperamos, sin embargo, que los ejemplos anteriores hayan mostrado cómo aprovechar la naturaleza reactiva de los formularios y responder en consecuencia.

## Resumen

En este capítulo, aprendimos que Angular ofrece dos opciones diferentes para crear formularios: basados en plantillas y reactivos, y ninguna es mejor que la otra. Exploramos cómo crear cada tipo de formulario y realizar validaciones con los datos de entrada, y cubrimos validaciones personalizadas para implementar escenarios de validación adicionales. También aprendimos a actualizar el estado de un formulario y a reaccionar cuando cambian los valores de dicho estado.

En el siguiente capítulo, exploraremos diversas maneras de gestionar errores de aplicación. El manejo de errores es una característica muy importante de una aplicación Angular y puede tener diferentes orígenes y motivos, como veremos.

# 11

## Manejo de errores de aplicación

Los errores de aplicación son parte integral del ciclo de vida de una aplicación web. Pueden ocurrir durante el tiempo de ejecución o durante el desarrollo de la aplicación. Las posibles causas de un error de ejecución son una solicitud HTTP fallida o un formulario HTML incompleto. Una aplicación web debe gestionar los errores de ejecución y mitigar los efectos no deseados para garantizar una experiencia de usuario fluida.

Los errores de desarrollo suelen ocurrir cuando no se utiliza correctamente un lenguaje o framework de programación según su semántica. En este caso, los errores pueden sobrescribir el compilador y aparecer en la aplicación durante su ejecución. Los errores de desarrollo se pueden mitigar siguiendo las mejores prácticas y técnicas de codificación recomendadas.

En este capítulo, aprenderemos a gestionar diferentes tipos de errores en una aplicación Angular y a comprender los errores del propio framework. Exploraremos los siguientes conceptos con más detalle:

- Manejo de errores de tiempo de ejecución
- Desmitificando los errores del marco

### Requisitos técnicos

Los ejemplos de código descritos en este capítulo se pueden encontrar en la carpeta ch11 del siguiente Repositorio de GitHub:  
<https://www.github.com/PacktPublishing/Aprendizaje-Angular-Quinta-Edición>

## Manejo de errores de tiempo de ejecución

Los errores de ejecución más comunes en una aplicación Angular se deben a la interacción con una API HTTP.

Introducir credenciales de inicio de sesión incorrectas o enviar datos en un formato incorrecto puede generar un error HTTP. Una aplicación Angular puede gestionar errores HTTP de las siguientes maneras:

- Explícitamente durante la ejecución de una solicitud HTTP particular
- Globalmente en el controlador de errores global de la aplicación
- Utilizando de forma centralizada un interceptor HTTP

En la siguiente sección, exploraremos cómo manejar un error HTTP en una solicitud HTTP específica.

## Cómo detectar errores de solicitud HTTP

La gestión de errores en las solicitudes HTTP suele requerir la inspección manual de la información devuelta en el objeto de respuesta de error. RxJS proporciona el operador catchError para simplificar esta tarea. Este operador puede detectar posibles errores al iniciar una solicitud HTTP con el operador de canalización .



Necesitará el código fuente de la aplicación Angular que creamos en el Capítulo 10, Recopilación de datos de usuario con formularios, para continuar con el resto del capítulo.

Veamos cómo podemos utilizar el operador catchError para capturar errores HTTP al obtener la lista de productos en nuestra aplicación:

1. Abra el archivo products.service.ts e importe los operadores catchError y throwError del paquete npm rxjs :

```
importar { Observable, mapa, de, toque, catchError, throwError } de  
'rxjs';
```

2. Importe la interfaz HttpErrorResponse desde el espacio de nombres @angular/common/http :

```
importar { HttpClient, HttpParams, HttpErrorResponse } desde '@angular/  
común/http';
```

3. Modifique el método getProducts según corresponda:

```
obtenerProductos(límite?: número): Observable<Producto[]> {  
    si (este.producto.longitud === 0) {  
        const opciones = new HttpParams().set('limit', limit || 10);  
        devuelve este.http.get<Product[]>(este.productsUrl, {
```

```

    parámetros: opciones
}).tubo(
  mapa(productos => {
    este.productos = productos;
    devolver productos;
  }),
  catchError((error: HttpErrorResponse) => {
    consola.error(error);
    devolver throwError(() => error);
  })
);
}
retorno de(este.productos);
}

```

La firma del operador catchError contiene el objeto HttpErrorResponse que devuelve el servidor. Tras detectar el error, usamos el operador throwError . que vuelve a arrojar el error como un observable.



Como alternativa, podríamos haber usado la palabra clave " throw " de los métodos estándar de la API web para generar el error. Sin embargo, el método "throwError" suele ser excesivo. Úselo como corresponda.

De esta manera, garantizamos que la ejecución de la aplicación continuará y se completará sin provocar una posible pérdida de memoria.

En un escenario del mundo real, probablemente crearíamos un método auxiliar para registrar el error en un sistema de seguimiento más sólido y devolver algo significativo según la causa del error:

1. En el mismo archivo, products.service.ts, importe la enumeración HttpStatusCode de el espacio de nombres @angular/common/http :

```
importar { HttpClient, HttpParams, HttpErrorResponse, HttpStatusCode } de '@angular/common/http';
```

HttpStatusCode es una enumeración que contiene una lista de todos los códigos de estado de respuesta HTTP.

2. Cree el siguiente método en la clase ProductsService :

```
manejador privadoError (error: HttpErrorResponse) {
  dejar mensaje = '';
```

```
switch(error.status) { caso
    HttpStatusCode.InternalServerError:
        mensaje = 'Error del servidor';
        romper;
    caso HttpStatusCode.BadRequest:
        mensaje = 'Error de solicitud';
        break;
    por defecto:
        mensaje = 'Error desconocido';
    }

    console.error(mensaje, error.error);

    devolver throwError(() => error);
}
```

El método anterior registra un mensaje diferente en la consola del navegador según el Estado de error. Utiliza una sentencia switch para diferenciar entre errores internos del servidor y solicitudes incorrectas. Para cualquier otro error, se recurre a la instrucción predeterminada , que registra un mensaje genérico en la consola.

3. Refactorice el método getProducts para utilizar el método handleError para detectar errores:

```
obtenerProductos(límite?: número): Observable<Producto[]> {
    if (this.products.length === 0) { const
        opciones = new HttpParams().set('limit', limit || 10); return
        this.http.get<Product[]>(this.productsUrl, { params:
            opciones }).pipe( map(products =>
            { this.products =
                productos; return productos; }), catchError(this.handleError)
        );
    }
} retorno de(este.productos);
```

El método handleError actualmente gestiona errores HTTP originados únicamente en la respuesta HTTP . Sin embargo, pueden ocurrir otros errores en una aplicación Angular desde el lado del cliente, como una solicitud que no llegó al servidor debido a un error de red o una excepción lanzada en un operador RxJS. Para gestionar cualquiera de los errores anteriores, debemos agregar una nueva declaración case en el Método handleError :

```
manejador privadoError (error: HttpErrorResponse) {  
    dejar mensaje = "";  
  
    switch(error.estado) {  
        caso 0:  
            mensaje = 'Error de cliente';  
            romper;  
        caso HttpStatusCode.InternalServerError:  
            mensaje = 'Error del servidor';  
            romper;  
        caso HttpStatusCode.BadRequest:  
            mensaje = 'Error de solicitud';  
            romper;  
        por defecto:  
            mensaje = 'Error desconocido';  
    }  
  
    console.error(mensaje, error.error);  
  
    devolver throwError(() => error);  
}
```

En el fragmento anterior, un error con un estado de 0 indica que es un error que ocurrió en el lado del cliente de la aplicación.

La gestión de errores en las solicitudes HTTP podría combinarse con un mecanismo que reintente una llamada HTTP determinada una cantidad específica de veces antes de gestionar el error. Existe un operador RxJS para casi todo, incluso uno para intentar solicitudes HTTP. Acepta el número de intentos necesarios para ejecutar la solicitud en particular hasta que se complete correctamente:

```
obtenerProductos(límite?: número): Observable<Producto[]> {  
    si (este.producto.longitud === 0) {  
        const opciones = new HttpParams().set('limit', limit || 10);  
    }  
}
```

```

devuelve este.http.get<Product[]>(este.productsUrl, {
    parámetros: opciones
}).tubo(
    mapa(productos => {
        este.productos = productos;
        devolver productos;
    }),
    reintentar(2),
    catchError(este.manejarError)
);
}

retorno de(este.productos);
}

```

Aprendimos que usamos el operador catchError de RxJS para capturar errores. La forma en que lo gestionamos depende del escenario. En nuestro caso, creamos un método handleError para todas las llamadas HTTP en un servicio. En un escenario real, seguiríamos el mismo enfoque de gestión de errores en otros servicios Angular de una aplicación. Crear un método para cada servicio no sería conveniente.

y no escala bien.

Como alternativa, podríamos utilizar el controlador de errores global que Angular proporciona para gestionar los errores de forma centralizada. Aprenderemos a crear un controlador de errores global en la siguiente sección.

## Creación de un controlador de errores global

El framework Angular proporciona la clase ErrorHandler para gestionar errores globalmente en una aplicación Angular. La implementación predeterminada de la clase ErrorHandler imprime mensajes de error en la ventana de la consola del navegador.

Para crear un controlador de errores personalizado para nuestra aplicación, necesitamos subclasicar la clase ErrorHandler y proporcionar nuestra implementación personalizada para el registro de errores:

1. Cree un archivo llamado app-error-handler.ts en la carpeta src\app de la CLI de Angular espacio de trabajo.

2. Abra el archivo y agregue las siguientes declaraciones de importación :

```

importar { HttpResponseMessage, HttpStatusCode } desde '@angular/common/
http';
importar { ErrorHandler, Injectable } desde '@angular/core';

```

3. Cree una clase TypeScript que implemente la interfaz ErrorHandler :

```
@Injectable()  
exporta la clase AppErrorHandler implementa ErrorHandler {}
```

La clase AppErrorHandler debe estar decorada con el decorador @Injectable() porque lo proporcionaremos más adelante en el archivo de configuración de la aplicación.

4. Implemente el método handleError desde la interfaz ErrorHandler de la siguiente manera:

```
handleError(error: cualquiera): void {  
    const err = error.rechazo || error;  
    dejar mensaje = "";  
  
    si (err instancia de HttpErrorResponse) {  
        switch(err.estado) {  
            caso 0:  
                mensaje = 'Error de cliente';  
                romper;  
            caso HttpStatusCode.InternalServerError:  
                mensaje = 'Error del servidor';  
                break;  
            caso HttpStatusCode.BadRequest:  
                mensaje = 'Error de solicitud';  
                romper;  
            por defecto:  
                mensaje = 'Error desconocido';  
        }  
    } else  
        { mensaje = 'Error de aplicación';  
    }  
  
    console.error(mensaje, err);  
}
```

En el método anterior, verificamos si el objeto de error contiene una propiedad de rechazo . Los errores que se originan en la biblioteca Zone.js , que es responsable de la detección de cambios en Angular, encapsulan el error real dentro de esa propiedad.

Tras extraer el error en la variable `err`, comprobamos si se trata de un error HTTP mediante el tipo `HttpErrorResponse`. Esta comprobación detectará cualquier error de las llamadas HTTP mediante el operador `throwError` de RxJS. Todos los demás errores se tratan como errores de aplicación que ocurren en el lado del cliente.

5. Abra el archivo `app.config.ts` e importe la clase `ErrorHandler` desde `@angular/core`:

```
importar { ApplicationConfig, ErrorHandler, provideZoneChangeDetection } desde  
'@angular/core';
```

6. Importe el controlador de errores personalizado que creamos en el archivo `app-error-handler.ts`:

```
importar { AppErrorHandler } desde './app-error-handler';
```

7. Registre la clase `AppErrorHandler` como el controlador de errores global de la aplicación agregando a la matriz de proveedores de la variable `appConfig`:

```
exportar const appConfig: ApplicationConfig = {  
  proveedores: [  
    proporcionarZoneChangeDetection({ eventCoalescing: true }),  
    proporcionarRouter(rutas),  
    proporcionarHttpClient(),  
    { proporcionar: APP_SETTINGS, valor de uso: appSettings },  
    { proporcionar: ErrorHandler, useClass: AppErrorHandler }  
  ]  
};
```

Para investigar el comportamiento del controlador de errores de aplicación global, ejecute los siguientes pasos:

1. Ejecute el comando `ng serve` para iniciar la aplicación.
2. Desconecte su computadora de Internet.
3. Vaya a `http://localhost:4200`.
4. Abra las herramientas para desarrolladores del navegador e inspeccione la salida de la ventana de la consola:



Figura 11.1: Error de aplicación

Uno de los errores HTTP más comunes en una aplicación web empresarial es el error 401 "Respuesta no autorizada". Aprenderemos a gestionar este error específico en la siguiente sección.

## Respondiendo al error 401 No autorizado

El error 401 No autorizado en una aplicación Angular puede ocurrir en los siguientes casos:

- El usuario no proporciona las credenciales correctas al iniciar sesión en la aplicación
- El token de autenticación proporcionado cuando el usuario inició sesión en la aplicación ha expirado

Un buen lugar para gestionar el error 401 No autorizado es dentro de un interceptor HTTP responsable de la autenticación. En el Capítulo 8, Comunicación con Servicios de Datos a través de HTTP, aprendimos a crear un interceptor de autenticación para pasar el token de autorización a cada solicitud HTTP. Para manejar el error 401 No autorizado, el archivo auth.interceptor.ts se puede modificar de la siguiente manera:

```

importar { HttpErrorResponse, HttpInterceptorFn, HttpStatusCode } de '@angular/common/
http'; importar { inyectar }
de '@angular/core'; importar { AuthService } de './
auth.service'; importar { catchError, EMPTY, throwError }
de 'rxjs';

exportar const authInterceptor: HttpInterceptorFn = (req, next) => {
  const authService = inyectar(AuthService); const
  authReq = req.clone({
    setHeaders: { Autorización: 'myToken' } });
  devolver

  next(authReq).pipe(
    catchError((error: HttpErrorResponse) => {
  
```

```
    si (error.estado === HttpStatusCode.No autorizado) {
        authService.logout();
        devuelve VACÍO;
    } demás {
        devolver throwError(() => error);
    }
}
);
};
```

El interceptor llamará al método logout de la clase AuthService cuando se produzca un error 401 No autorizado y devolverá un observable VACÍO para detener la emisión de datos. Utilizará el operador throwError para enviar el error al gestor de errores global en todos los demás errores. Como ya hemos visto, el gestor de errores global examinará el error devuelto y tomará medidas según el código de estado.

Como vimos en el controlador de errores global que creamos en la sección anterior, algunos errores no están relacionados con la interacción con el cliente HTTP. Hay errores de aplicación que ocurren en el lado del cliente, y aprenderemos a comprenderlos en la siguiente sección.

## Desmitificando los errores del marco

Los errores de aplicación que se originan en el lado del cliente en una aplicación Angular pueden tener muchas causas.

Una de ellas es la interacción de nuestro código fuente con el framework Angular. A los desarrolladores les gusta probar cosas y enfoques nuevos al crear aplicaciones. A veces, todo sale bien, pero otras veces, pueden causar errores en una aplicación.

El marco Angular proporciona un mecanismo para informar algunos de estos errores comunes con el siguiente formato:

NGWXYZ: {Mensaje de error}.<Enlace>

Analicemos el formato de error anterior:

- NG: Indica que es un error de Angular para diferenciarlo entre otros errores originados en TypeScript y el navegador.
- W: Un número de un solo dígito que indica el tipo de error. 0 representa un error de tiempo de ejecución y todos los demás números del 1 al 9 representan un error del compilador.
- X: Un número de un solo dígito que indica la categoría del área de tiempo de ejecución del marco, como detección de cambios, inyección de dependencia y plantilla.
- YZ: Un código de dos dígitos utilizado para indexar el error específico

- {Mensaje de error}: El mensaje de error real
- <Link>: Un enlace a la documentación de Angular que proporciona más información sobre el error especificado

Los mensajes de error que se ajustan al formato anterior se muestran en la consola del navegador a medida que ocurren. Veamos un ejemplo de error con el error ExpressionChangedAfterChecked , el más común en las aplicaciones Angular:

### 1. Abra el archivo app.component.ts e importe el artefacto AfterViewInit desde el

Paquete npm de @angular/core :

```
importar { AfterViewInit, Component, inyectar } desde '@angular/core';
```

### 2. Agregue AfterViewInit en la lista de interfaces implementadas:

```
La clase de exportación AppComponent implementa AfterViewInit
```

### 3. Cree la siguiente propiedad de título en la clase AppComponent :

```
título = "";
```

### 4. Implemente el método ngAfterViewInit y cambie la propiedad del título dentro del método.

cuerpo del od:

```
ngAfterViewInit(): vacío {
  este.título = este.configuración.título;
}
```

### 5. Abra el archivo app.component.html y vincule la propiedad de título al elemento HTML <h2> :

```
<h2>{{ título }}</h2>
```

### 6. Ejecute el comando ng serve y navegue a http://localhost:4200.

Inicialmente, todo parece funcionar correctamente. El valor de la propiedad del título se muestra correctamente en la página.

### 7. Abra las herramientas para desarrolladores del navegador e inspeccione la ventana de la consola:

```
Error de aplicación RuntimeError: NG0100:
ExpressionChangedAfterItHasBeenCheckedError: La expresión ha cambiado después
de ser verificada. Valor anterior: ". Valor actual: 'Mi tienda online'. Ubicación
de la expresión: componente _AppComponent. Más información en https://angular.dev/
errors/NG0100
```

El mensaje anterior indica que cambiar el valor de la propiedad del título provocó el error.

8. Haga clic en <https://angular.dev/errors/NG0100> El enlace nos redirigirá a la guía de errores correspondiente en la documentación de Angular para obtener más información. La guía explica el error específico y describe cómo solucionarlo en el código de nuestra aplicación.

Cuando entendemos los mensajes de error que se originan en el marco Angular, podemos solucionarlos fácilmente.

## Resumen

Gestionar errores durante la ejecución o el desarrollo es crucial para cualquier aplicación Angular. En este capítulo, aprendimos a gestionar errores que ocurren durante la ejecución de una aplicación Angular, como errores HTTP o del lado del cliente. También aprendimos a comprender y corregir los errores de aplicación generados por el framework Angular.

En el siguiente capítulo, aprenderemos a mejorar la apariencia de nuestra aplicación con Angular Material. Angular Material cuenta con numerosos componentes y estilos listos para usar en tus proyectos. Así que, demos a tu proyecto Angular el cariño que se merece.

# 12

## Introducción al material angular

Al desarrollar una aplicación web, debes decidir cómo crear tu interfaz de usuario (UI). Idealmente, debería usar colores con el contraste adecuado, tener una apariencia uniforme, ser responsive y funcionar bien en diferentes dispositivos y navegadores. En resumen, hay muchos aspectos a considerar en cuanto a la UI y la UX. Muchos desarrolladores consideran que crear la UI/UX es una tarea abrumadora y recurren a frameworks de UI que se encargan de gran parte del trabajo pesado. Algunos frameworks se usan más que otros, como Bootstrap y Tailwind CSS. Sin embargo, Angular Material, un framework basado en las técnicas de Material Design de Google , ha ganado popularidad. En este capítulo, explicaremos qué es Material Design y cómo Angular Material lo utiliza para proporcionar una biblioteca de componentes de UI para el framework Angular. También aprenderemos a usar varios componentes de Angular Material aplicándolos en nuestra aplicación de tienda online.

En este capítulo haremos lo siguiente:

- Introducción al diseño de materiales
- Introducción a Angular Material
- Integración de componentes de UI

### Requisitos técnicos

El capítulo contiene varios ejemplos de código para guiarlo a través del concepto de Angular Material.

Puede encontrar el código fuente relacionado en la carpeta ch12 del siguiente repositorio de GitHub:

<https://www.github.com/PacktPublishing/Aprendizaje-Angular-Quinta-Edición>

## Introducción a Material Design

Material Design es un lenguaje de diseño desarrollado por Google con los siguientes objetivos en mente:

- Desarrollar un único sistema subyacente, que permita una experiencia unificada en todas las plataformas y tamaños de dispositivos.
- Los preceptos móviles son fundamentales, pero el tacto, la voz, el ratón y el teclado son de primera clase. métodos de entrada.

El propósito de un lenguaje de diseño es que el usuario comprenda cómo debe verse y sentirse la interfaz de usuario y la interacción en diferentes dispositivos. Material Design se basa en tres principios fundamentales:

- El material es la metáfora: se inspira en el mundo físico con diferentes texturas y medios, como el papel y la tinta.
- Atrevido, gráfico e intencional: se guía por diferentes métodos de diseño de impresión, como tipografía, cuadrículas y color, para crear una experiencia inmersiva para el usuario.
- El movimiento aporta significado: los elementos se muestran en la pantalla mediante la creación de animaciones. y las interacciones que reorganizan el entorno.

Material Design tiene mucha teoría detrás, y hay documentación adecuada disponible sobre el tema si deseas profundizar. Puedes encontrar más información en el sitio web oficial de la documentación: <https://material.io>

Un lenguaje de diseño por sí solo no es tan interesante si no eres diseñador. En la siguiente sección, aprenderemos cómo los desarrolladores de Angular pueden beneficiarse de Material Design usando la biblioteca Angular Material.

## Introducción a Angular Material

La biblioteca Angular Material se desarrolló para implementar Material Design en el framework Angular . Se basa en los siguientes conceptos:

- Sprint de cero a la aplicación: La intención es facilitarle las cosas a usted, como desarrollador de aplicaciones. oper, para empezar a funcionar de inmediato. El esfuerzo necesario para su puesta en marcha debería ser mínimo.
- Rápido y consistente: el rendimiento ha sido un punto de enfoque importante y Angular Material Se garantiza que funcionará bien en todos los navegadores principales.
- Versátil: Muchos temas deberían ser fácilmente personalizables y también hay un gran soporte para localización e internacionalización.

- Optimizado para Angular: el hecho de que el equipo de Angular lo haya creado significa que el soporte para Angular es una gran prioridad.

La biblioteca se divide en las siguientes partes principales:

- Componentes: muchos componentes de UI, como diferentes tipos de entrada, botones, diseño, navegación, modales y otras formas de mostrar datos tabulares, están disponibles para ayudarlo a tener éxito.
- Temas: La biblioteca viene con temas preinstalados, pero también hay una guía de temas si quieras crear el tuyo propio en <https://material.angular.io/guide/theming>.



Cada parte y componente de la biblioteca Angular Material encapsula las mejores técnicas de accesibilidad web listas para usar.

El núcleo de la biblioteca Angular Material es el Angular CDK, un conjunto de herramientas que implementan patrones de interacción similares, sin relación con ningún estilo de presentación. El comportamiento de los componentes Angular Material se diseñó utilizando el Angular CDK. El Angular CDK es tan abstracto que permite crear componentes personalizados. Si eres creador de bibliotecas de interfaz de usuario, deberías considerarlo seriamente.

Hemos cubierto toda la teoría básica sobre Angular Material, así que pongámosla en práctica en la siguiente sección integrándola con una aplicación Angular.

## Instalación de material angular

La biblioteca Angular Material es un paquete npm. Para instalarla, debemos ejecutar manualmente el comando `npm install` e importar varios artefactos de Angular a nuestra aplicación. El equipo de Angular ha automatizado estas interacciones creando los esquemas necesarios para su instalación mediante la CLI de Angular.



Necesitará el código fuente de la aplicación Angular que creamos en el Capítulo 11, Manejo de errores de aplicación, para seguir con el resto del capítulo.

Podemos usar el comando `ng add` de la CLI de Angular para instalar Angular Material en nuestra aplicación de tienda electrónica:

1. Ejecute el siguiente comando en el espacio de trabajo actual de Angular CLI:

```
ng agregar @angular/material
```

La CLI de Angular encontrará la última versión estable de la biblioteca Angular Material y nos solicitará que la descarguemos.



En este libro, trabajamos con Angular Material 19, que es compatible con Angular 19. Si la versión que le solicita es diferente, debe ejecutar el comando `ng add @angular/material@19` para instalar la última versión de Angular Material 19 en su sistema.

2. Una vez finalizada la descarga, nos preguntará si queremos utilizar un tema prediseñado.

para nuestra aplicación Angular o una personalizada:

```
Elija un nombre de tema prediseñado o "personalizado" para un tema personalizado: (Use las teclas de flecha)
```

Acepte el valor predeterminado, Azure/Azul, presionando Enter.

3. Tras seleccionar un tema, la CLI de Angular nos preguntará si queremos configurar estilos tipográficos globales en nuestra aplicación. La tipografía se refiere a cómo se organiza el texto en nuestra aplicación:

```
¿Configurar estilos tipográficos globales de Angular Material? (sí/no)
```

Queremos mantener nuestra aplicación lo más simple posible, así que acepte el valor predeterminado, No, presionando Enter.



La tipografía Angular Material se basa en las pautas de Material Design y utiliza la fuente Roboto de Google para el estilo.

4. La siguiente pregunta trata sobre las animaciones. No es estrictamente necesario, pero queremos que nuestra aplicación muestre una animación atractiva al hacer clic en un botón o abrir un cuadro de diálogo modal.

```
¿Incluir el módulo de animaciones angulares? (Usar las teclas de flecha)
```

Acepte el valor predeterminado, incluya y habilite las animaciones, presionando Enter.

La CLI de Angular comenzará a instalar y configurar Angular Material en nuestra aplicación. Creará el andamiaje e importará todos los artefactos necesarios para que podamos empezar a trabajar con Angular Material inmediatamente.

- angular.json: Agrega el archivo de la hoja de estilo del tema en el archivo de configuración de Angular

Espacio de trabajo CLI:

```
"estilos": [  
    "@angular/material/temas-preconstruidos/azure-blue.css",  
    "src/styles.css"  
]
```

- package.json: agrega los paquetes npm @angular/cdk y @angular/material .
- index.html: Agrega los archivos de la hoja de estilo de las fuentes Roboto y los íconos de Material en el archivo HTML principal.
- styles.css: Agrega los estilos CSS globales necesarios para las etiquetas <html> y <body> :

```
html, cuerpo { altura: 100%; }  
cuerpo { margen: 0; familia de fuentes: Roboto, "Helvetica Neue", sans-serif; }
```

- app.config.ts: Habilita animaciones en el archivo de configuración de la aplicación:

```
importar { provideHttpClient } desde '@angular/common/http';  
importar { ApplicationConfig, ErrorHandler, provideZoneChangeDetection } desde  
'@angular/core';  
importar { provideRouter } desde '@angular/router';  
  
importar { rutas } desde './app.routes';  
importar { APP_SETTINGS, appSettings } desde './app.settings';  
importar { AppErrorHandler } desde './app-error-handler';  
importar { provideAnimationsAsync } desde '@angular/platform-browser/  
animaciones/async';  
  
exportar const appConfig: ApplicationConfig = {  
    proveedores: [  
        proporcionarZoneChangeDetection({ eventCoalescing: true }),  
        proporcionarRouter(rutas),  
        proporcionarHttpClient(),  
        { proporcionar: APP_SETTINGS, valor de uso: appSettings },
```

```

    { proporcionar: ErrorHandler, useClass: AppErrorHandler },
    proporcionarAnimacionesAsync()
]
};

```

Una vez finalizado el proceso, podemos comenzar a agregar componentes de UI de la biblioteca Angular Material a nuestra aplicación.

#### Agregar componentes de interfaz de usuario

El componente de botón es uno de los componentes más utilizados de la biblioteca Angular Material.

A modo de ejemplo, aprenderemos lo fácil que es agregar un componente de botón a nuestra aplicación de tienda electrónica.

Antes de poder usarlo en nuestra aplicación Angular, debemos eliminar todos los estilos CSS de la etiqueta nativa <button> que hemos usado hasta ahora:

1. Abra el archivo style.css y elimine el botón, button:hover y button:disabled

Estilos CSS.

2. Abra el archivo product-detail.component.css y elimine la variable --button-accent de los estilos button.secondary y button.delete .

3. Elimine por completo el estilo CSS .button-group .

4. Agrega un color al estilo del botón eliminar :

```

botón.eliminar {
  pantalla: en línea;
  margen izquierdo: 5px;
  color: marrón;
}

```

Para empezar a usar un componente de interfaz de usuario de la biblioteca Angular Material, debemos importar su componente Angular correspondiente. Veamos cómo se hace añadiendo un componente de botón al componente de autenticación de la aplicación Angular:

1. Abra el archivo auth.component.ts y agregue la siguiente declaración de importación para usar Angular Botones de material:

```
importar { MatButton } desde '@angular/material/button';
```

No importamos directamente desde el paquete @angular/material porque cada componente tiene un espacio de nombres dedicado. El componente del botón se encuentra en @angular/espacio de nombres material/botón .



Los componentes de Angular Material también se pueden usar importando su módulo correspondiente, como MatButtonModule para botones. Se recomienda importar los componentes directamente, ya que nos ayuda a mantener la coherencia con los patrones modernos de Angular. Sin embargo, veremos que algunas funciones requieren demasiados componentes para importar. En esos casos, es aceptable importar el módulo directamente.

2. Agregue la clase MatButton en la matriz de importaciones del decorador @Component :

```
@Component({
  selector: 'app-auth',
  importaciones: [MatButton],
  URL de plantilla: './auth.component.html',
  URL de estilo: './auth.component.css'
})
```

3. Abra el archivo auth.component.html y agregue la directiva mat-button en el <button>

Elementos HTML:

```
@if (!authService.isLoggedIn()) {
  Iniciar sesión
} @demás {
  Cerrar sesión
}
```

En la plantilla anterior, la directiva mat-button , en esencia, modifica el <button> elemento para que aparezca y se comporte como un botón de Material Design.

Si ejecutamos el comando ng serve y navegamos a http://localhost:4200, observaremos que el estilo del botón es diferente. Se parece más a un enlace, que es la apariencia predeterminada de un botón Material. En la siguiente sección, aprenderemos sobre temas y variaciones del componente del botón.

Tematización de componentes de interfaz de usuario

La biblioteca Angular Material viene con cuatro temas integrados:

- Azul celeste
- Rosa/Rojo

- Magenta/Violeta • Cian/
- Naranja

Al añadir Angular Material a una aplicación Angular, podemos elegir cuál de los temas anteriores queremos aplicar. Siempre podemos cambiarlo modificando el archivo de hoja de estilos CSS incluido en el archivo de configuración angular.json . Aquí tienes un ejemplo:

```
"estilos": [
  "/@angular/material/temas-preconstruidos/azure-blue.css", "src/
  styles.css"
]
```

Como vimos en la sección anterior, el componente del botón se muestra como un enlace. La directiva mat-button muestra un color de fondo solo al pasar el cursor sobre el botón. Para establecer el color de fondo de forma permanente, debemos usar la directiva mat-flat-button de la siguiente manera:

```
@if (!authService.isLoggedIn()) {
  <button mat-flat-button (clic)="iniciar sesión()">
    Iniciar
    sesión
} @else
{ <button mat-flat-button (clic)="cerrar sesión()">
  Cerrar sesión
</botón>
}
```

Ahora que sabemos cómo interactuar con el componente de botón en una aplicación Angular, aprendamos algunas de sus variaciones:

1. Abra el archivo product-create.component.ts y agregue la siguiente declaración de importación :

```
importar { MatButton } desde '@angular/material/button';
```

2. Agregue la clase MatButton en la matriz de importaciones del decorador @Component :

```
@Component({
  selector: 'app-product-create',
  importaciones: [ReactiveFormsModule, MatButton],
  templateUrl: './product-create.component.html',
  estiloUrl: './product-create.component.css' })
```

3. Abra el archivo `product-create.component.html` y agregue la directiva `mat-raised-button` en el elemento HTML `<button>` :

```
<botón  
    botón levantado del tapete  
    tipo="enviar"  
    [deshabilitado]="productForm.invalid">  
    Crear  
</botón>
```

La directiva `mat-raised-button` agregará una sombra al elemento del botón:

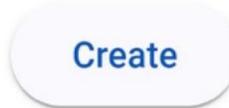


Figura 12.1: Botón elevado

4. Abra el archivo `product-detail.component.ts` y repita los pasos 1 y 2.

5. Abra el archivo `product-detail.component.html` y agregue la directiva `mat-stroked-button` en el botón Cambiar :

```
<botón  
    botón acariciado por el tapete  
    clase="secundaria"  
    tipo="enviar"  
    [deshabilitado]="priceForm.invalid">  
    Cambiar  
</botón>
```

La directiva `mat-stroked-button` agrega un borde alrededor del elemento del botón:



Figura 12.2: Botón pulsado

6. Elimine el elemento HTML `<div>` con la clase `button-group` y agregue el `mat-raised-button` Directiva de botón en ambos elementos HTML `<button>` :

```
@if (authService.isLoggedIn()) {
```

```

<botón
    botón levantado del tapete
    (clic)="addToCart(producto.id)">
    Añadir a la cesta
</botón>
}

<botón
    botón levantado del tapete
    clase="eliminar"
    (clic)="eliminar(producto)">
    Borrar
</botón>

```

Los dos botones aparecen de la siguiente manera cuando ejecutamos la aplicación:

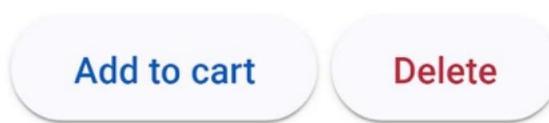


Figura 12.3: Botones de acción de detalles del producto

7. Abra el archivo product-list.component.ts y agregue las siguientes declaraciones de importación :

```

importar { MatMiniFabButton } de '@angular/material/button'; importar { MatIcon }
de '@angular/material/icon';

```

8. Agregue las clases importadas anteriores en la matriz de importaciones del decorador @Component :

```

@Component({
  selector: 'app-product-list',
  imports: [
    SortPipe,
    Tubería asíncrona,
    Enlace de enrutador,
    Botón MatMiniFab,
    MatIcon
  ],
  URL de plantilla: './lista-de-productos.componente.html',
  URL de estilo: './lista-de-productos.componente.css'
})

```

9. Abra el archivo product-list.component.html y reemplace el elemento de anclaje que lleva al componente de creación de producto con el siguiente fragmento HTML:

```
<button mat-mini-fab routerLink="nuevo">  
  <mat-icon>añadir</mat-icon>  
</botón>
```

La directiva mat-mini-fab muestra un botón cuadrado con esquinas redondeadas y un ícono indicado por el elemento HTML <mat-icon>. El texto del elemento <mat-icon> corresponde al nombre del ícono de adición de la colección de iconos de Material Design:



Figura 12.4: Botón FAB

La creación de temas en Angular Material es tan amplia que podemos usar variables CSS existentes para crear temas personalizados, un tema que está fuera del alcance de este libro.

Para continuar nuestro viaje por el mundo del estilo con Angular Material, aprenderemos cómo integrar varios componentes de UI en la siguiente sección.

## Integración de componentes de UI

Angular Material contiene muchos componentes de UI organizados en categorías en <https://material.angular.io/componentes/categorías>. En este capítulo, exploraremos un subconjunto de la colección anterior que se puede agrupar en las siguientes categorías:

- Controles de formulario: se pueden usar dentro de un formulario Angular, como autocompletar, entrada, y lista desplegable.
- Navegación: Proporcionan capacidades de navegación, como un encabezado y un pie de página.
- Diseño: Definen cómo se representan los datos, como una tarjeta o una tabla.
- Ventanas emergentes y superposiciones: son ventanas superpuestas que muestran información y pueden bloquear cualquier interacción del usuario hasta que se descarten de alguna manera.

En las siguientes secciones, exploraremos cada categoría con más detalle.

## Controles de formulario

Aprendimos en el Capítulo 10, Recopilación de datos de usuario con formularios, que los controles de formulario tienen como objetivo recopilar datos de entrada de diferentes maneras y tomar medidas adicionales, como enviar datos a una API de backend. a través de HTTP.

Hay bastantes controles de formulario en la biblioteca Angular Material de distintos tipos, a saber:

- Autocompletar: Permite al usuario comenzar a escribir en un campo de entrada y recibir sugerencias mientras escribe. Ayuda a limitar los posibles valores que puede tomar la entrada.
- Casilla de verificación: una casilla de verificación clásica que representa un estado marcado o desmarcado.
- Selector de fecha: permite al usuario seleccionar una fecha en un calendario.
- Entrada: un control de entrada clásico mejorado con una animación significativa mientras se escribe.
- Botón de opción: un botón de opción clásico mejorado con animaciones y transiciones mientras Edición para crear una mejor experiencia de usuario.
- Seleccionar: un control desplegable que solicita al usuario que seleccione uno o más elementos de una lista.
- Control deslizante: permite al usuario aumentar o disminuir un valor arrastrando un botón deslizante hacia la derecha o hacia la izquierda.
- Interruptor deslizante: un interruptor que el usuario puede deslizar para activarlo o desactivarlo.
- Chips: una lista que muestra, selecciona y filtra elementos.

En las siguientes secciones, examinaremos algunos de estos controles de formulario con más detalle. Empecemos por el componente de entrada.

## Aporte

El componente de entrada suele estar asociado a un elemento HTML <input> . También podemos añadir la opción de mostrar errores en el campo de entrada.

Antes de poder usar el componente de entrada en nuestra aplicación Angular, debemos eliminar todos los estilos CSS de la etiqueta nativa <input> que hemos usado hasta ahora:

1. Abra el archivo style.css y elimine cualquier estilo CSS que haga referencia a la etiqueta de entrada .
2. Elimine el estilo CSS de entrada de product-create.component.css y del carrito. archivos component.css .

Para aprender a utilizar el componente de entrada, lo integraremos en los componentes de nuestra aplicación:

1. Abra el archivo product-create.component.ts y agregue las siguientes declaraciones de importación :

```
importar { MatInput } de '@angular/material/input'; importar
{ MatFormField, MatError, MatLabel } de '@angular/material/ campo-de-formulario';
```

2. Agregue las clases importadas anteriores en la matriz de importaciones del decorador @Component :

```
@Componente(
  selector: 'app-product-create',
  importaciones: [
    Módulo de formularios reactivos,
    Botón Mat,
    MatInput,
    MatFormField,
    MatError,
    MatLabel
  ],
  URL de plantilla: './product-create.component.html', URL de
  estilo: './product-create.component.css' })
```

3. Abra el archivo product-create.component.html y reemplace las etiquetas <div> de <input>

Elementos HTML como sigue:

```
<campo-mat-form>
  <mat-label>Título</mat-label>
  <input formControlName="título" matInput requerido />
  <mat-error>El título es obligatorio</mat-error>
</mat-form-field>
<mat-form-field>
  <mat-label>Precio</mat-label>
  <input formControlName="precio" matInput tipo="número" requerido /> @if
  (productForm.controls.price.touched && productForm.controls.
  precio.hasError('requerido')) {
    <mat-error> Se requiere precio</mat-error>
  }
}
```

```

@if (productForm.controls.price.touched && productForm.controls.
precio.hasError('min')) {
    <mat-error>El precio debe ser mayor que 0</mat-error>
}
@if (productForm.controls.price.touched && productForm.controls.
precio.hasError('precioMáximo')) {
    <mat-error>El precio debe ser menor o igual a 1000</mat-error>
}
</mat-form-field>

```

En el fragmento HTML anterior, utilizamos la directiva matInput para indicar que un <input> El elemento HTML es un componente de entrada de Angular Material. Un control de formulario en Angular Ma- El material debe estar encerrado en un elemento <mat-form-field> .

Hemos reemplazado todos los elementos HTML <label> por elementos <mat-label> . Un <mat-label> El elemento HTML es una etiqueta que apunta a un control de formulario de Angular Material específico.

El elemento <mat-error> muestra mensajes de error en los controles de formulario cuando Angular activa errores de validación. Se muestra por defecto cuando el estado del control de formulario no es válido. En todos los demás casos, podemos usar un bloque @if para controlar cuándo se activará el elemento <mat-error> . se mostrará.

4. Abra el archivo global style.css y agregue el siguiente estilo CSS:

```

campo-mat-form-field {
    ancho: 100%;
}

```

En el fragmento anterior, configuramos los elementos mat-form-field para que tomen todos los valores disponibles. ancho.

5. Ejecute el comando ng serve para iniciar la aplicación y navegue a <http://localhost:4200/products/new>. Enfóquese en la apariencia de los campos de entrada:



Figura 12.5: Componente de entrada

En la figura anterior, la etiqueta de cada control de formulario lleva un asterisco como sufijo. Este asterisco indica comúnmente que el control de formulario debe tener un valor. Angular Material lo agrega automáticamente al reconocer el atributo requerido en el elemento HTML <input> .

6. Abra el archivo cart.component.ts y repita los pasos 1 y 2, pero no incluya MatError clase.

7. Abra el archivo cart.component.html y modifique el contenido del bloque @for de la siguiente manera:

```
@for(producto de cartForm.controls.products.controls; pista $index) {
  <mat-form-field>
    <mat-label>{{productos[$índice].título}}</mat-label> <entrada
      [formControlName]="$index"
      marcador de posición="{{productos[$index].título}}"
      tipo="número"
      matInput /> </
    mat-form-field>
}
```

El componente restante de nuestra aplicación que contiene un elemento HTML <input> es el componente de detalle del producto. Este componente es un caso especial de una entrada de Angular Material, ya que debemos agruparlo con el botón que cambia el precio del producto:

1. Abra el archivo product-detail.component.ts y modifique la declaración de importación del paquete npm de Angular Material de la siguiente manera:

```
importar { MatButton, MatIconButton } de '@angular/material/button'; importar { MatInput } de
'@angular/material/input';
importar { MatFormField, MatError, MatSuffix } de '@angular/ material/form-field';
importar { MatIcon } de
'@angular/material/icon';
```

2. Agregue las clases importadas anteriores en la matriz de importaciones del decorador @Component :

```
@Component({
  selector: 'aplicación-producto-detalle',
  importaciones: [
    Módulo común,
    Módulo de formularios,
    Directiva de Precio Máximo,
    Botón Mat,
```

```
    MatInput,  
    MatFormField,  
    MatError,  
    MatIcon,  
    MatSuffix,  
    Botón MatIcon  
],  
URL de plantilla: './producto-detalle.componente.html',  
styleUrl: './producto-detalle.componente.css' })
```

3. Abra el archivo product-detail.component.html y modifique el elemento HTML <form> como sigue:

```
<form (ngSubmit)="cambiarPrecio(producto)" #priceForm="ngForm">  
  <mat-form-field>  
    <entrada  
      placeholder="Nuevo precio"  
      type="número"  
      nombre="precio"  
      requerido min="1"  
      Umbral de appPriceMaximum="500"  
      matInput  
      #priceCtrl="ngModel"  
      [(ngModel)]="precio" />  
    <botón  
      botón de ícono de tapete  
      matSuffix  
      tipo="enviar"  
      [deshabilitado]="priceForm.invalid">  
      <mat-icon>editar</mat-icon>  
    </botón>  
    @if (priceCtrl.dirty && (priceCtrl.invalid || priceCtrl.  
      hasError('min'))) {  
      <mat-error>Por favor, introduzca un precio válido</mat-error>  
    } @if (priceCtrl.dirty && priceCtrl.hasError('precioMáximo')) {
```

```

<mat-error>El precio debe ser menor o igual a 500</mat-error>
}
</mat-form-field>
</form>

```

En el fragmento anterior, modificamos el botón que cambia el precio para que muestre un ícono de lápiz y se coloque en línea con el elemento HTML `<input>`.

La directiva `mat-icon-button` indica que el botón no contendrá texto. En su lugar, mostrará un ícono definido por el elemento HTML `<mat-icon>`. La directiva `matSuffix` coloca el botón en línea y al final del elemento HTML `<input>`.

4. Navegue a la lista de productos en el navegador y seleccione uno. La entrada para cambiar el precio del producto debe ser la siguiente:




Figura 12.6: Componente de entrada con botón en línea

En la siguiente sección, aprenderemos cómo usar un componente de selección de material angular para elegir una categoría en el componente de creación de producto.

#### Seleccionar

El componente `select` funciona de forma similar al elemento HTML nativo `<select>`. Muestra un elemento desplegable con una lista de opciones para los usuarios.

Agregaremos uno en el componente de creación de producto para seleccionar la categoría de un nuevo producto:

1. Abra el archivo `product-create.component.ts` y agregue la siguiente declaración de importación :

```
importar { MatSelect, MatOption } desde '@angular/material/select';
```

2. Agregue las clases importadas anteriores en la matriz de importaciones del decorador `@Component` :

```

@Component({
  selector: 'app-product-create',
  importaciones: [
    Módulo de formularios reactivos,
    Botón Mat,
    MatInput,
  ]
})

```

```
    MatFormField,
    MatError,
    MatLabel,
    MatSelect,
    MatOption ],  
  
    URL de plantilla: './product-create.component.html', URL de  
    estilo: './product-create.component.css'  
})
```

3. Abra el archivo product-create.component.html y reemplace el elemento HTML <div> que encierra el elemento <select> con el siguiente fragmento HTML:

```
<mat-form-field>  
  <mat-label>Categoría</mat-label>  
  <mat-select formControlName="categoría">  
    <mat-option value="electronics">Electrónica</mat-option>  
    <mat-option value="jewelery">Joyas</mat-option> <mat-  
    option>Otros</mat-option>  
  </mat-select>  
</mat-form-field>
```

En el fragmento anterior, reemplazamos los elementos HTML <select> y <option> con los elementos <mat-select> y <mat-option> , respectivamente.

4. Vaya a <http://localhost:4200/products/new> y haga clic en el menú desplegable Categoría.

lista descendente:

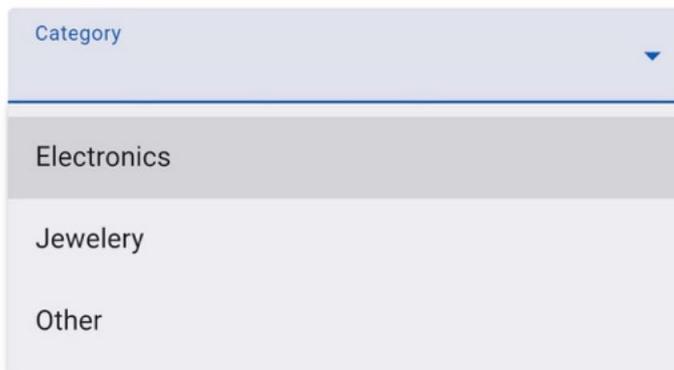


Figura 12.7: Seleccionar componente

El componente de detalles del producto muestra la categoría del producto como un elemento de párrafo con una clase CSS específica. En la siguiente sección, aprenderemos a representar la categoría del producto con el componente de chips de Angular Material.

Papas fritas

El componente chips se utiliza a menudo para mostrar información agrupada por una propiedad específica. También ofrece funciones de filtrado y selección de datos. Podemos usar chips en nuestra aplicación para mostrar la categoría en el componente de detalles del producto.



Nuestros productos solo tienen una categoría, pero los chips tendrían más sentido si tuviéramos categorías adicionales asignadas a nuestros productos.

Empecemos:

1. Abra el archivo product-detail.component.ts y agregue la siguiente declaración de importación :

```
importar { MatChipSet, MatChip } desde '@angular/material/chips';
```

2. Agregue las clases importadas anteriores en la matriz de importaciones del decorador @Component :

```
@Componente({  
    selector: 'aplicación-producto-detalle',  
    importaciones: [  
        Módulo común,  
        Módulo de formularios,  
        Directiva de Precio Máximo,  
        Botón Mat,  
        MatInput,  
        MatFormField,  
        MatError,  
        MatIcon,  
        MatSuffix,  
        Botón MatIcon,  
        Conjunto de chips Mat,  
        MatChip  
    ],  
    URL de plantilla: './producto-detalle.componente.html',  
    styleUrl: './producto-detalle.componente.css'  
})
```

3. Abra el archivo product-detail.component.html y reemplace el elemento HTML <div> que contiene la clase pill-group con el siguiente contenido:

```
<mat-chip-set>
  <mat-chip>{{ producto.categoría }}</mat-chip>
</mat-chip-set>
```

El elemento HTML <mat-chip> indica un componente de chip. Los chips siempre deben estar encerrados en un elemento contenedor. La forma más simple de un contenedor de chips es el elemento <mat-chip-set> .

4. Abra el archivo product-detail.component.css y agregue el siguiente estilo CSS:

```
conjunto de chips mat {
  margen inferior: 1.375rem;
}
```

5. Ejecute el comando ng serve para iniciar la aplicación y seleccionar un producto de la lista.

La categoría debería verse, por ejemplo, como la siguiente:



electronics

Figura 12.8: Componente Chips

El componente chips completa nuestra exploración de los controles de formulario de Angular Material. En la siguiente sección, adquiriremos experiencia práctica al diseñar el diseño de navegación de la aplicación.

## Navegación

Existen diferentes maneras de navegar en una aplicación Angular, como hacer clic en un enlace o en un elemento del menú. Angular Material ofrece los siguientes componentes para este tipo de interacción:

- Menú: una lista emergente donde puede elegir entre un conjunto predefinido de opciones.
- Navegación lateral: Un componente que actúa como un menú acoplado a la izquierda o a la derecha de la página. Puede superponerse a la aplicación, a la vez que atenua su contenido.
- Barra de herramientas: una barra de herramientas estándar que permite al usuario acceder a acciones utilizadas comúnmente.

En esta sección, demostraremos cómo usar el componente de la barra de herramientas. Convertiremos los elementos HTML <header> y <footer> del componente principal de la aplicación a Angular Material.

Para crear una barra de herramientas, seguiremos los siguientes pasos:

1. Abra el archivo app.component.ts y agregue las siguientes declaraciones de importación :

```
importar { MatToolbarRow, MatToolbar } desde '@angular/material/ barra de
herramientas';
importar { MatButton } desde '@angular/material/button';
```

2. Agregue las clases importadas anteriores en la matriz de importaciones del decorador @Component y eliminar la clase RouterLinkActive :

```
@Componente({
  selector: 'app-root',
  importaciones: [
    Salida de enrutador,
    Enlace de enrutador,
    Directiva sobre derechos de autor,
    Componente de autenticación,
    MatToolbarRow,
    MatToolbar,
    Botón Mat
  ],
  URL de plantilla: './app.component.html',
  estiloUrl: './app.component.css' })
```

3. Abra el archivo app.component.html y modifique el elemento HTML <header> de la siguiente manera:

```
<encabezado>
  <mat-toolbar>
    <mat-toolbar-row>
      <h2>{{configuración.título}}</h2>
      <span class="spacer"> <button mat-
        button routerLink="/products">Productos</button>
        Mi carrito
      </span>
      <app-auth></app-auth>
    </mat-toolbar-row>
  </mat-toolbar>
</encabezado>
```

En la plantilla anterior, añadimos los enlaces principales de la aplicación y el componente de autenticación dentro de un elemento `<mat-toolbar>`. El componente de la barra de herramientas consta de una sola fila, indicada por el elemento HTML `<mat-toolbar-row>`.

4. Abra el archivo `app.component.css` y elimine el estilo CSS de la etiqueta de encabezado y la enlaces del menú.
5. Si ejecutamos la aplicación usando el comando `ng serve`, veremos la nueva barra de herramientas de nuestra aplicación en la parte superior de la página:



Figura 12.9: Encabezado de la aplicación

6. Ahora, modifica el elemento HTML `<footer>` para convertirlo en una barra de herramientas de Angular Material. componente:



7. Guarde los cambios, espere a que la aplicación se actualice y observe la barra de herramientas en la parte inferior de la aplicación:

`Copyright ©2024 All Rights Reserved - v1.0`

Figura 12.10: Pie de página de la aplicación

La barra de herramientas es totalmente personalizable y podemos ajustarla según las necesidades de la aplicación. Podemos añadir iconos e incluso crear barras de herramientas con contenido en varias filas. Ahora que conoces los fundamentos de la creación de una barra de herramientas sencilla, puedes explorar más posibilidades.

En la siguiente sección aprenderemos cómo diseñar el contenido de manera diferente dentro de nuestra aplicación.

## Disposición

Cuando hablamos de diseño, hablamos de cómo colocamos el contenido en nuestras plantillas. Angular Material nos proporciona diferentes componentes para este propósito:

- **Lista:** Visualiza el contenido como una lista de elementos. Se puede enriquecer con enlaces e iconos. Incluso multilínea.
- **Lista de cuadrícula:** Nos ayuda a organizar el contenido en bloques. Solo necesitamos definir el número de columnas; el componente llenará el espacio visual.
- **Tarjeta :** Envuelve el contenido y añade una sombra de cuadro. También podemos definirle un encabezado.
- **Pestañas:** divide el contenido en diferentes pestañas.
- **Paso a paso:** divide el contenido en pasos tipo asistente.
- **Panel de expansión:** Nos permite colocar el contenido en forma de lista con un título para cada uno.
- **Artículo.** Los artículos solo se pueden expandir uno a la vez.
- **Tabla:** Representa datos en formato tabular con filas y columnas.

En este libro, cubriremos los componentes de la tarjeta y la mesa.

## Tarjeta

Aprenderemos cómo mostrar cada producto en la lista como una tarjeta:

1. Abra el archivo product.ts y agregue una propiedad de imagen a la interfaz del Producto :

```
Interfaz de exportación Producto {
  id: numero;
  título: cadena;
  precio: número;
  categoría: cadena;
  imagen: cadena;
}
```

La propiedad de imagen es una URL que apunta al archivo de imagen del producto en la API de Fake Store.

2. Abra el archivo product-list.component.ts y agregue la siguiente declaración de importación :

```
importar { MatCardModule } desde '@angular/material/card';
```

3. Agregue la clase MatCardModule en la matriz de importaciones del decorador @Component :

```
@Component({ selector: 'lista-de-productos-de-la-aplicación',
  importaciones: [
    SortPipe,
    Tubería asíncrona,
    Enlace de enrutador,
```

```

    Botón MatMiniFab,
    MatIcon,
    Módulo MatCard
],
templateUrl: './lista-de-productos.componente.html',
styleUrl: './lista-de-productos.componente.css'
})

```



El componente de tarjeta de material de Angular consta de muchos otros componentes y directivas. Optamos por importar el módulo de Angular completo, ya que no sería conveniente importarlos todos individualmente.

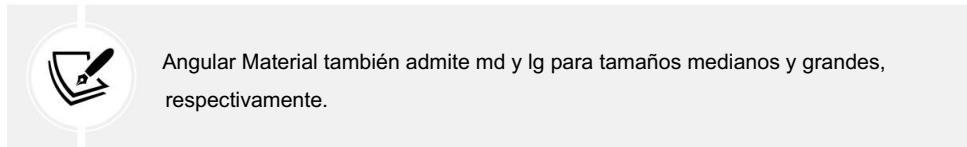
4. Abra el archivo product-list.component.html y reemplace el elemento de lista desordenada con el siguiente fragmento HTML:

```

@for (producto de productos | ordenar; rastrear producto.id) {
  <mat-card [enlace del enrutador]=[id del producto]>
    <mat-card-header>
      <mat-card-title-group>
        <mat-card-title>{{ producto.título }}</mat-card-title>
        <mat-card-subtitle>{{ producto.categoría }}</mat-card-
        subtítulo>
        <img mat-card-sm-image [src]="producto.imagen" />
        Grupo de títulos de tarjetas de tapete
        encabezado de tarjeta mat
    </mat-card-header>
    </mat-card>
} @vacío {
  <p>¡No se encontraron productos!</p>
}

```

Un componente de tarjeta de Angular Material consta de un encabezado, indicado por el elemento HTML `<mat-card-header>`. El componente de encabezado contiene un `<mat-card-title-group>`. Elemento HTML que alinea el título, el subtítulo y la imagen de la tarjeta en una sola sección. El título de la tarjeta, indicado por el elemento HTML `<mat-card-title>`, muestra el título del producto. El subtítulo de la tarjeta, indicado por el elemento HTML `<mat-card-subtitle>`, muestra la categoría del producto. Finalmente, la imagen del producto se muestra adjuntando la imagen `mat-card-sm-image`. Directiva para un elemento HTML `<img>`. La palabra clave `sm` en la directiva indica que queremos renderizar una imagen de tamaño pequeño.



Angular Material también admite md y lg para tamaños medianos y grandes, respectivamente.

5. Abra el archivo product-list.component.css y agregue el siguiente estilo CSS:

```
tarjeta de tapete {  
    margen: 1.375rem;  
    cursor: puntero;  
}
```

6. Ejecute la aplicación usando el comando ng serve y navegue a <http://localhost:4200>:

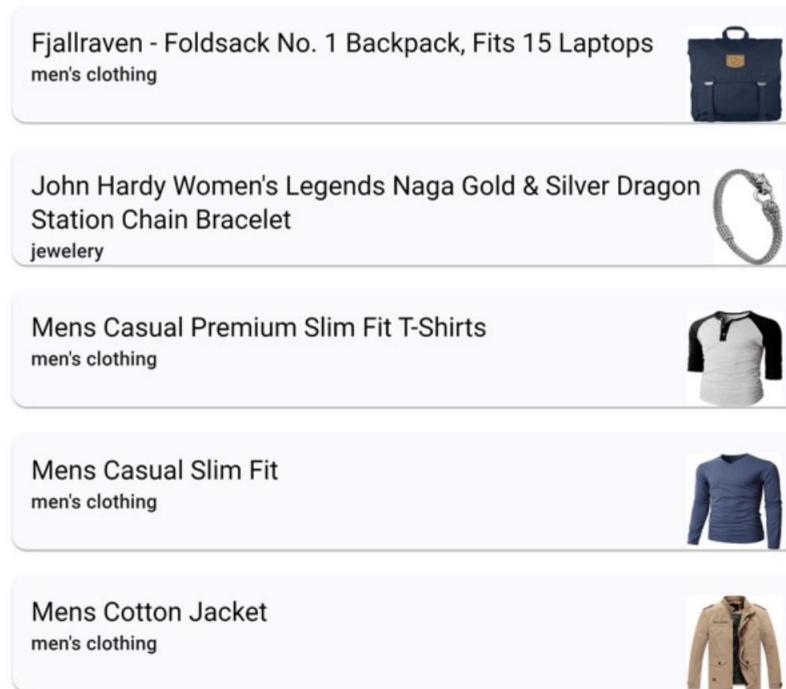


Figura 12.11: Representación de la tarjeta de lista de productos

Puede explorar más opciones para el componente de tarjeta navegando a <https://material.angular.io/componentes/tarjeta/descripción general>.

En la siguiente sección, aprenderemos cómo cambiar la lista de productos a una vista tabular.

## Tabla de datos

El componente de tabla de la biblioteca Angular Material nos permite mostrar nuestros datos en columnas y filas.

Para crear una tabla, debemos importar la clase MatTableModule del espacio de nombres `@angular/material/table`.



La tabla de datos de material angular consta de muchos otros componentes y directivas. Elegimos importar todo el módulo Angular porque no sería conveniente importarlos todos individualmente.

Empecemos:

1. Abra el archivo `product-list.component.ts` e importe `CurrencyPipe` y el

Artefactos de `MatTableModule`:

```
importar { AsyncPipe, CurrencyPipe } desde '@angular/common';
importar { MatTableModule } desde '@angular/material/table';
```

2. Agregue las clases importadas anteriores a la matriz de importaciones del decorador `@Component`:

```
@Component({
  selector: 'app-product-list',
  imports: [
    SortPipe,
    Tubería asíncrona,
    CurrencyPipe,
    Enlace de enrutador,
    Botón MatMiniFab,
    MatIcon,
    Módulo MatCard,
    Módulo de tabla de tapetes
  ],
  templateUrl: './lista-de-productos.component.html',
  styleUrls: ['./product-list.component.css']
})
```

3. Cree la siguiente propiedad en la clase `ProductListComponent` para definir la tabla

nombres de columnas:

```
columnNames = ['título', 'precio'];
```

El nombre de cada columna coincide con una propiedad de la interfaz del Producto .

4. Abra el archivo product-list.component.html y agregue el siguiente fragmento después del

@para bloque:

```
<table mat-table [dataSource]="productos"></table>
```

Una tabla de material angular es un elemento HTML `<table>` estándar con la directiva `mat-table` adjunta.

La propiedad `dataSource` de la directiva `mat-table` define los datos que queremos mostrar en la tabla. Puede ser cualquier dato enumerable, como un array. En nuestro caso, la vinculamos a la variable de referencia de la plantilla " products" .

5. Agregue un elemento `<ng-container>` para cada columna que queremos mostrar:

```
<table mat-table [dataSource]="productos">
  <ng-contenedor matColumnDef="título">
    <th mat-header-cell *matHeaderCellDef>Título</th>
  <td mat-cell *matCellDef="dejar producto">
    <a [routerLink]=[product.id]">{{ producto.título }}</a>
  </td>
</ng-contenedor>
<ng-contenedor matColumnDef="precio">
  Precio
  <td mat-cell *matCellDef="let producto">{{ producto.precio | moneda }}</td>
</ng-contenedor>
</table>
```



El elemento `<ng-container>` es un elemento de propósito único que agrupa elementos con funcionalidad similar. No interfiere con el estilo de los elementos secundarios ni se muestra en pantalla.

El elemento `<ng-container>` utiliza la directiva `matColumnDef` para establecer el nombre de la columna específica.



El valor de la directiva `matColumnDef` debe coincidir con un valor de la

Propiedad del componente `columnNames` ; de lo contrario, la aplicación generará un error que indica que no puede encontrar el nombre de la columna definida.

Contiene un elemento HTML <th> con una directiva mat-header-cell que indica el encabezado de la celda y un elemento HTML <td> con una directiva mat-cell para los datos de la celda. El elemento HTML <td> usa la directiva matCellDef para crear una variable de plantilla local para los datos de la fila actual que podemos usar más adelante.

6. Agregue el siguiente fragmento después de los elementos <ng-container> :

```
<tr mat-header-row *matHeaderRowDef="nombresDeColumna"></tr>
<tr mat-row *matRowDef="let row; columns: nombresDeColumna;"></tr>
```

En el fragmento anterior, definimos la fila de encabezado de la tabla que muestra los nombres de las columnas y las filas reales que contienen datos.

Si ejecutamos la aplicación, la salida debería ser la siguiente:

Title	Price
Fjallraven - Foldsack No. 1 Backpack, Fits 15 Laptops	\$109.95
John Hardy Women's Legends Naga Gold & Silver Dragon Station Chain Bracelet	\$695.00
Mens Casual Premium Slim Fit T-Shirts	\$22.30
Mens Casual Slim Fit	\$15.99
Mens Cotton Jacket	\$55.99
Pierced Owl Rose Gold Plated Stainless Steel Double	\$10.99
SanDisk SSD PLUS 1TB Internal SSD - SATA III 6 Gb/s	\$109.00
Solid Gold Petite Micropave	\$168.00
WD 2TB Elements Portable External Hard Drive - USB 3.0	\$64.00
White Gold Plated Princess	\$9.99

Figura 12.12: Componente de tabla

El componente de lista de productos muestra la representación de datos en tarjeta y en tabla simultáneamente.

Usaremos el componente de alternancia de botón de Angular Material para distinguirlos.

El componente de alternancia de botones activa o desactiva los botones según una condición específica:

1. Abra el archivo product-list.component.ts y agregue la siguiente declaración de importación :

```
importar { MatButtonToggle, MatButtonToggleGroup } de '@angular/ material/button-
toggle';
```

2. Agregue las clases importadas anteriores en la matriz de importaciones del decorador @Component :

```
@Component({ selector: 'app-product-
list', importaciones: [
  SortPipe,
  Tubería asíncrona,
  CurrencyPipe,
  Enlace de enrutador,
  Botón MatMiniFab,
  MatIcon,
  Módulo MatCard,
  MatTableModule,
  Botón de tapete para alternar,
  MatButtonToggleGroup ],
  templateUrl: './product-list.component.html', styleUrls: './
product-list.component.css' })
```

3. Abra el archivo product-list.component.html y agregue el siguiente fragmento HTML dentro el elemento HTML <div> con la clase caption :

```
<span class="spacer">
<mat-button-toggle-group #group="matButtonToggleGroup">
  <mat-button-toggle value="tarjeta" marcada>
    <mat-icon>lista</mat-icon>
  </mat-button-toggle>
  <mat-button-toggle value="tabla">
    <mat-icon>cuadrícula activada</mat-icon>
  </mat-button-toggle> </
  mat-button-toggle-group>
```

En el fragmento anterior, usamos el elemento `<mat-button-toggle-group>` para crear dos botones de alternancia uno junto al otro. La instancia del grupo de botones de alternancia se asigna a la variable de referencia de la plantilla de grupo para que podamos acceder a ella posteriormente.

Declaramos los botones de alternancia mediante el elemento `<mat-button-toggle>` y asignamos un valor adecuado . La propiedad `value` se establecerá al hacer clic en cualquiera de los botones. También tenemos un ícono para cada botón de alternancia para mejorar la experiencia de usuario mientras los usuarios interactúan con la lista de productos.

4. Cree un nuevo bloque `@if` después del elemento HTML `<div>` con la clase `caption` y muévalo el bloque `@for` dentro de él:

```
@if (grupo.valor === 'tarjeta') {
  @for (producto of productos | ordenar; rastrear producto.id) {
    <mat-card [enlace del enrutador]=[id del producto]>
      <mat-card-header>
        <mat-card-title-group>
          <mat-card-title>{{ producto.título }}</mat-card-title>
          <mat-card-subtitle>{{ producto.categoría }}</mat-card-
        subtítulo>
          <img mat-card-sm-image [src]="producto.imagen" />
        Grupo de títulos de tarjetas de tapete
        encabezado de tarjeta mat
      </mat-card>
    } @vacío {
      <p>No se encontraron productos!</p>
    }
  }
}
```

De acuerdo con el fragmento anterior, la representación de la tarjeta de los productos se mostrará cuando la propiedad de valor del grupo de alternancia de botones se establezca en tarjeta.

5. Agregue el siguiente bloque `@else` y mueva el componente de tabla de datos dentro de él para mostrar la lista de productos en formato tabular cuando se haga clic en el segundo botón de alternancia:

```
@demás {
  <table mat-table [dataSource]="productos">
    <ng-contenedor matColumnDef="título">
      <th mat-header-cell *matHeaderCellDef>Título</th>
    <td mat-cell *matCellDef="dejar producto">
```

```

<a [routerLink]=[“product.id”]> {{ producto.título }}</a> </td>

</ng-container>
<ng-container matColumnDef="precio">
  Precio
  <td mat-cell *matCellDef="let producto">{{ producto.precio | moneda }}</td> </ng-
  container> <tr mat-
  header-row *matHeaderRowDef="columnNames"></tr>
  <tr mat-row *matRowDef="let row; columns: nombresDeColumna;"></tr> </table>

}

```

6. Ejecute el comando ng serve para iniciar la aplicación y verificar que la representación de la tarjeta se muestra inicialmente.

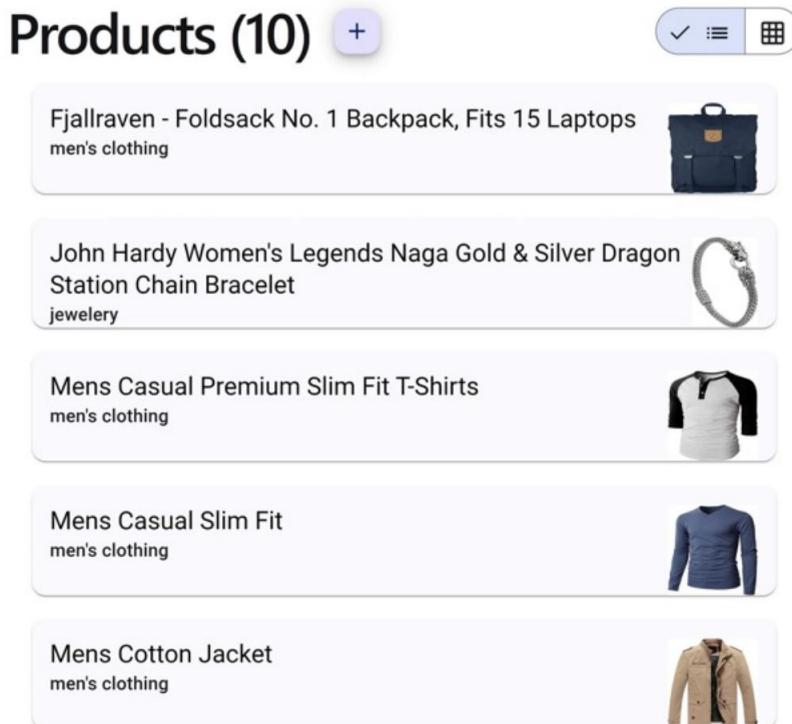


Figura 12.13: Lista de productos

7. Haga clic en el segundo botón de alternancia y verifique que los productos ahora se muestran en forma de tabla.  
formato.

En esta sección, aprendimos a mostrar la lista de productos en formato tabular. También usamos la función de alternancia. botones para cambiar de la vista de tarjeta a la vista tabular.

En la siguiente sección, aprenderemos cómo utilizar ventanas emergentes y superposiciones para proporcionar información adicional a los usuarios.

### Ventanas emergentes y superposiciones

Existen diferentes maneras de captar la atención del usuario en una aplicación web. Una de ellas es mostrar un cuadro de diálogo emergente sobre el contenido de la página e incitar al usuario a actuar en consecuencia. Otra forma es mostrar información como una notificación en diferentes partes de la página.

Angular Material ofrece tres componentes diferentes para manejar estos casos:

- Diálogo: un cuadro de diálogo emergente modal que se muestra en la parte superior del contenido de la página.
- Insignia: una pequeña indicación en un círculo para actualizar el estado de un elemento de la interfaz de usuario.
- Snackbar: Mensaje informativo que se muestra en la parte inferior de una página y que es visible durante un breve periodo de tiempo.

Su propósito es notificar al usuario el resultado de una acción, como guardar un formulario.

Aprenderemos cómo utilizar los componentes anteriores en nuestra aplicación de tienda electrónica, comenzando con cómo crear un cuadro de diálogo modal simple.

### Creación de un cuadro de diálogo de confirmación

El componente de diálogo es bastante potente y se puede personalizar y configurar fácilmente. Es un componente Angular común con directivas personalizadas que lo obligan a comportarse como un diálogo. Para explorar las capacidades del diálogo de Angular Material, usaremos un diálogo de confirmación en la protección de pago para notificar a los usuarios sobre los artículos restantes en sus carritos de compra:

1. Ejecute el siguiente comando CLI de Angular para crear un nuevo componente Angular:

```
ng generar pago de componentes
```

El comando anterior creará un componente Angular que albergará nuestro diálogo.

2. Abra el archivo checkout.component.ts y agregue las siguientes declaraciones de importación :

```
importar { MatButton } desde '@angular/material/button';
importar { MatDialogModule } desde '@angular/material/dialog';
```



El componente de diálogo Material de Angular consta de muchos otros componentes y directivas. Optamos por importar el módulo Angular completo, ya que no sería conveniente importarlos todos individualmente.

3. Agregue las clases importadas anteriores en la matriz de importaciones del decorador @Component :

```
@Component({
  selector: 'app-checkout',
  imports: [MatButton, MatDialogModule],
  templateUrl: './checkout.component.html',
  styleUrls: ['./checkout.component.css']
})
```

4. Abra el archivo checkout.component.html y reemplace su contenido con el siguiente HTML plantilla:

```
<h1 mat-dialog-title> Finalizar compra</h1>
<mat-dialog-content>
  Tienes artículos pendientes en tu carrito. ¿Quieres continuar?

</mat-dialog-content>
<mat-dialog-actions>
  Sí
  <button mat-button>No</button>
</mat-dialog-actions>
```

La plantilla de componente contiene varias directivas y elementos pertenecientes al componente de diálogo Material Angular. La directiva mat-dialog-title define el título del diálogo, y <mat-dialog-content> representa el contenido real. <mat-dialog-actions> El elemento define las acciones que puede realizar el cuadro de diálogo y generalmente envuelve los elementos del botón.

5. Se debe activar un cuadro de diálogo para que se muestre en una página. Abra el archivo checkout.guard.ts y agregue las siguientes instrucciones de importación :

```
importar { MatDialog } desde '@angular/material/dialog';
importar { CheckoutComponent } desde './checkout/checkout.component';
```

6. Inyecte el servicio MatDialog en el cuerpo de la función checkoutGuard :

```
const dialog = inyectar(MatDialog);
```

7. Modifique la asignación de la variable de confirmación de la siguiente manera:

```
si (cartService.cart) {  
  const confirmación = diálogo.open(CheckoutComponent).afterClosed();  
  confirmación de devolución ;  
}
```

En el fragmento anterior, usamos el servicio MatDialog para mostrar el componente de pago. El servicio MatDialog acepta el tipo de clase de componente que representa el cuadro de diálogo. como parámetro.

El método open del servicio MatDialog devuelve una propiedad observable "afterClosed" , que nos notificará cuando se cierre el diálogo. Esta propiedad observable emite cualquier valor que se envíe desde el diálogo.



Más adelante en el capítulo, aprenderemos cómo devolver un valor booleano del componente de diálogo que coincide con el tipo devuelto por la función CanDeactivateFn .

Ahora podemos verificar que el componente de diálogo funciona como se espera ejecutando los siguientes pasos:

1. Ejecute la aplicación usando el comando ng serve y navegue a <http://localhost:4200>.
2. Inicie sesión en la aplicación.
3. Seleccione un producto de la lista y agréguelo al carrito de compras.
4. Repita el paso anterior para agregar más productos al carrito.
5. Vaya al carrito de compras y haga clic en el botón Atrás del navegador o en cualquier enlace de la aplicación para salir del carrito. Se mostrará el siguiente cuadro de diálogo en la pantalla:

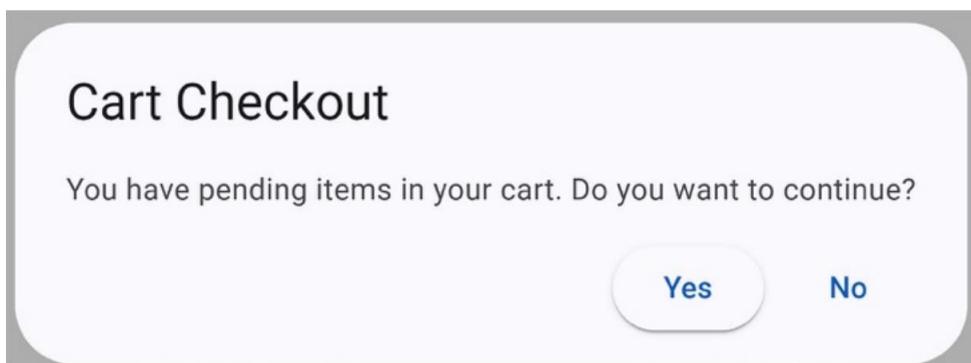


Figura 12.14: Componente de diálogo de pago

Podríamos mejorar aún más la experiencia de usuario (UX) de la aplicación mostrando información en el cuadro de diálogo sobre la cantidad de artículos añadidos al carrito. En la siguiente sección, aprenderemos a pasar datos en el cuadro de diálogo y mostrar la cantidad de artículos del carrito.

## Configuración de diálogos

En un escenario real, probablemente necesitará crear un componente reutilizable para mostrar un diálogo en un proyecto Angular. El componente podría estar incluido en una biblioteca Angular como paquete. Por lo tanto, debería configurar el componente de diálogo para que acepte datos dinámicamente.

En el proyecto Angular actual, nos gustaría mostrar la cantidad de productos que hemos agregado al carrito de compras:

1. Abra el archivo checkout.component.ts y modifique las declaraciones de importación de la siguiente manera:

```
importar { Componente, inyectar } desde '@angular/core';
importar { MatButton } desde '@angular/material/button';
importar { MatDialogModule, MAT_DIALOG_DATA } desde '@angular/material/
diálogo';
```

2. Inyecte MAT\_DIALOG\_DATA en la clase CheckoutComponent de la siguiente manera:

```
clase de exportación CheckoutComponent {
  datos = inyectar(MAT_DIALOG_DATA);
}
```

MAT\_DIALOG\_DATA es un token de inyección que nos permite pasar datos arbitrarios al componente de diálogo. La variable de datos contendrá cualquier dato que pasemos al diálogo al llamar a su método de apertura .

3. Abra el archivo checkout.component.html y agregue la propiedad de datos al texto interno de el elemento HTML <span> :

```
<span>
  Tienes {{ data }} artículos pendientes en tu carrito.
  ¿Quieres continuar?
```

4. Abra el archivo checkout.guard.ts y configure la propiedad de datos en la configuración del cuadro de diálogo objeto, que es el segundo parámetro del método abierto :

```
const confirmación = diálogo.open(
  Componente de pago,
```

```
{ datos: cartService.cart.products.length }  
.afterClosed();
```

5. Si intentamos salir de la página del carrito mientras ejecutamos la aplicación, obtendremos un diálogo similar a lo siguiente:

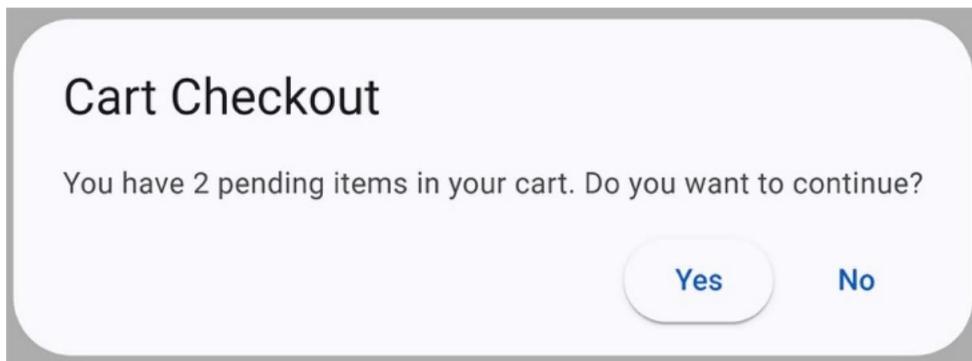


Figura 12.15: Componente de diálogo de pago con datos personalizados

Los botones del componente de diálogo aún no tienen ninguna función específica. En la siguiente sección, aprenderemos a configurarlos y a devolver datos al guardia.

## Obtener datos de los diálogos

El módulo de diálogo Material de Angular expone la directiva mat-dialog-close , que permite configurar el botón que cerrará el diálogo. Abra el archivo checkout.component.html y agregue la directiva mat-dialog-close a ambos botones:

```
<mat-dialog-actions>  
  Sí  
  <button mat-button [mat-dialog-close]="false">No</button>  
</mat-dialog-actions>
```

En el fragmento anterior, utilizamos la directiva mat-dialog-close de dos maneras:

- Sin pasar un valor en el botón Sí , el cuadro de diálogo devolverá verdadero como valor predeterminado, lo que permitirá al guardia salir de la página del carrito de compras.
- Con la vinculación de propiedades en el botón No , pasamos falso como valor para cancelar la navegación del guardia.

Ejecute los siguientes pasos para verificar que el comportamiento del diálogo sea correcto:

1. Ejecute el comando `ng serve` para iniciar la aplicación y navegue a `http://localhost:4200`.
2. Inicie sesión en la aplicación.
3. Seleccione un producto de la lista y agréguelo al carrito.
4. Haga clic en el enlace `Mi carrito` para navegar al carrito de compras.
5. Haga clic en el enlace `Productos`, seleccione `No` en el cuadro de diálogo de pago y verifique que la aplicación permanece en la página del carrito de compras.
6. Haga clic nuevamente en el enlace `Productos`, seleccione `Sí` en el cuadro de diálogo y debería navegar hasta el lista de productos.

Los diálogos son una excelente característica de Angular Material que ofrece potentes funciones a tus aplicaciones. En la siguiente sección, exploraremos los componentes de insignia y barra de notificaciones para notificar al usuario cuando se añade un producto al carrito de compra.

## Visualización de notificaciones de usuario

La biblioteca Angular Material aplica patrones y comportamientos que mejoran la experiencia de usuario (UX) de la aplicación.

Un aspecto de la experiencia de usuario de la aplicación se refiere a proporcionar notificaciones a los usuarios sobre acciones específicas.

Angular Material nos proporciona la insignia y los componentes `snackbar` que podemos utilizar en este caso.

## Aplicación de insignias

El componente insignia es un círculo colocado sobre otro elemento y suele mostrar un número. Aprenderemos a aplicar insignias mostrando el número de artículos del carrito de compra en el enlace de la aplicación `Mi Carrito`:

1. Abra el archivo `app.component.ts` y agregue las siguientes declaraciones de importación :

```
importar { MatBadge } de '@angular/material/badge';
importar { CartService } desde './cart.service';
```

La clase `MatBadge` exporta el componente de insignia. La clase `CartService` nos proporcionará el número de artículos en el carrito de compras.

2. Agregue la clase `MatBadge` en la matriz de importaciones del decorador `@Component` :

```
@Component({
  selector: 'app-root',
  importaciones: [
    Salida de enrutador,
```

```

    Enlace de enrutador,
    Directiva sobre derechos de autor,
    Componente de autenticación,
    MatToolbarRow,
    MatToolbar,
    Botón Mat,
    MatBadge
],
URL de plantilla: './app.component.html',
URL de estilo: './app.component.css'
})

```

3. Inyecte la clase CartService en la clase AppComponent :

```
cartService = injectar(CartService);
```

4. Abra el archivo app.component.html y agregue la directiva matBadge al botón Mi carrito :

```

<botón
    botón de tapete
    routerLink="/carrito"
    [matBadge]="cartService.cart?.products?.length">
    Mi carrito
</botón>

```

En el fragmento anterior, la directiva matBadge indica el número que se muestra en la insignia. En este caso, la asociamos con la longitud del array de productos del carrito de compra actual.

5. Abra el archivo app.component.css y agregue el siguiente estilo CSS:

```

botón {
    margen: 5px;
}

```

El estilo anterior agregará espacio alrededor de cada enlace de la aplicación para que los botones no se superpongan con el componente de insignia.

6. Ejecute el comando ng serve para iniciar la aplicación y añadir algunos productos al carrito de compras.

Observe que el ícono de la insignia actualiza su valor cuando se añaden productos al carrito; a continuación, un ejemplo:

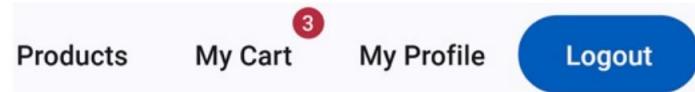


Figura 12.16: Componente de insignia

Aplicación de una barra de snacks. Otro buen patrón de UX al trabajar con aplicaciones CRUD es mostrar una notificación al completar una acción. Podemos aplicar este patrón mostrando una notificación al añadir un producto al carrito de compra. Usaremos el componente de barra de snacks de Angular. Material para mostrar la notificación:

1. Abra el archivo product-detail.component.ts y agregue la siguiente declaración de importación :

```
importar { MatSnackBarModule, MatSnackBar } de '@angular/material/ snack-bar';
```

Snackbar no es un componente Angular como todos los componentes Angular Material que hemos visto. Es un servicio Angular llamado MatSnackBar y se puede usar importando la clase MatSnackBarModule a nuestros componentes.

2. Agregue la clase MatSnackBarModule en la matriz de importaciones del decorador @Component :

```
@Componente(  
  selector: 'app-product-detail', importa: [  
  
    Módulo común,  
    Módulo de formularios,  
    Directiva de Precio Máximo,  
    Botón Mat,  
    MatInput,  
    MatFormField,  
    MatError,  
    MatIcon,  
    MatSuffix,  
    Botón MatIcon,  
    Conjunto de chips Mat,  
    MatChip,  
    Módulo MatSnackBar  
,  
,
```

```

    URL de plantilla: './producto-detalle.componente.html',
    styleUrls: ['./producto-detalle.componente.css'
  })
}

```

3. Inyecte el servicio MatSnackBar en el constructor de la clase ProductDetailComponent :

```

constructor(
  productoServicio privado : ProductosServicio,
  servicio de autenticación público : AuthService,
  ruta privada : ActivatedRoute,
  enrutador privado : enrutador,
  carrito privadoService : CartService,
  MatSnackBar privado: MatSnackBar
) {}

```

4. Modifique el método addToCart para mostrar una barra de snack cuando se agregue el producto al carrito:

```

addToCart(id: número) {
  este.cartService.addProduct(id).subscribe(() => {
    this.snackBar.open('¡Producto añadido al carrito!', undefined, {
      duración: 1000
    });
  });
}

```

En el método anterior, usamos el método abierto del servicio MatSnackBar para mostrar una barra de snacks. El método abierto acepta tres parámetros: el mensaje que queremos mostrar, la acción que queremos realizar al cerrar la barra de snacks y un objeto de configuración.

El objeto de configuración nos permite establecer varias opciones, como la duración en la que La barra de snacks será visible en milisegundos.

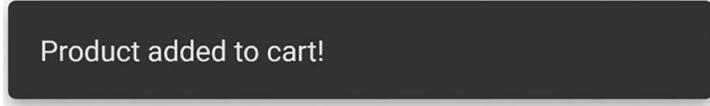


No pasamos un parámetro para la acción porque no queremos reaccionar Cuando se despidie el snack-bar.

5. Ejecute el comando ng serve para iniciar la aplicación y seleccionar un producto de la lista.

6. Asegúrate de haber iniciado sesión y haz clic en el botón "Añadir al carrito" . Aparecerá la siguiente notificación.

El mensaje se mostrará en la parte inferior de la página:



Product added to cart!

Figura 12.17: Componente Snackbar



La posición de la barra de snacks se puede cambiar desde las opciones de configuración. Más información en <https://material.angular.io/components/snack-bar/overview>.

En esta sección, aprendimos a utilizar modelos emergentes y superposiciones de notificaciones para mejorar la experiencia de usuario de la aplicación y brindar un excelente flujo de trabajo a nuestros usuarios.

## Resumen

En este capítulo, analizamos los fundamentos del sistema Material Design. Nos centramos principalmente en Angular Material, la implementación de Material Design diseñada para Angular, y en cómo se compone de diferentes componentes. Analizamos una explicación práctica sobre cómo instalarlo, configurarlo y usar algunos de sus componentes y temas principales.

Con suerte, habrá leído este capítulo y habrá descubierto que ahora comprende Material Design en general y Angular Material en particular y puede determinar si es una buena opción para su próxima aplicación Angular.

Las aplicaciones web deben ser testeables para garantizar su funcionalidad y cumplimiento de los requisitos de la aplicación. En el siguiente capítulo, aprenderemos a aplicar diferentes técnicas de prueba en aplicaciones Angular.

Machine Translated by Google

# 13

## Pruebas unitarias angulares Aplicaciones

En los capítulos anteriores, analizamos muchos aspectos de cómo crear una aplicación empresarial Angular desde cero. Pero ¿cómo podemos garantizar que una aplicación pueda mantenerse en el futuro sin mayores complicaciones? Una capa integral de pruebas automatizadas puede ser nuestro salvavidas una vez que nuestra aplicación comience a escalar y tengamos que mitigar el impacto de los errores.

Las pruebas, en concreto las unitarias, deben ser realizadas por el desarrollador durante el desarrollo del proyecto. Ahora que conocemos bien el framework, en este capítulo abordaremos brevemente todos los detalles de las pruebas unitarias en una aplicación Angular, incluyendo el uso de herramientas de prueba.



Para simplificar, los ejemplos de este capítulo no están relacionados con la aplicación de tienda electrónica que hemos creado a lo largo del libro.

Con más detalle aprenderemos lo siguiente:

- ¿Por qué necesitamos pruebas unitarias?
- La anatomía de una prueba unitaria
- Introducción a las pruebas unitarias en Angular
- Componentes de prueba
- Servicios de pruebas

- Prueba de tuberías
- Directivas de prueba
- Formularios de prueba
- Probar el enrutador

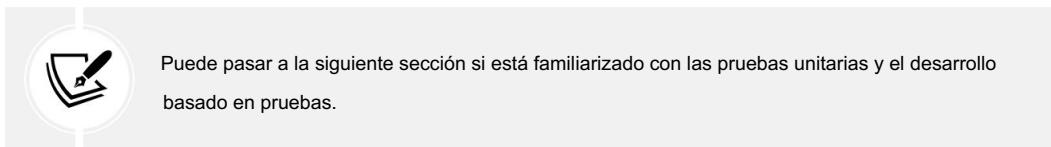
## Requisitos técnicos

Este capítulo contiene varios ejemplos de código que te guiarán a través del concepto de pruebas unitarias en Angular. Puedes encontrar el código fuente relacionado en la carpeta ch13 del siguiente repositorio de GitHub:

<https://www.github.com/PacktPublishing/Aprendizaje-Angular-Quinta-Edición>.

### ¿Por qué necesitamos pruebas unitarias?

En esta sección, aprenderemos qué son las pruebas unitarias y por qué son útiles en el desarrollo web.



Puede pasar a la siguiente sección si está familiarizado con las pruebas unitarias y el desarrollo basado en pruebas.

Las pruebas unitarias son parte de una filosofía de ingeniería para procesos de desarrollo eficientes y ágiles.

Añaden una capa de pruebas automatizadas al código de la aplicación antes de su desarrollo. El concepto central es que cada fragmento de código va acompañado de su prueba, ambas creadas por el desarrollador que trabaja en ese código.

Primero, diseñamos la prueba con la función que queremos ofrecer, verificando la precisión de su resultado y comportamiento.

Dado que la función aún no está implementada, la prueba fallará, por lo que el trabajo del desarrollador es crear la función para que la supere.

Las pruebas unitarias son bastante controvertidas. Si bien el desarrollo basado en pruebas es beneficioso para garantizar la calidad y el mantenimiento del código a lo largo del tiempo, no todos las realizan en su flujo de trabajo diario.

Desarrollar pruebas a medida que desarrollamos nuestro código a veces puede resultar una carga, especialmente cuando los resultados superan la funcionalidad que se pretende probar. Sin embargo, los argumentos a favor superan a los argumentos en contra:

- Desarrollar pruebas contribuye a un mejor diseño de código. Nuestro código debe cumplir con los requisitos de la prueba, y no al revés. Si intentamos probar un código existente y nos bloqueamos en algún punto, es probable que el código no esté bien diseñado y requiera una revisión. Por otro lado, desarrollar características comprobables puede ayudar a detectar tempranamente efectos secundarios.

Refactorizar el código probado es fundamental para evitar la introducción de errores en etapas posteriores. El desarrollo está diseñado para evolucionar con el tiempo, y el riesgo de introducir un error con cada refactorización es alto.

Las pruebas unitarias son una excelente manera de garantizar que detectemos errores de forma temprana, ya sea al introducir nuevas características o actualizar las existentes.

- Desarrollar pruebas es una excelente manera de documentar nuestro código. Se convierte en un recurso invaluable cuando alguien que no está familiarizado con el código base se encarga del desarrollo.

Estos son solo algunos argumentos, pero puedes encontrar innumerables recursos en la web sobre los beneficios de probar tu código. Si aún no te convence, inténtalo; de lo contrario, continuemos con nuestro recorrido y veamos el formato general de una prueba unitaria.

## La anatomía de una prueba unitaria

Hay muchas maneras diferentes de probar un fragmento de código. En este capítulo, analizaremos la estructura de una prueba unitaria: las distintas partes que la componen.

Para probar cualquier código, necesitamos un marco de trabajo para escribir la prueba y un ejecutor para ejecutarla. En esta sección, nos centraremos en el marco de trabajo de pruebas. Este marco de trabajo debe proporcionar funciones de utilidad para crear conjuntos de pruebas que contengan una o varias especificaciones de prueba. Por lo tanto, las pruebas unitarias abarcan los siguientes conceptos:

- Conjunto de pruebas: Un conjunto que crea una agrupación lógica para muchas pruebas. Un conjunto, por ejemplo, puede Contiene todas las pruebas para una característica específica.
- Especificación de prueba: la prueba unitaria real.

En este capítulo, usaremos Jasmine , un popular framework de pruebas que también se usa por defecto en proyectos de Angular CLI. Así es como se ve una prueba unitaria en Jasmine:

```
describe('Calculadora', () => {
  it('debería sumar dos números', () => {
    esperar(1+1).toBe(2);
  });
});
```

El método describe define el conjunto de pruebas y acepta un nombre y una función de flecha como parámetros . La función de flecha es el cuerpo del conjunto de pruebas y contiene varias pruebas unitarias . El método define una sola prueba unitaria. Acepta un nombre y una función de flecha como parámetros.

Cada especificación de prueba valida una funcionalidad específica de la característica descrita en el nombre de la suite y declara una o varias expectativas en su cuerpo. Cada expectativa toma un valor, llamado valor esperado , que se compara con un valor real mediante una función de comparación . Esta función comprueba si los valores esperados y reales coinciden, lo que se denomina aserción. El marco de pruebas aprueba o no la especificación según el resultado de dichas aserciones. En el ejemplo anterior, `1+1` devolverá el valor real que se supone que coincide con el valor esperado, `2`, declarado en la función de comparación `toBe` .



El marco Jasmine contiene varias funciones de comparación según las necesidades específicas del usuario, como veremos más adelante en el capítulo.

Supongamos que el código anterior contiene otra operación matemática que debe probarse. Sería conveniente agrupar ambas operaciones en la suite Calculadora , como se indica a continuación:

```
describe('Calculadora', () => {
  it('debería sumar dos números', () => {
    esperar(1+1).toBe(2);
  });

  it('debería restar dos números', () => {
    esperar(1-1).toBe(0);
  });
});
```

Hasta ahora, hemos aprendido sobre las suites de pruebas y cómo usarlas para agrupar las pruebas según su funcionalidad. Además, hemos aprendido a invocar el código que queremos probar y a confirmar que funciona correctamente. Sin embargo, las pruebas unitarias implican más conceptos que vale la pena conocer, como las funcionalidades de configuración y desmontaje .

Una función de configuración prepara el código antes de empezar a ejecutar las pruebas. Es una forma de mantener el código limpio, centrándose en invocarlo y comprobar las aserciones. Una función de desmontaje es lo opuesto. Se encarga de desmantelar la configuración inicial, lo que implica actividades como la limpieza de recursos. Veamos cómo funciona esto en la práctica con un ejemplo de código:

```
describe('Calculadora', () => {
  dejar total: numero;

  antes de cada(() => total = 1);
```

```
it('debería sumar dos números', () => {
    total = total + 1;
    esperar(total).toBe(2);
});

it('debería restar dos números', () => {
    total = total - 1;
    esperar(total).toBe(0);
});

después de cada(() => total = 0);
});
```

El método `beforeEach` se utiliza para la configuración y se ejecuta antes de cada prueba unitaria. En este ejemplo, establecemos el valor de la variable `total` en 1 antes de cada prueba. El método `afterEach` se utiliza para ejecutar la lógica de desmontaje. Después de cada prueba, restablecemos el valor de la variable `total` a 0.

Es evidente que la prueba solo tiene que preocuparse por invocar el código de la aplicación y confirmar el resultado, lo que hace que las pruebas sean más limpias; sin embargo, las pruebas tienden a tener mucha más configuración en una aplicación real. Más importante aún, el método `beforeEach` facilita la adición de nuevas pruebas, lo cual es excelente. Queremos un código bien probado; cuanto más fácil sea escribir y mantener dicho código, mejor para nuestro software.

Ahora que hemos cubierto los conceptos básicos de una prueba unitaria, veamos cómo podemos implementarlos en el contexto del marco Angular.

## Introducción a las pruebas unitarias en Angular

En la sección anterior, nos familiarizamos con las pruebas unitarias y sus conceptos generales, como suites de pruebas, especificaciones de prueba y aserciones. Con este conocimiento, es hora de adentrarnos en las pruebas unitarias con Angular. Antes de empezar a escribir pruebas para Angular, veamos las herramientas que nos ofrecen el framework Angular y la CLI de Angular:

- Jazmín: Ya hemos aprendido que este es el marco de pruebas.
- Karma: el ejecutor de pruebas para ejecutar nuestras pruebas unitarias.
- Utilidades de pruebas angulares: un conjunto de métodos auxiliares que nos ayudan a configurar nuestras pruebas unitarias y escribir nuestras afirmaciones en el contexto del marco Angular.



Al usar la CLI de Angular, no es necesario configurar Jasmine ni Karma en una aplicación Angular. Las pruebas unitarias funcionan de forma inmediata al crear un nuevo proyecto de CLI de Angular. Normalmente, interactuaremos con las utilidades de pruebas de Angular.

Las utilidades de prueba de Angular nos ayudan a crear un entorno de pruebas que facilita la escritura de pruebas para nuestros artefactos de Angular. Consiste en la clase TestBed y varios métodos auxiliares en @angular/core/testing. A medida que avance este capítulo, aprenderemos qué son y cómo pueden ayudarnos a probar diversos artefactos. Por ahora, veamos los conceptos más comunes para que se familiarice con ellos cuando los analicemos con más detalle más adelante:

**TestBed** : Una clase que crea un módulo de prueba. Adjuntamos un artefacto Angular a este módulo de prueba al probarlo. La clase TestBed contiene el método configureTestingModule que usamos para configurar el módulo de prueba según sea necesario.

- **ComponentFixture**: Una clase contenedora de una instancia de componente Angular. Nos permite... para interactuar con el componente y su elemento HTML correspondiente.
- **DebugElement**: Un contenedor del elemento DOM del componente. Es una abstracción multiplataforma que permite que nuestras pruebas sean independientes de la plataforma.

Ahora que conocemos nuestro entorno de pruebas y los frameworks y bibliotecas utilizados, podemos empezar a escribir nuestras primeras pruebas unitarias en Angular.



Todos los ejemplos descritos en este capítulo se han creado en un nuevo proyecto Angular CLI.

Nos embarcaremos en este gran viaje desde el bloque de construcción más fundamental en Angular, el componente.

## Componentes de prueba

Es posible que hayas notado que cada vez que usamos la CLI de Angular para crear una nueva aplicación de Angular o generar un artefacto de Angular, creó algunos archivos de prueba para nosotros.

Los archivos de prueba en la CLI de Angular contienen la palabra "spec" en su nombre. El nombre de archivo de una prueba es el mismo que el del artefacto de Angular que está probando, seguido del sufijo .spec.ts. Por ejemplo, el archivo de prueba del componente principal de una aplicación Angular es app.component.spec.ts y se encuentra en la misma ruta que el archivo del componente.



Debemos considerar un artefacto Angular y su prueba correspondiente como algo. Al cambiar la lógica del artefacto, es posible que también debamos modificar la prueba unitaria. Colocar los archivos de pruebas unitarias con sus artefactos de Angular nos facilita recordarlos y editarlos. También nos ayuda cuando necesitamos refactorizar nuestro código, como mover artefactos (sin olvidar mover la prueba unitaria).

Al implementar el andamiaje de una nueva aplicación Angular, la CLI de Angular crea automáticamente una prueba para el componente principal, `AppComponent`. Al principio del archivo, hay una sentencia `beforeEach` que se utiliza para la configuración:

```
antes de cada(async () => {
  esperar TestBed.configureTestingModule({
    importaciones: [AppComponent],
  }).compileComponents();
});
```

Utiliza el método `configureTestingModule` de la clase `TestBed` y pasa un objeto como parámetro.

Podemos especificar una matriz de importaciones que contenga el componente que queremos probar. Además, podemos definir opciones de desmontaje mediante la propiedad de desmontaje .

La propiedad `teardown` contiene un objeto del tipo `ModuleTeardownOptions` que puede establecer las siguientes propiedades:

- `destroyAfterEach`: Crea una nueva instancia del módulo en cada prueba para eliminar errores causado por la limpieza incompleta de elementos HTML.
- `rethrowErrors`: Lanza cualquier error que ocurra cuando se destruye el módulo.

Finalmente, llamamos al método `compileComponents` para compilar la clase TypeScript y la plantilla HTML de nuestro componente.

La primera prueba unitaria verifica si podemos crear una nueva instancia de `AppComponent` usando el método `createComponent` :

```
it('debería crear la aplicación', () => {
  constante fixture = TestBed.createComponent(AppComponent);
  const aplicación = fixture.componentInstance;
```

```
    esperar(aplicación).toBeTruthy();
});
```

El resultado del método `createComponent` es una instancia `ComponentFixture` de `AppComponent`

Tipo que nos proporciona la instancia del componente mediante la propiedad `componentInstance`. También usamos la función de comparación `toBeTruthy` para comprobar si la instancia resultante es válida.

Tan pronto como tengamos acceso a la instancia del componente, podremos consultar cualquiera de sus propiedades públicas y métodos:

```
it('debería tener el título "my-app"', () => {
  constante fixture = TestBed.createComponent(AppComponent);
  const aplicación = fixture.componentInstance;
  esperar(app.title).toEqual('mi-aplicación');
});
```

En la prueba anterior, verificamos si la propiedad del componente de título está configurada en `my-app` usando otra función de comparación, `toEqual`.



El valor de la propiedad del componente de título en una nueva aplicación Angular será el nombre que pasó en el comando `ng new` al crear la aplicación.

Como hemos aprendido, un componente consta de una clase TypeScript y un archivo de plantilla. Por lo tanto, probarlo solo desde la perspectiva de la clase, como en la prueba anterior, no es suficiente. También debemos comprobar si la clase interactúa correctamente con el DOM:

```
it('debería mostrar el título', () => {
  constante fixture = TestBed.createComponent(AppComponent);
  accesorio.detectChanges();
  const compilado = fixture.nativeElement como HTMLElement;
  expect(compiled.querySelector('h1')?.textContent).toContain('Hola, mi-aplicación');

});
```



Muchos desarrolladores prefieren las pruebas de clase a las pruebas DOM y recurren a las pruebas de extremo a extremo (E2E), que son más lentas y tienen un rendimiento deficiente. Las pruebas E2E suelen validar la integración de una aplicación con una API de backend y son fáciles de descifrar. Por lo tanto, se recomienda realizar pruebas unitarias DOM en las aplicaciones Angular.

En la prueba anterior, creamos un componente y llamamos al método `detectChanges` de  `ComponentFixture`. Este método activa el mecanismo de detección de cambios de Angular , forzando la actualización de los enlaces de datos. Ejecuta el evento del ciclo de vida `ngOnInit` del componente la primera vez que se llama y `ngOnChanges` en las llamadas posteriores para poder consultar el elemento DOM del componente mediante la propiedad `nativeElement` . En este ejemplo, comprobamos el `textContent` del elemento HTML correspondiente a la propiedad `title` .

Para ejecutar pruebas, usamos el comando `ng test` de la CLI de Angular. Este comando iniciará el ejecutor de pruebas Karma, obtendrá todos los archivos de pruebas unitarias, los ejecutará y abrirá un navegador para mostrar los resultados de cada prueba. La CLI de Angular usa Google Chrome por defecto. El resultado será similar a esto:

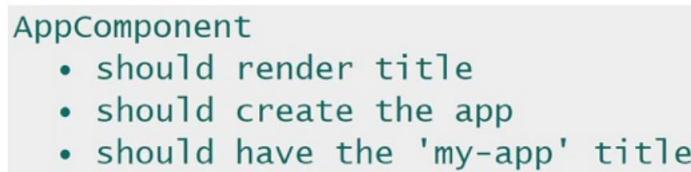


Figura 13.1: Salida de la ejecución de la prueba

En la figura anterior, podemos ver el resultado de cada prueba en la parte superior de la página. También podemos ver cómo Karma agrupa visualmente cada prueba por conjunto. En nuestro caso, el único conjunto de pruebas es `AppComponent`.

Ahora, hagamos que una de nuestras pruebas falle. Abra el archivo `app.component.ts` , cambie el valor de la propiedad `title` a `my-new-app` y guarde el archivo. Karma volverá a ejecutar nuestras pruebas y mostrará los resultados en la página:

```

AppComponent > should have the 'my-app' title
Expected 'my-new-app' to equal 'my-app'.
  at <Jasmine>
  at UserContext.apply (http://localhost:9876/_karma_webpack_/webpack:/src/app/app.component.spec.ts:20:23)
  at _ZoneDelegate.invoke (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone.js:368:26)
  at ProxyZoneSpec.onInvoke (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone-testing.js:273:39)
  at _ZoneDelegate.invoke (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone.js:367:52)

AppComponent > should render title
Expected 'Hello, my-new-app' to contain 'Hello, my-app'.
  at <Jasmine>
  at UserContext.apply (http://localhost:9876/_karma_webpack_/webpack:/src/app/app.component.spec.ts:27:55)
  at _ZoneDelegate.invoke (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone.js:368:26)
  at ProxyZoneSpec.onInvoke (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone-testing.js:273:39)
  at _ZoneDelegate.invoke (http://localhost:9876/_karma_webpack_/webpack:/node_modules/zone.js/fesm2015/zone.js:367:52)
  
```

Figura 13.2: Falla de prueba



Karma se ejecuta en modo de observación, por lo que no necesitamos ejecutar el comando de prueba de Angular CLI cada vez que realizamos un cambio.

A veces, leer el resultado de las pruebas en el navegador no es muy práctico. Como alternativa, podemos inspeccionar la ventana de consola que usamos para ejecutar el comando `ng test`, que contiene una versión reducida de los resultados de la prueba:

```
Se ejecutaron 3 de 3 con ÉXITO (0,117 segundos / 0,044 segundos)
TOTAL: 3 ÉXITOS
```

Hemos obtenido mucha información con solo observar la prueba de AppComponent que Angular CLI creó automáticamente. En la siguiente sección, veremos un escenario más avanzado para probar un componente con dependencias.

## Pruebas con dependencias

En un escenario real, los componentes no suelen ser tan simples como el componente principal. Es casi seguro que dependerán de uno o más servicios. También es posible que contengan otros componentes secundarios en su plantilla.

En estas situaciones, existen diferentes maneras de abordar las pruebas. Una cosa está clara: si probamos el componente, no deberíamos probar el servicio ni sus componentes secundarios. Por lo tanto, al configurar dicha prueba, la dependencia no debería ser la clase real. Existen diferentes maneras de abordar esto en las pruebas unitarias; ninguna solución es estrictamente mejor que otra:

- **Stubbing:** Un método que indica al inyector de dependencia que inyecte un stub de la dependencia. dencia que proporcionamos en lugar de la clase real.
- **Espionaje:** Un método que inyecta la dependencia real, pero adjunta un espía al método que llamamos en nuestro componente. Podemos devolver datos simulados o permitir que se ejecute la llamada al método.



Es preferible usar stubbing en lugar de espionaje cuando una dependencia es compleja. Algunos servicios inyectan otros servicios, por lo que usar la dependencia real en una prueba requiere compensar otras dependencias. También es el método preferido cuando el componente que queremos probar contiene componentes secundarios en su plantilla.

Independientemente del enfoque, nos aseguramos de que la prueba no realice acciones no deseadas, como acceder al sistema de archivos o intentar comunicarse a través de HTTP; estamos probando el componente de forma completamente aislada.

## Reemplazar la dependencia con un stub

Reemplazar una dependencia con un stub significa que reemplazamos completamente la dependencia con un falso.

Podemos crear una dependencia falsa de las siguientes maneras:

- Crear una variable o clase constante que contenga propiedades y métodos del valor real.  
pendencia.
- Cree una definición simulada de la clase real de la dependencia.

Los enfoques no son tan diferentes. En esta sección, analizaremos el primero, ya que es el más común en el desarrollo de Angular. Siéntete libre de explorar el segundo a tu propio ritmo.

Considere el siguiente archivo de componente stub.component.ts :

```
importar { Componente, OnInit } desde '@angular/core';
importar { StubService } desde './stub.service';

@Component({
  selector: 'app-stub',
  plantilla: '<span>{{msg}}</span>'
})
clase de exportación StubComponent implementa OnInit {
  mensaje = "";

  constructor(servicio auxiliar privado : Servicio auxiliar) {}

  ngOnInit(): vacío {
    este.msg = este.stubService.isBusy
      ? este.stubService.name +      'está en misión'
      : este.stubService.name +      'está disponible';
  }
}
```

Inyecta StubService, que contiene dos propiedades públicas. Proporcionar un stub para este servicio en las pruebas es bastante sencillo, como se muestra en el siguiente ejemplo:

```
const serviceStub: Partial<StubService> = {
  nombre: 'Boothstomper'
};
```

Hemos declarado el servicio como Parcial porque solo queremos establecer la propiedad del nombre inicialmente.

Ahora podemos usar la sintaxis literal de objeto para injectar el servicio stub en nuestro módulo de prueba:

```
esperar TestBed.configureTestingModule({
  importaciones: [StubComponent],
  proveedores: [
    { proporcionar: StubService, useValue: serviceStub }
  ]
})
.compileComponents();
```

La propiedad del componente msg depende del valor de la propiedad de servicio isBusy . Por lo tanto, necesitamos obtener una referencia al servicio en el conjunto de pruebas y proporcionar valores alternativos para esta propiedad en cada prueba. Podemos obtener la instancia inyectada de StubService mediante el método inject de la clase TestBed :

```
describe('estado', () => {
  dejar servicio: StubService;

  antes de cada(() => {
    servicio = TestBed.inject(StubService);
  })
});
```



Pasamos el StubService real como parámetro al método de inyección , no la versión del stub que creamos. Modificar el valor del stub no afectará el servicio inyectado, ya que nuestro componente usa una instancia del servicio real. El método de inyección solicita el servicio solicitado al inyector raíz de la aplicación. Si el servicio se proporcionó desde el inyector del componente, necesitaríamos obtenerlo de este mediante fixture.debugElement.injector.get(StubService).

Ahora podemos escribir nuestras pruebas para verificar si la propiedad del componente msg se comporta correctamente durante el enlace de datos:

```
describe('estado', () => {
  dejar servicio: StubService;
  dejar msgDisplay: HTMLElement;

  antes de cada() => {
    servicio = TestBed.inject(StubService); msgDisplay
    = fixture.nativeElement.querySelector('span');

    it('debería estar en una misión', () =>
      { service.isBusy = true;
        fixture.detectChanges();
        expect(msgDisplay.textContent).toContain('está en misión'); });

    it('debería estar disponible', () =>
      { service.isBusy = false;
        fixture.detectChanges();
        expect(msgDisplay.textContent).toContain('está disponible'); });
  }
})
```



Hemos eliminado la línea fixture.detectChanges de la declaración beforeEach porque queremos activar la detección de cambios en nuestras pruebas por separado.

Crear un stub de una dependencia no siempre es viable, especialmente cuando el inyector raíz no lo proporciona. Se puede proporcionar un servicio a nivel del inyector de componentes. Proporcionar un stub mediante el proceso que vimos anteriormente no tiene ningún efecto. Para abordar este escenario, podemos usar el método overrideComponent de la clase TestBed :

```
esperar TestBed.configureTestingModule({
  importaciones: [StubComponent],
  proveedores: [
    { proporcionar: StubService, useValue: serviceStub }
```

```
    ])}.overrideComponent(StubComponent, {  
  colocar: {  
    proveedores: [  
      { proporcionar: StubService, useValue: serviceStub }  
    ]  
  }  
});  
}) .compileComponents();
```

El método overrideComponent acepta dos parámetros: el tipo de componente que proporciona el servicio y un objeto de metadatos de anulación. Este objeto contiene la propiedad set , que proporciona servicios al componente.

Supongamos que el componente que queremos probar contiene un componente secundario en su plantilla, como por ejemplo:

```
@Componente({  
  selector: 'app-stub',  
  plantilla: `  
    <span>{{msg}}  
    <app-child></app-child>  
  `,  
})
```

En el caso anterior, cuando probamos StubComponent, también necesitábamos importar la clase TypeScript del componente <app-child> al configurar el módulo de prueba:

```
esperar TestBed.configureTestingModule({  
  importaciones: [StubComponent],  
  proveedores: [  
    { proporcionar: StubService, useValue: serviceStub }  
  ],  
  importa: [ChildComponent] })
```

La clase ChildComponent también puede tener otras dependencias. No es viable proporcionar stubs para dichas dependencias, ya que no es responsabilidad del componente en prueba. En su lugar, podemos crear una clase stub de TypeScript para el componente e importarla al configurar el módulo de prueba:

```
@Component({ selector: 'app-child', plantilla: " " })  
clase ChildStubComponent {}
```

En el fragmento anterior, pasamos una matriz vacía en la propiedad de plantilla del componente porque no nos interesa la implementación interna del componente secundario.



Si el componente secundario contiene propiedades y métodos que se utilizan al probar el componente principal, también debemos definirlos en ChildStubComponent.

Como alternativa, para proporcionar un fragmento del componente, podemos pasar NO\_ERRORS\_SCHEMA desde el paquete npm @angular/core mientras configuramos el módulo de prueba:

```
esperar TestBed.configureTestingModule({  
    importaciones: [StubComponent],  
    proveedores: [  
        { proporcionar: StubService, useValue: serviceStub },  
    ],  
    esquemas: [NO_ERRORS_SCHEMA]  
})
```

El fragmento anterior le indica a Angular que ignore cualquier componente que no se haya importado al módulo de prueba.

Crear una dependencia es muy sencillo, pero no siempre es posible, como veremos a continuación.

## Espiando el método de dependencia

Usar un stub no es la única manera de aislar la lógica en una prueba unitaria. No es necesario reemplazar toda la dependencia, solo las partes que usa nuestro componente. Reemplazar ciertas partes significa señalar métodos específicos en la dependencia y asignarles un espía. Un espía puede responder lo que quieras, pero también puedes ver cuántas veces se llamó y con qué argumentos. Por lo tanto, un espía te proporciona mucha más información sobre lo que está sucediendo.

Hay dos formas de configurar un espía en una dependencia:

- Inyectar la dependencia real y espia sus métodos.
- Use el método createSpyObj de Jasmine para crear una instancia de dependencia falsa. Luego podemos espia los métodos de esta dependencia como lo haríamos con la real.

El primer caso es el más común en el desarrollo con Angular. Veamos cómo configurarlo. Consideré el siguiente archivo spy.component.ts , que utiliza el servicio Title del framework Angular:

```
importar { Componente, OnInit } desde '@angular/core';
importar { Título } desde '@angular/platform-browser';

@Component({
  selector: 'app-spy',
  plantilla: '{{ caption }}'
})
clase de exportación SpyComponent implementa OnInit {
  título = "";

  constructor( título privado : Título ) {}

  ngOnInit(): vacío {
    this.title.setTitle('Mi aplicación Angular');
    este.caption = este.title.getTitle();
  }
}
```



El servicio de Título interactúa con el título del documento HTML principal en una aplicación Angular.

No tenemos control sobre el servicio Title , ya que está integrado en el framework. Puede tener dependencias desconocidas. Analizar sus métodos es la forma más sencilla y segura de usarlo en nuestras pruebas. Lo inyectamos en el módulo de pruebas mediante la matriz de proveedores y luego lo usamos en nuestra prueba, por ejemplo:

```
it('debería establecer el título', () => {
  const título = TestBed.inject(Título);
  const spy = spyOn(título, 'setTitle');
  componente.ngOnInit();
  expect(spy).toHaveBeenCalledWith('Mi aplicación Angular');
});
```

Utilizamos el método `spyOn` de Jasmine , que acepta dos parámetros: el objeto y su método específico para espiar. Lo usamos antes de llamar al método del componente `ngOnInit` para adjuntar el espía antes de activar el mecanismo de detección de cambios. La instrucción `expect` valida que `setTitle` El método fue llamado con los argumentos correctos.

Nuestro componente también utiliza el método `getTitle` para obtener el título del documento. Podemos espiarlo directamente. ese método y devuelve datos simulados:

1. Primero, necesitamos definir el servicio de título como un objeto espía e inicializarlo pasando dos parámetros: el nombre del servicio y una matriz de los nombres de los métodos que el componente usa actualmente:

```
const titleSpy = jasmine.createSpyObj('Título', [
  'obtenerTítulo', 'establecerTítulo'
]);
```

2. Luego adjuntamos un espía al método `getTitle` y devolvemos un título personalizado usando Jasmine método `returnValue` :

```
titleSpy.getTitle.and.returnValue('Mi título');
```

3. Finalmente, agregamos la variable `titleSpy` en la matriz de proveedores del módulo de prueba:

```
esperar TestBed.configureTestingModule({
  importaciones: [SpyComponent],
  proveedores: [
    { proporcionar: Título, useValue: titleSpy }
  ]
})
.compileComponents();
```

La prueba resultante debería verse así:

```
it('debería obtener el título', async () => {
  const titleSpy = jasmine.createSpyObj('Título', [
    'obtenerTítulo', 'establecerTítulo'
  ]);
  titleSpy.getTitle.and.returnValue('Mi título');

  esperar TestBed.configureTestingModule({
    importaciones: [SpyComponent],
```

```

proveedores: [
  { proporcionar: Título, useValue: titleSpy }
]
})
.compileComponents();

constante fixture = TestBed.createComponent(SpyComponent);
accesorio.detectChanges();

expect(fixture.nativeElement.textContent).toContain('Mi título');
});

```

Muy pocos servicios, como el servicio Title , se comportan correctamente y son sencillos, en el sentido de que son síncronos. La mayoría de las veces, son asíncronos y pueden devolver observables o promesas. En la siguiente sección, aprenderemos a probar dependencias asíncronas.

## Prueba de servicios asíncronos

Las utilidades de pruebas angulares proporcionan dos artefactos para abordar escenarios de pruebas asíncronas:

- waitForAsync: Un enfoque asíncrono para los servicios de pruebas unitarias. Se combina con...
   
Método whenStable de la clase ComponentFixture .
- fakeAsync: Un enfoque sincrónico para los servicios de pruebas unitarias. Se utiliza en combinación con
 la función de tictac .

Ambos enfoques proporcionan aproximadamente la misma funcionalidad; solo difieren en cómo los usamos.

Veamos cómo podemos utilizar cada uno mirando un ejemplo.

Considere el siguiente archivo async.component.ts :

```

importar { AsyncPipe } desde '@angular/common';
importar { Componente, OnInit } desde '@angular/core';
importar { Observable } de 'rxjs';
importar { AsyncService } desde './async.service';

@Component({
  selector: 'app-async',
  importaciones: [AsyncPipe],
  plantilla: `
    @for(item of items$ | async; rastrear elemento) {

```

```
<p>{{ artículo }}</p>
}

})
clase de exportación AsyncComponent implementa OnInit {
    items$: Observable<string[]> | indefinido;

constructor(servicio asíncrono privado : Servicio asíncrono) {}

ngOnInit(): vacío {
    este.items$ = este.asyncService.getItems();
}
}
```

Inyecta AsyncService desde el archivo async.service.ts y llama a su método getItems dentro del método ngOnInit . Como podemos ver, el método getItems devuelve un observable de cadenas. También introduce un ligero retraso para que el escenario parezca asíncrono:

```
getItems(): Observable<cadena[]> {
    retorno de(items).pipe(delay(500));
}
```

La prueba unitaria consulta el elemento nativo del componente y verifica si el valor del observable items\$ se muestra correctamente:

```
it('debería obtener datos con waitForAsync', waitForAsync(async() => {
    accesorio.detectChanges();
    esperar fixture.whenStable();
    accesorio.detectChanges();

    constante itemDisplay: HTMLElement[] = fixture.nativeElement.
    consultaSelectorAll('p');
    esperar(itemDisplay.length.toBe(2));
}););
```

Envolvemos el cuerpo de la prueba dentro del método waitForAsync y llamamos al método detectChanges para activar la detección de cambios. Además, llamamos al método whenStable , que devuelve una promesa que se resuelve inmediatamente al completarse el observable items\$ . Una vez resuelta la promesa, volvemos a llamar al método detectChanges para activar la vinculación de datos y consultar el DOM en consecuencia.



El método `whenStable` también se utiliza cuando queremos probar un componente que contiene un formulario basado en plantillas. La naturaleza asíncrona de este método hace que sea preferible usar formularios reactivos en nuestras aplicaciones Angular.

Un enfoque sincrónico alternativo sería utilizar el método `fakeAsync` y escribir lo mismo.

Prueba unitaria de la siguiente manera:

```
it('debería obtener elementos con fakeAsync', fakeAsync(() => {
  accesorio.detectChanges();
  garrapata(500);
  accesorio.detectChanges();

  constante itemDisplay: HTMLElement[] = fixture.nativeElement.
  consultaSelectorAll('p');
  esperar(itemDisplay.length).toBe(2);
}));
```

En el fragmento anterior, envolvimos el cuerpo de la prueba en un método `fakeAsync` y reemplazamos el método `whenStable` con la función `tick`. Esta función adelanta el tiempo 500 ms, que es el retardo virtual que introdujimos en el método `getItems` de `AsyncService`.

Probar componentes con servicios asíncronos a veces puede ser una pesadilla. Aun así, cada uno de los enfoques descritos puede ser de gran ayuda en esta tarea. Sin embargo, los componentes no solo se refieren a servicios, sino también a enlaces de entrada y salida. En la siguiente sección, aprenderemos a probar la API pública de un componente.

## Pruebas con entradas y salidas

Hasta ahora, hemos aprendido a probar componentes con propiedades simples y a abordar dependencias síncronas y asíncronas. Pero un componente es mucho más que eso. Como aprendimos en el Capítulo 3, "Estructura de interfaces de usuario con componentes", un componente tiene una API pública que consta de entradas y salidas que también deben probarse.

Dado que queremos probar la API pública de un componente, conviene comprobar cómo interactúa al alojarse desde otro componente. Hay dos maneras de probar un componente de este tipo:

- Podemos verificar que nuestro enlace de entrada esté configurado correctamente.
- Podemos verificar que nuestro enlace de salida se dispara correctamente y que lo que emite se recibe.

Supongamos que tenemos el siguiente archivo bindings.component.ts con un enlace de entrada y salida:

```
importar { Componente, entrada, salida } de '@angular/core';

@Component({
  selector: 'app-bindings', plantilla:

  <p>{{ title() }}</p> <button
    (click)="liked.emit()">¡Me gusta!</button>

})
clase de exportación BindingsComponent {
  título = entrada(); me
  gustó = salida();
}
```

Antes de comenzar a escribir nuestras pruebas, debemos crear un componente host de prueba dentro del archivo bindings.component.spec.ts que va a utilizar el componente bajo prueba:

```
@Component({
  importaciones: [BindingsComponent],
  plantilla: `

<app-bindings [title]="Título de prueba" (me gusta)="esFavorite = verdadero"></app-
bindings>

`)
export class TestHostComponent { testTitle
  = 'Mi título';
  isFavorite = falso;
}
```

En la fase de configuración, tenga en cuenta que ComponentFixture es del tipo TestHostComponent :

```
describe('BindingsComponent', () => { deja
  componente: TestHostComponent;
  dejar que el accesorio: ComponentFixture<TestHostComponent>;
  antes de cada(async () => {
```

```
esperar TestBed.configureTestingModule({ importaciones:  
    [TestHostComponent]  
  
}) .compileComponents();  
  
accesorio = TestBed.createComponent(TestHostComponent);  
componente = fixture.componentInstance;  
fixture.detectChanges();  
});  
  
it('debería crear', () => {  
    esperar(componente.toBeTruthy());  
  
});});
```

Nuestras pruebas unitarias validarán el comportamiento de BindingsComponent al interactuar con Componente de host de prueba.

La primera prueba verifica si la vinculación de entrada a la propiedad del título se ha aplicado correctamente:

```
it('debería mostrar el título', () => {  
    constante titleDisplay: HTMLElement = fixture.nativeElement.  
    consultaSelector('p');  
    esperar(títuloMostrar.textContent).toEqual(componente.pruebaTítulo);});
```

La segunda prueba valida si la propiedad isFavorite está conectada correctamente con la propiedad likes. evento de salida:

```
it('debería emitir el evento deseado', () => {  
    botón constante : HTMLButtonElement = fixture.nativeElement.  
    querySelector('botón');  
    botón.click();  
    esperar(componente.isFavorite).toBeTruthy();});
```

En la prueba anterior, consultamos el DOM para el elemento <button> mediante la propiedad nativeElement de la clase ComponentFixture . Luego, hacemos clic en él para que se emita el evento de salida. Como alternativa, podríamos haber usado la propiedad debugElement para encontrar el botón y usar su triggerEventHandler. Método para hacer clic en él:

```
it('debería emitir el evento deseado usando debugElement', () => {
  const buttonDe = fixture.debugElement.query(By.css('botón'));
  buttonDe.triggerEventHandler('clic');
  esperar(componente.isFavorite).toBeTrue();
});
```

En la prueba anterior, usamos el método de consulta , que acepta una función de predicado como parámetro. El predicado usa el método CSS de la clase By para localizar un elemento mediante su selector CSS.



Como aprendimos en la sección " Introducción a las pruebas unitarias en Angular" , debugElement es independiente del framework. Si está seguro de que sus pruebas solo se ejecutarán en un navegador, debería usar la propiedad nativeElement .

El método triggerEventHandler acepta el nombre del evento que queremos activar como parámetro; en este caso, es el evento clic .

Podríamos haber evitado mucho código si solo hubiéramos probado BindingsComponent, que habría seguido siendo válido. Pero habríamos perdido la oportunidad de probarlo en un escenario real.

La API pública de un componente está destinada a ser utilizada por otros componentes, por lo que debemos probarla de esta manera.

Actualmente, el botón que utilizamos en la plantilla de BindingsComponent es un HTML nativo <button> Elemento. Si el botón fuera un componente de Angular Material, podríamos usar un enfoque alternativo para interactuar con él, que es el tema de la siguiente sección.

## Prueba con un arnés de componentes

La biblioteca Angular CDK, núcleo de Angular Material, contiene un conjunto de utilidades que permiten que una prueba interactúe con un componente a través de una API de pruebas pública. Las utilidades de prueba de Angular CDK nos permiten acceder a los componentes de Angular Material sin depender de su implementación interna mediante un arnés de componentes.

El proceso de prueba de un componente Angular utilizando un arnés consta de las siguientes partes:

- @angular/cdk/testing: El paquete npm que contiene la infraestructura para interactuar con un arnés de componentes.

Entorno de pruebas: El entorno donde se cargará la prueba del componente. El CDK de Angular incluye un entorno de pruebas integrado para pruebas unitarias con Karma. También proporciona un amplio conjunto de herramientas que permiten a los desarrolladores crear pruebas personalizadas. entornos.

- Arnés de componentes: una clase que da al desarrollador acceso a la instancia de un componente dentro en el DOM del navegador.

Para aprender a utilizar los arneses de componentes, convertiremos el elemento <button> del EnlacesComponente en un botón de material angular:

```
importar { Componente, entrada, salida } de '@angular/core';
importar { MatButton } desde '@angular/material/button';

@Component({
  selector: 'enlaces de la aplicación',
  importaciones: [MatButton],
  plantilla: `
    <p>{{ título() }}</p>
    <button mat-button (click)="liked.emit()">¡Me gusta!</button>
  `
})
```



El fragmento anterior asume que ha agregado la biblioteca Angular Material al proyecto en el que está trabajando.

Para comenzar a usar un arnés de componentes del CDK de Angular, necesitamos importar los siguientes artefactos desde el espacio de nombres @angular/cdk/testing :

```
importar { TestbedHarnessEnvironment } desde '@angular/cdk/testing/testbed';
importar { MatButtonHarness } desde '@angular/material/button/testing';
```

En el fragmento anterior, hemos agregado las siguientes clases:

- TestbedHarnessEnvironment: representa el entorno de prueba para ejecutar pruebas unitarias con Karma.
- MatButtonHarness: el arnés del componente para el componente de botón de material angular.

Casi todos los componentes de la biblioteca Angular Material tienen un arnés de componentes correspondiente que podemos utilizar.



Si es autor de una biblioteca de componentes, Angular CDK proporciona todas las herramientas necesarias para crear arneses para sus componentes de UI.

Una vez que hayamos terminado de importar todos los artefactos necesarios, podemos escribir nuestra prueba:

```
it('debería emitir el evento deseado usando el arnés', async () => {
  const loader = TestBedHarnessEnvironment.loader(fixture);
  constante buttonHarness = await loader.getHarness(MatButtonHarness);
  esperar botónHarness.click();
  esperar(componente.isFavorite).toBeTrue();
});
```

En la prueba anterior, el método de carga del entorno de prueba acepta `ComponentFixture`

Instancia del componente actual como parámetro y devuelve un objeto `HarnessLoader`. La abstracción que proporciona un arnés de Angular CDK se basa en el concepto de que opera sobre la estructura del componente, que es una capa de abstracción sobre el elemento DOM real.

Rodeamos el cuerpo de la prueba con una función asíncrona , ya que los arneses de componentes se basan en promesas.

Usamos el método `getHarness` del cargador de arneses para cargar el arnés específico del componente botón. Finalmente, llamamos al método `click` del arnés del componente botón para activar el evento de clic.



No necesitamos llamar al método `detectChanges` porque el arnés del componente Angular CDK activa la detección de cambios automáticamente.

El arnés de componentes es una poderosa herramienta CDK de Angular que garantiza que interactuemos con los componentes de forma abstracta y segura durante las pruebas.

Hemos analizado muchas maneras de probar un componente con una dependencia. Ahora es el momento de aprender a probar la dependencia en sí.

## Servicios de pruebas

Como aprendimos en el Capítulo 5, Administración de tareas complejas con servicios, un servicio puede inyectar otros servicios. Probar un servicio independiente es bastante sencillo: obtenemos una instancia del inyector y luego comenzamos a consultar sus propiedades y métodos públicos.



Solo nos interesa probar la API pública de un servicio, que es la interfaz que utilizan los componentes y otros artefactos. Las propiedades y métodos privados no tienen valor al probarse, ya que representan la implementación interna del servicio.

Hay dos tipos diferentes de pruebas que podemos realizar en un servicio:

- Probar operaciones sincrónicas y asincrónicas, como un método que devuelve una matriz o una que devuelva un observable
- Probar servicios con dependencias, como un método que realiza solicitudes HTTP

En las siguientes secciones analizaremos cada uno de ellos con más detalle.

## Prueba de métodos sincrónicos/asincrónicos

Cuando creamos un servicio Angular usando Angular CLI, también crea un archivo de prueba correspondiente.

Considere el siguiente archivo `async.service.spec.ts`, que es el archivo de prueba para `AsyncService`

Usamos anteriormente:

```
importar { TestBed } desde '@angular/core/testing';

importar { AsyncService } desde './async.service';

describe('AsyncService', () => {
  dejar servicio: AsyncService;

  antes de cada() => {
    TestBed.configureTestingModuleTestingModule({});
    servicio = TestBed.inject(AsyncService);
  });

  it('debería ser creado', () => {
    esperar(servicio.toBeTruthy());
  });
});
```

El `AsyncService` no depende de nada. Además, se proporciona con el inyector raíz de la aplicación Angular, por lo que pasa un objeto vacío al método `configureTestingModule`. Podemos obtener una instancia del servicio que probamos usando el método `inject` de la clase `TestBed`.

La primera prueba que podemos escribir es bastante sencilla, ya que llama al método `setItems` e inspecciona su resultado:

```
it('debería establecer elementos', () => {
  const result = service.setItems('Cámara');
```

```
esperar(resultado.longitud.toBe(3));
```

Escribir una prueba para métodos sincrónicos, como en el caso anterior, suele ser relativamente fácil; sin embargo, las cosas son diferentes cuando queremos probar un método asíncrono como el siguiente.

Esta segunda prueba es un poco complicada porque involucra un observable. Necesitamos suscribirnos al método `getItems` e inspeccionar el valor en cuanto se complete el observable:

```
it('debería obtener los elementos', (hecho: DoneFn) =>
  { service.getItems().subscribe(items => {
    esperar(items.length.toBe(2);

    hecho();
  });
});
```

El ejecutor de pruebas Karma no sabe cuándo se completará un observable, por lo que proporcionamos el método `done` para indicar que el observable se ha completado y ahora podemos afirmar la expectativa. declaración.

## Servicios de prueba con dependencias

Probar servicios con dependencias es similar a probar componentes con dependencias. Todos los métodos que vimos en la sección "Probar componentes" se pueden aplicar de forma similar; sin embargo, seguimos un enfoque diferente al probar un servicio que inyecta el servicio `HttpClient`.

Considere el siguiente archivo `deps.service.ts` que utiliza el cliente HTTP:

```
importar { HttpClient } desde '@angular/common/http'; importar
{ Injectable } desde '@angular/core';

@Injectable({
  proporcionado en:
  'root'
})
clase de exportación DepsService {

  constructor(privado http: HttpClient) { }

  getItems()
  { devolver este.http.get('http://alguna.url');
  }
}
```

```
addItem(elemento: cadena) {
  devuelve esto.http.post('http://some.url', { nombre: elemento });
}
}
```

Las utilidades de prueba de Angular proporcionan dos artefactos para simular solicitudes HTTP en pruebas unitarias: la función provideHttpClientTesting , que proporciona un cliente HTTP para las pruebas, y HttpTestingController, que simula el servicio HttpClient . Podemos importar ambos desde @ espacio de nombres angular/common/http/testing :

```
importar { TestBed } desde '@angular/core/testing';
importar { provideHttpClient } desde '@angular/common/http'; importar
{ HttpTestingController, provideHttpClientTesting } desde '@angular/ common/http/testing';

importar { DepsService } desde './deps.service';

describe('DepsService', () => {
  dejar servicio: DepsService; dejar
  httpTestingController: HttpTestingController;

  beforeEach(() =>
    { TestBed.configureTestingModule({ proveedores:
      [ provideHttpClient(),
        provideHttpClientTesting()
      ]
    });
    servicio = TestBed.inject(DepsService);
    httpTestingController = TestBed.inject(HttpTestingController); });
});
```

Nuestras pruebas no deberían realizar una solicitud HTTP real. Solo necesitan validar que se realice con las opciones correctas. La siguiente es la primera prueba que valida el método getItems :

```
it('debería obtener elementos', () => {
  servicio.getItems().subscribe();
```

```
const req = httpTestingController.expectOne('http://alguna.url');
esperar(req.solicitud.método).toBe('GET');
});
```

En la prueba anterior, creamos una solicitud falsa usando el método `expectOne` de `HttpTestingController`, que toma una URL como argumento. El método `expectOne` crea un objeto de solicitud simulado y confirma que solo se realiza una solicitud a la URL específica. Tras crear la solicitud, podemos validar que su método sea GET.

Seguimos un enfoque similar al probar el método `addItem`, excepto que debemos asegurarnos de que el cuerpo de la solicitud contenga los datos correctos:

```
it('debería agregar un elemento', () => {
  servicio.addItem('Cámara').subscribe();
  const req = httpTestingController.expectOne('http://alguna.url');
  esperar(req.solicitud.método).toBe('POST');
  esperar(req.solicitud.cuerpo).toEqual({
    nombre: 'Cámara'
  });
});
```

Después de cada prueba, nos aseguramos de que no haya solicitudes pendientes sin coincidir utilizando el método de verificación . dentro de un bloque `afterEach` :

```
después de cada(() => {
  httpTestingController.verify();
});
```

En la siguiente sección, continuamos nuestro viaje a través del mundo de las pruebas aprendiendo cómo probar una tubería.

## Prueba de tuberías

Como aprendimos en el Capítulo 4, Enriquecimiento de Aplicaciones con Pipes y Directivas, una pipe es una clase de TypeScript que implementa la interfaz `PipeTransform`. Expone un método de transformación , que suele ser síncrono, lo que significa que es fácil de probar.

Considere el archivo `list.pipe.ts` que contiene una tubería que convierte una cadena separada por comas en una lista:

```
importar { Pipe, PipeTransform } desde '@angular/core';
```

```

@Tubo({
  nombre: 'lista'
})
La clase de exportación ListPipe implementa PipeTransform {

  transformar(valor: cadena): cadena[] {
    valor de retorno.split(',');
  }

}

```

Escribir una prueba es sencillo. Solo necesitamos crear una instancia de la clase ListPipe y verificar el resultado del método de transformación con datos simulados:

```

it('debería devolver una matriz', () => {
  constante pipe = nueva ListPipe();
  esperar(pipe.transform('A,B,C')).toEqual(['A', 'B', 'C']);
});

```



Las utilidades de prueba de Angular no intervienen al probar una tubería. Creamos una instancia de la clase tubería y podemos empezar a llamar al método de transformación .

Las directivas Angular son artefactos que no solemos crear con frecuencia, ya que la colección integrada del framework es más que suficiente. Sin embargo, si creamos directivas personalizadas, también deberíamos probarlas. En la siguiente sección, aprenderemos cómo lograrlo.

## Directivas de prueba

Las directivas suelen ser bastante sencillas en su estructura general, ya que son componentes sin vista asociada. El hecho de que las directivas suelen funcionar con componentes nos da una buena idea de cómo proceder al probarlos.

Considere el archivo copyright.directive.ts que creamos en el Capítulo 5, Enriquecimiento de aplicaciones mediante tuberías y directivas:

```

importar { Directiva, ElementRef } de '@angular/core';

@Directiva({
  selector: '[appCopyright]'
}

```

```

        })
exportar clase CopyrightDirective {

    constructor(el: ElementRef) {
        const currentYear = new Date().getFullYear(); const targetEl:
        HTMLElement = el.nativeElement;
        targetEl.classList.add('copyright');
        targetEl.textContent = `Copyright ©${currentYear} Todos los derechos
Reservado`;
    }

}

```

Una directiva se usa generalmente con un componente, por lo que conviene realizar pruebas unitarias mientras se usa en un componente. Creamos un componente host de prueba y añádalo a la matriz de importaciones de la prueba. módulo:

```

@Component({
    importaciones: [Directiva de Derechos de Autor],
    plantilla: '<span appCopyright>' })

clase TestHostComponent {}

```

Ahora podemos escribir nuestras pruebas que verifican si el elemento <span> contiene la clase de copyright y muestra el año actual en su propiedadtextContent :

```

describe('CopyrightDirective', () => { deja contenedor:
    HTMLElement;

    antes de cada() => {
        constante fixture = TestBed.configureTestingModule({
            importaciones:

                [TestHostComponent] }).createComponent(TestHostComponent);
        contenedor = fixture.nativeElement.querySelector('span');

        it('debería tener clase de derechos de autor', () => {
            esperar(contenedor.classList).toContain('derecho de autor');
        });
    };
}

```

```
});  
  
it('debería mostrar detalles de derechos de autor', () =>  
  { expect(container.textContent).toContain(new Date().getFullYear().  
toString())); });});});
```

Así de sencillo es probar una directiva. La clave es que se necesita un componente donde colocar la directiva y que se prueba implícitamente mediante el componente.

En la siguiente sección, aprenderemos cómo probar formularios reactivos.

## Formularios de prueba

Como vimos en el Capítulo 10, Recopilación de datos de usuario con formularios, los formularios son esenciales para una aplicación Angular . Es raro que una aplicación Angular no tenga al menos un formulario simple, como un formulario de búsqueda. En este capítulo, nos centraremos en los formularios reactivos, ya que son más fáciles de probar que los formularios basados en plantillas.

Considere el siguiente archivo search.component.ts :

```
importar { Componente } de '@angular/core'; importar  
{ FormGroup, FormControl, Validators, ReactiveFormsModule } de '@ angular/forms';  
  
@Componente({  
  selector: 'app-search', importa:  
  [ReactiveFormsModule],  
  plantilla: `  
    <form [formGroup]="Formulario de búsqueda" (ngSubmit)="buscar()">  
      <input type="text" formControlName="searchText"> <button  
        type="submit" [disabled]="searchForm.invalid">Buscar</br>  
    botón>  
    </form>  
  `  
})  
exportar clase SearchComponent  
{ searchForm = nuevo  
  FormGroup({ searchText: nuevo FormControl('', Validators.required)
```

```
});  
  
    buscar() {  
        si(este.formulariodebúsqueda.válido) {  
            console.log('Has buscado: ' + este.searchForm.controls.  
            textoDeBuscar.valor);  
        }  
    }  
}
```

En el componente anterior, podemos escribir nuestras pruebas unitarias para verificar que:

- El valor del control de formulario searchText se puede configurar correctamente
- El botón Buscar se desactiva cuando el formulario no es válido.
- El método console.log se llama cuando el formulario es válido y el usuario hace clic en Buscar.

botón

Para probar un formulario reactivo, primero debemos importar ReactiveFormsModule al módulo de prueba:

```
esperar TestBed.configureTestingModule({  
    importaciones: [SearchComponent, ReactiveFormsModule]  
})  
.compileComponents();
```

Para la primera prueba, necesitamos comprobar si el valor se propaga al control de formulario searchText cuando escribimos algo en el control de entrada:

```
it('debería establecer el texto de búsqueda', () => {  
    entrada constante : HTMLInputElement = fixture.nativeElement.  
    querySelector('entrada');  
    entrada.valor = 'Angular';  
    entrada.dispatchEvent(new CustomEvent('input'));  
    esperar(componente.searchForm.controles.searchText.valor.toBe('Angular'));  
});
```

En la prueba anterior, usamos el método querySelector de la propiedad nativeElement para encontrar el elemento HTML <input> y establecer su valor. Sin embargo, esto por sí solo no será suficiente para que el valor se propague al control de formulario. El framework Angular no sabrá si el valor del elemento HTML <input> ha cambiado hasta que activemos el evento DOM de entrada para ese elemento. Usamos el método dispatchEvent para activar el evento, que acepta un único método como parámetro que apunta a una instancia de la clase CustomEvent .

Ahora que estamos seguros de que el control de formulario searchText está conectado correctamente, podemos usarlo para escribir las pruebas restantes:

```
it('debería deshabilitar el botón de búsqueda', () => {
  botón constante : HTMLButtonElement = fixture.nativeElement.
    querySelector('botón');

  componente.searchForm.controls.searchText.setValue("");
  esperar(botón.deshabilitado.toBeTrue()); });

it('debería iniciar sesión en la consola', () => {
  botón constante : HTMLButtonElement = fixture.nativeElement.
    querySelector('botón');

  const spy = spyOn(console, 'log');
  componente.searchForm.controls.searchText.setValue('Angular');
  fixture.detectChanges();
  button.click();
  expect(spy).toHaveBeenCalledWith(' Buscaste: Angular');});
```

Tenga en cuenta que en la segunda prueba, configuramos el valor del control de formulario searchText y, a continuación, llamamos al método detectChanges para habilitar el botón. Al hacer clic en el botón, se activa el evento de envío del formulario, y finalmente podemos confirmar la expectativa de nuestra prueba.

En los casos en que un formulario tiene muchos controles, no es conveniente consultarlos dentro de nuestras pruebas. Como alternativa, podemos crear un objeto Página que se encargue de consultar elementos HTML y espia sobre los servicios:

```
clase Página {
  obtener searchText() { devolver esto.query<HTMLInputElement>('entrada'); } obtener
  submitButton() { devolver esto.query<HTMLButtonElement>('botón'); } consulta privada
  <T>(selector: cadena): T {
    devolver fixture.nativeElement.querySelector(selector);
  }
}
```

Luego podemos crear una instancia del objeto Page en la declaración beforeEach y acceder a sus propiedades y métodos en nuestras pruebas.

Como hemos visto, los formularios reactivos son muy fáciles de probar, ya que el modelo del formulario es la única fuente de información. En la siguiente sección, aprenderemos a probar partes de una aplicación Angular que... Utilice el enrutador.

## Probando el enrutador

Probar el código que interactúa con el enrutador Angular podría fácilmente ser un capítulo aparte. En esta sección, nos centraremos en los siguientes conceptos del enrutador:

- Componentes enrutados y de enrutamiento
- Guardias
- Resuelve

Veamos primero cómo probar componentes enrutados y enrutados.

Componentes enrutados y de enrutamiento. Un componente

enrutado se activa al navegar a una ruta específica de la aplicación. Considere el siguiente archivo app.routes.ts :

```
importar { Rutas } desde '@angular/router', importar
{ RoutedComponent } desde './routed/routed.component';

exportar const rutas: Rutas = [
  { ruta: 'enrutado', componente: RoutedComponent }
];
```

La clase RoutedComponent se define en el siguiente archivo routed.component.ts :

```
importar { Componente } desde '@angular/core';

@Component({
  selector: 'aplicación enrutada',
  plantilla: '<span>{{ título }}' })

clase de exportación RoutedComponent {
  título = 'Mi componente enrutado';
}
```

El componente anterior vincula el valor de la propiedad del componente title a un elemento HTML <span> . La prueba que escribiremos comprobará si la vinculación funciona correctamente.

Las pruebas de enrutadores Angular se basan en el enfoque de arnés de componentes que vimos en la sección "Probar componentes". Expone la clase RouterTestingModule , que contiene varios métodos de utilidad para trabajar con componentes enrutados en las pruebas:

```
importar { RouterTestingModule } desde '@angular/router/testing';
```

Antes de que podamos comenzar a probar un componente enrutado, debemos registrar la configuración de enrutamiento de la aplicación en el módulo de prueba:

```
antes de cada(async () => {
  esperar TestBed.configureTestingModule({
    proveedores: [provideRouter(rutas)]
  })
  .compileComponents();

  accesorio = TestBed.createComponent(RoutedComponent);
  componente = fixture.componentInstance;
  accesorio.detectChanges();
});
```

En el proceso de configuración anterior, proporcionamos la configuración de enrutamiento de la aplicación como en la aplicación. archivo config.ts .

Ya hemos aprendido que podemos consultar el DOM del elemento HTML nativo desde la clase ComponentFixture .

Cuando se carga un componente mediante el enrutador, usamos la propiedad routeNativeElement de la clase RouterTestingModule :

```
it('debería mostrar un elemento span', async () => {
  const arnés = await RouterTestingModule.crear();
  esperar arnés.navigateByUrl('/enrutado');
  esperar(arnés.routeNativeElement?.querySelector('span')?.textContent).
  toBe('Mi componente enrutado');
});
```

La prueba anterior se divide en los siguientes pasos:

1. Usamos el método de creación de RouterTestingModule para crear un nuevo arnés de enrutamiento para nuestro componente.
2. Navegamos a la ruta registrada mediante el método browseByUrl . Según la configuración de enrutamiento de la aplicación, la URL /routed activará el componente en prueba.

3. Usamos métodos de consulta estándar de la propiedad routeNativeElement para verificar que  
El elemento HTML <span> muestra el texto correcto.



La clase RouterTestingModule también contiene la propiedad routeDebugElement , que funciona en varias plataformas de manera similar a la propiedad debugElement de la clase ComponentFixture .

Un componente de enrutamiento se utiliza para navegar a otro componente en una aplicación Angular. Generalmente, implica llamar al método de navegación del servicio Router , como se indica a continuación:

```
importar { Componente } desde '@angular/core';
importar { Router } desde '@angular/router';

@Component({
  selector: 'aplicación enrutada',
  plantilla: '<span>{{ título }}</span>'
})
clase de exportación RoutedComponent {
  título = 'Mi componente enrutado';

  constructor(enrutador privado: Enrutador) {}

  volver() {
    este.router.navigate(['/']);
  }
}
```

Según el fragmento anterior, nuestra prueba debería verificar que el enrutador navegará a la ruta raíz cuando llamemos al método goBack :

```
it('debería navegar a la ruta raíz', () => {
  componente.goBack();
  esperar(TestBed.inject(Router).url).toBe('/');
});
```

En la prueba anterior, usamos el método inject de la clase TestBed para obtener una referencia al servicio Router . A continuación, accedemos a la propiedad url para verificar que el proceso de navegación se completó correctamente.

En la siguiente sección, aprenderemos cómo probar los protectores del enrutador.

## Guardias

En el Capítulo 9, Navegación a través de aplicaciones con enrutamiento, aprendimos que los protectores de enrutador son funciones simples.

Considere el siguiente guardia que verifica el estado de autenticación de un usuario:

```
importar { inyectar } desde '@angular/core';
importar { CanActivateFn, Router } desde '@angular/router'; importar
{ AuthService } desde './auth.service';

exportar const authGuard: CanActivateFn = () => { const
  authService = inject(AuthService); const router =
  inject(Router);

  si (authService.isLoggedIn) {
    devuelve verdadero;

  } devolver router.parseUrl('/');
};
```

En la guardia anterior, verificamos la propiedad isLoggedIn de la siguiente clase AuthService :

```
importar { Inyectable } desde '@angular/core';

@.Injectable({ providedIn:
  'root' }) exportar clase AuthService
  { isLoggedIn = false;
}
```



Decidimos mantener la clase AuthService simple y centrarnos en la lógica de la protección de autenticación.

Si la propiedad isLoggedIn es verdadera, el guardián también devuelve verdadero. De lo contrario, ejecuta el método parseUrl del servicio Router para redirigir a los usuarios a la ruta raíz.

La CLI de Angular ha creado la siguiente prueba unitaria para la protección:

```
importar { TestBed } desde '@angular/core/testing'; importar
{ CanActivateFn } desde '@angular/router';

importar { authGuard } desde './auth.guard';

describe('authGuard', () => {
  constante executeGuard: CanActivateFn = (...guardParameters) =>
    TestBed.runInInjectionContext(() => authGuard(...parámetros de protección));

  antes de cada(() => {
    TestBed.configureTestingModule({}); });

  it('debería ser creado', () => {
    esperar(executeGuard).toBeTruthy(); });

});
```

En el fragmento anterior, la variable "executeGuard" encapsula la creación de la función "authGuard" . Utiliza el método "runInInjectionContext " de la clase "TestBed" para permitir la inyección de los servicios requeridos mediante el método "inject" .

Para crear pruebas unitarias que validen el uso del guardián de autenticación, debemos ejecutar los siguientes pasos:

1. Modifique la declaración de importación del paquete npm @angular/router de la siguiente manera:

```
importar {
  Instantánea de ruta activada,
  CanActivateFn,
  Enrutador,
  RouterStateSnapshot }
de '@angular/router';
```

2. Agregue la siguiente declaración de importación :

```
importar { AuthService } desde './auth.service';
```

3. Cree las siguientes variables que correspondan a los servicios injectados:

```
dejar que authService: AuthService;  
deje que el enrutadorSpy: jasmine.SpyObj<Router>;
```

4. Inicialice las variables anteriores en la declaración beforeEach del conjunto de pruebas:

```
antes de cada(() => {  
    routerSpy = jasmine.createSpyObj('Enrutador', ['parseUrl']);  
  
    TestBed.configureTestingModule({  
        proveedores: [  
            { proporcionar: Enrutador, useValue: routerSpy }  
        ]  
    });  
  
    servicioAuth = TestBed.inject(ServicioAuth);  
});
```

En el fragmento anterior, usamos el método createSpyObj para crear un objeto espía para el servicio Router y proporcionárselo al módulo de pruebas. Además, obtenemos la instancia de la clase AuthService mediante el método inject de la clase TestBed , ya que es un servicio simple sin dependencias.

5. La primera prueba unitaria debe afirmar que la ejecución del protector devuelve verdadero cuando el usuario está autenticado:

```
it('debería devolver verdadero', () => {  
    authService.isLoggedIn = verdadero;  
    esperar(executeGuard({} como ActivatedRouteSnapshot, {} como  
    RouterStateSnapshot)).toBeTrue();  
});
```



Pasamos un objeto vacío para los parámetros ActivatedRouteSnapshot y RouterStateSnapshot porque no son necesarios en la protección.

6. La segunda prueba unitaria debe verificar que la ejecución del protector provoque una redirección a la ruta raíz:

```
it('debería redirigir', () => {
```

```
authService.isLoggedIn = false;
executeGuard({} como ActivatedRouteSnapshot, {} como
RouterStateSnapshot);
esperar(routerSpy.parseUrl).toHaveBeenCalledWith('/');
```

En la siguiente sección, aprenderemos cómo probar los resolutores de protección.

## Resolvedores

Los resolvers de enrutador son funciones simples de un tipo específico, similares a las guardas. El escenario más común al probar los resolvers es verificar que los datos devueltos sean correctos.

Considere el siguiente solucionador, que devuelve una lista de elementos:

```
importar { ResolveFn } desde '@angular/router';
importar { AsyncService } desde './async.service'; importar
{ inyectar } desde '@angular/core';

exportar const itemsResolver: ResolveFn<string[]> = () => {
  const asyncService = inyectar (AsyncService); devolver
  asyncService.getItems();
};
```



El solucionador utiliza el AsyncService que vimos anteriormente, que devuelve un observable de elementos usando el método getItems .

La CLI de Angular creará inicialmente el siguiente archivo de prueba unitaria al estructurar el solucionador:

```
importar { TestBed } desde '@angular/core/testing';
importar { ResolveFn } desde '@angular/router';

importar { itemsResolver } desde './items.resolver';

describe('itemsResolver', () => {
  const executeResolver: ResolveFn<boolean> = (...resolverParameters) =>
    TestBed.runInInjectionContext(() => itemsResolver(
      ...resolverParameters));

  antes de cada(() => {
```

```
 TestBed.configureTestingModule({});
```

```
it('debería ser creado', () => {
  esperar(ejecutarResolver).toBeTruthy();
});

});
```

En el fragmento anterior, la variable executeResolver encapsula la creación de la función itemsResolver , de forma similar a como lo hace con las protecciones. También utiliza el método runInInjectionContext de la clase TestBed para permitir la inyección de los servicios necesarios.

La lógica de nuestro resolver es muy simple, por lo que debemos escribir una sola prueba unitaria:

1. Modifique la declaración de importación del paquete npm @angular/router de la siguiente manera:

```
importar {
  Instantánea de ruta activada,
  ResolveFn,
  Instantánea del estado del enrutador
} de '@angular/router';
```

2. Agregue la siguiente declaración de importación :

```
importar { Observable } de 'rxjs';
```

3. Cambie el tipo de la variable executeResolver a ResolveFn<string[]> para que

coincide con la firma de la función itemsResolver :

```
const executeResolver: ResolveFn<string[]> = (...resolverParameters)
=>
  TestBed.runInInjectionContext(() => itemsResolver(
    ...resolverParameters));
```

4. Escribe la siguiente prueba unitaria:

```
it('debería devolver elementos', () => {
  (executeResolver() como ActivatedRouteSnapshot, {} como
  RouterStateSnapshot) como Observable<string[]>.subscribe(items =>
  { expect(items).toEqual(['Micrófono', 'Teclado']); })});
```

Para verificar que el solucionador devuelva datos correctos, debemos suscribirnos a `executeResolver` función.

En esta sección, aprendimos cómo realizar pruebas unitarias de algunas características importantes del enrutador Angular.

## Resumen

Hemos llegado al final de nuestro proceso de pruebas, un proceso largo pero emocionante. En este capítulo, vimos la importancia de introducir las pruebas unitarias en nuestras aplicaciones Angular, la estructura básica de una prueba unitaria y el proceso de configuración de Jasmine para nuestras pruebas.

También aprendimos a escribir pruebas robustas para nuestros componentes, directivas, tuberías y servicios. También hablamos sobre cómo probar los formularios reactivos de Angular y el enrutador.

Este capítulo de pruebas unitarias casi ha completado el rompecabezas de la creación de una aplicación Angular completa. Solo falta la última pieza, lo cual es importante porque las aplicaciones web están destinadas, en última instancia, a la web. Por lo tanto, en el siguiente capítulo, aprenderemos a producir una compilación de producción para una aplicación Angular y a implementarla para compartirla con el resto del mundo.

Machine Translated by Google

# 14

## Llevando aplicaciones a Producción

Una aplicación web normalmente debería ejecutarse en la web y ser accesible para cualquier persona desde cualquier lugar. Necesita dos ingredientes esenciales: un servidor web que la aloje y una compilación de producción para implementarla en dicho servidor. En este capítulo, nos centraremos en la segunda parte de la receta.

En resumen, una compilación de producción de una aplicación web es una versión optimizada del código de la aplicación: más pequeña, más rápida y con mayor rendimiento. Básicamente, es un proceso que toma todos los archivos de código de la aplicación, aplica técnicas de optimización y los convierte en un solo paquete. archivo.

En los capítulos anteriores, repasamos los diversos pasos necesarios para crear una aplicación Angular. Solo nos falta un último paso para conectar los puntos y poner nuestra aplicación a disposición de todos: crearla e implementarla en un servidor web.

En este capítulo aprenderemos los siguientes conceptos:

- Construyendo una aplicación Angular
- Limitar el tamaño del paquete de aplicaciones
- Optimización del paquete de aplicaciones
- Implementación de una aplicación Angular

## Requisitos técnicos

El capítulo contiene varios ejemplos de código que lo guiarán a través del concepto de llevar aplicaciones a producción.

Puede encontrar el código fuente relacionado en la carpeta ch14 del siguiente repositorio de GitHub:

<https://www.github.com/PacktPublishing/Aprendizaje-Angular-Quinta-Edición>

## Construyendo una aplicación Angular

Para construir una aplicación Angular, usamos el siguiente comando de la CLI de Angular:

```
construcción ng
```

El proceso de compilación inicia el compilador de Angular, que recopila principalmente todos los archivos TypeScript y HTML del código de nuestra aplicación y los convierte a JavaScript. Los archivos de hojas de estilo CSS, como SCSS, se convierten en archivos CSS puros. El proceso de compilación garantiza una representación rápida y óptima de nuestra aplicación en el navegador.

Una aplicación Angular contiene varios archivos TypeScript que no suelen usarse durante la ejecución, como pruebas unitarias o herramientas auxiliares. El compilador sabe qué archivos recopilar para el proceso de compilación leyendo la propiedad files del archivo tsconfig.app.json :

```
{
  "extiende": "./tsconfig.json",
  "opcionesdelcompilador": {
    "outDir": "./out-tsc/app",
    "tipos": []
  },
  "archivos": [
    "src/main.ts"
  ],
  "incluir": [
    "origen/**/*.d.ts"
  ]
}
```

El archivo src/main.ts es el punto de entrada principal de la aplicación y ayuda a Angular a revisar todos los componentes, servicios y otros artefactos Angular que nuestra aplicación necesita.

La salida del comando ng build se ve así:

Archivos de fragmentos iniciales   Nombres tamaño	Tamaño bruto   Transferencia estimada
principal-N4USDVTP.js kB	206,91 kB   55.87
polyfills-SCHOHYNV.js   polirellenlos kB	34,52 kB   11.29
estilos-5INURTSO.css   bytes de estilos	0 bytes   0
   Total inicial   241,44 kB	
KB	67.16

Esta salida muestra los archivos JavaScript y CSS generados a partir de la creación de la aplicación Angular, a saber:

- principal: El código de aplicación real que hemos escrito
- polyfills: Polyfills de funciones para navegadores más antiguos
- estilos: Estilos CSS globales de nuestra aplicación

El compilador de Angular genera los archivos anteriores en la carpeta dist\appName\browser , donde appName es el nombre de la aplicación. También contiene los siguientes archivos:

- favicon.ico: El ícono de la aplicación Angular
- index.html: El archivo HTML principal de la aplicación Angular

El comando ng build de la CLI de Angular se puede ejecutar en dos modos: desarrollo y producción. Por defecto, se ejecuta en modo de producción. Para ejecutarlo en modo de desarrollo, debemos ejecutar el siguiente comando de la CLI de Angular:

```
ng construir --configuración = desarrollo
```

El comando anterior tendrá una salida como la siguiente:

Archivos de fragmentos iniciales   Nombres main.js   main   polyfills estilos	Tamaño bruto   1,25 MB    90,23 kB    95 bytes
   Total inicial   1,35 MB	

En el resultado anterior, puede observar que los nombres de los archivos del fragmento inicial no contienen números hash, como en el caso de una compilación de producción. En el modo de producción, la CLI de Angular realiza diversas técnicas de optimización en el código de la aplicación, como la optimización de imágenes y la compilación anticipada (AOT) , para que el resultado final sea adecuado para su alojamiento en un servidor web y un entorno de producción. El número hash añadido a cada archivo garantiza que la caché del navegador los invalide rápidamente al implementar una versión más reciente de la aplicación.

Al ejecutar el comando `ng build` de la CLI de Angular en modo de desarrollo, usamos la opción `--configuration` . Esta opción nos permite ejecutar una aplicación Angular en diferentes entornos. Aprenderemos a definir entornos Angular en la siguiente sección.

## Edificios para diferentes entornos

Una organización podría querer desarrollar una aplicación Angular para múltiples entornos que requieren diferentes variables, como un punto final de API de backend y la configuración local de la aplicación. Un caso de uso común es un entorno de pruebas para probar la aplicación antes de implementarla en producción.

La CLI de Angular nos permite definir diferentes configuraciones para cada entorno y compilar nuestra aplicación con cada una. Podemos ejecutar el comando `ng build` pasando el nombre de la configuración como parámetro con la siguiente sintaxis:

```
ng build --configuration=nombre
```



También podemos pasar una configuración en otros comandos CLI de Angular, como `ng serve` y `ng prueba`.

Podemos utilizar el siguiente comando CLI de Angular para comenzar a trabajar con entornos:

```
ng genera entornos
```

Este comando creará una carpeta `src/environments` en el proyecto Angular que contiene los siguientes archivos:

- `environment.ts`: El entorno predeterminado de la aplicación, que se utiliza durante el proceso de producción
- `environment.development.ts`: El entorno de la aplicación utilizado durante el desarrollo

También agregará una sección fileReplacements en el archivo de configuración angular.json del proyecto Angular:

```
"desarrollo": {  
    "optimización": falso,  
    "extraerLicencias": falso,  
    "sourceMap": verdadero,  
    "Reemplazos de archivos": [  
        {  
            "reemplazar": "src/environments/environment.ts",  
            "con": "src/environments/environment.development.ts"  
        }  
    ]  
}
```

En el fragmento anterior, la propiedad fileReplacements define el archivo de entorno que reemplazará al predeterminado al ejecutar el comando de compilación en el entorno de desarrollo . Si ejecutamos el comando ng build --configuration=development , la CLI de Angular reemplazará el archivo environment.ts con el archivo environment.development.ts en el paquete de la aplicación.

Cada archivo de entorno exporta un objeto de entorno donde podemos definir propiedades de aplicación adicionales, como la URL de una API de backend:

```
exportar const entorno = {  
    apiUrl: 'https://mi-url-predeterminada'  
};
```



Las mismas propiedades del objeto exportado deben definirse en todos los archivos de entorno.

Necesitamos importar el entorno predeterminado para acceder a una propiedad de entorno en una aplicación Angular. Por ejemplo, para usar la propiedad apiUrl en el componente principal de la aplicación, debemos hacer lo siguiente:

```
importar { Componente } desde '@angular/core';  
importar { RouterOutlet } desde '@angular/router';  
importar { entorno } desde '../entornos/entorno';
```

```
@Componente({  
    selector: 'app-root',  
    importaciones: [RouterOutlet],  
    URL de plantilla: './app.component.html', URL  
    de estilo: './app.component.css'  
})  
exportar clase AppComponent {  
    título = 'mi-aplicación';  
    apiUrl = entorno.apiUrl;  
}
```

No todas las bibliotecas de una aplicación Angular se pueden importar como módulos JavaScript, a diferencia de la mayoría de las bibliotecas propias de Angular. En la siguiente sección, aprenderemos a importar bibliotecas que requieren el objeto de ventana global .

## Edificio para el objeto ventana

Una aplicación Angular puede usar una biblioteca como jQuery que debe adjuntarse al objeto de ventana .

Otras bibliotecas, como Bootstrap, tienen fuentes, íconos y archivos CSS que deben incluirse en el paquete de la aplicación.

En todos estos casos, necesitamos informar a la CLI de Angular sobre su existencia para que pueda incluirlos. en el paquete final.

El archivo de configuración angular.json contiene un objeto de opciones en la configuración de compilación que Podemos utilizar para definir dichos archivos:

```
"opciones": {  
    "outputPath": "dist/mi-aplicación",  
    "índice": "src/index.html",  
    "navegador": "src/main.ts",  
    "polyfills": [  
        "zona.js"  
    ],  
    "tsConfig": "tsconfig.app.json", "activos": [ {  
  
        "glob": "**/*",  
        "entrada": "público"  
    } ]  
}
```

```
        },
      ],
      "estilos": [
        "src/styles.css"
      ],
      "guiones": []
    }
```

El objeto de opciones contiene las siguientes propiedades que podemos utilizar:

- activos: contiene archivos estáticos de la carpeta pública , como íconos, fuentes y traducciones.
- estilos: Contiene archivos de hojas de estilo CSS externas. El archivo de hoja de estilo CSS global de la aplicación  
La catión está incluida de forma predeterminada.
- scripts: contiene archivos JavaScript externos.

A medida que añadimos más funciones a una aplicación Angular, el paquete final crecerá con el tiempo. En la siguiente sección, aprenderemos a mitigar este efecto mediante presupuestos.

## Limitar el tamaño del paquete de aplicaciones

Como desarrolladores, siempre buscamos crear aplicaciones impresionantes con características atractivas para el usuario final. Por ello, añadimos cada vez más funciones a nuestra aplicación Angular, a veces según las especificaciones y otras para aportar valor añadido a los usuarios. Sin embargo, añadir nuevas funcionalidades a una aplicación Angular hará que crezca, lo que podría resultar inaceptable en algún momento. Para solucionar este problema, podemos usar presupuestos.

Los presupuestos son umbrales que podemos definir en el archivo de configuración angular.json y que nos permiten asegurar que el tamaño de nuestra aplicación no los supere. Para establecerlos, podemos usar la propiedad "presupuestos" de la configuración de producción en el comando de compilación :

```
"presupuestos": [
  {
    "tipo": "inicial",
    "advertenciamáxima": "500kB",
    "Errormáximo": "1 MB"
  },
  {
    "tipo": "cualquierEstiloDeComponente",
    "advertenciamáxima": "4kB",
```

```
"Error máximo": "8 kB"
```

```
}
```

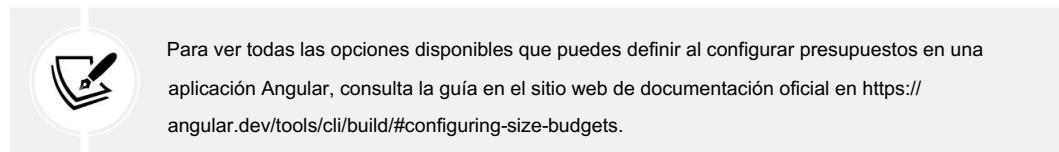
```
]
```

La CLI angular define los presupuestos predeterminados anteriores al crear un nuevo proyecto CLI angular.

Podemos definir un presupuesto para diferentes tipos, como la aplicación Angular completa o partes de ella. El límite de un presupuesto puede definirse en bytes, kilobytes, megabytes o un porcentaje. La CLI de Angular muestra una advertencia o un error cuando se alcanza o se supera el tamaño del límite definido.

Para entenderlo mejor, describamos el ejemplo predeterminado anterior:

- Se muestra una advertencia cuando el tamaño de la aplicación Angular supera los 500 KB y se produce un error cuando supere 1 MB.
- Se muestra una advertencia cuando el tamaño de cualquier estilo de componente supera los 4 KB y un error cuando supera los 8.



Los presupuestos son ideales cuando queremos proporcionar un mecanismo de alerta en caso de que nuestra aplicación Angular crezca significativamente. Sin embargo, son solo un nivel de información y precaución. En la siguiente sección, aprenderemos a minimizar el tamaño de nuestro paquete.

## Optimización del paquete de aplicaciones

Como aprendimos en la sección "Creación de una aplicación Angular" , la CLI de Angular realiza técnicas de optimización al crear una aplicación Angular. El proceso de optimización que se realiza en el código de la aplicación incluye técnicas y herramientas web modernas, entre ellas:

- Minificación: Convierte archivos fuente multilínea en una sola línea, eliminando espacios en blanco y comentarios. Este proceso permite que los navegadores los analicen más rápidamente posteriormente.
- Fealdad: cambia el nombre de las propiedades y los métodos a un formato no legible para los humanos, de modo que sean difíciles de entender y de usar con fines maliciosos.
- Agrupación: concatena todos los archivos fuente de la aplicación en un solo archivo, llamado manjo.

- Tree-shaking: elimina archivos no utilizados y artefactos angulares, como componentes y servicios, dando como resultado un paquete más pequeño.

Optimización de fuentes: Integra archivos de fuentes externas en el archivo HTML principal de la aplicación sin bloquear las solicitudes de renderizado. Actualmente es compatible con Google Fonts y Adobe Fonts, y requiere conexión a internet para descargarlas.

- Caché de compilación: almacena en caché el estado de compilación anterior y lo restaura cuando ejecutamos la misma compilación. disminuyendo el tiempo necesario para construir la aplicación.

Si el paquete final de una aplicación Angular sigue siendo grande después de todas las técnicas de optimización anteriores , podemos usar una herramienta externa llamada source-map-explorer para investigar la causa. Quizás hayamos importado una biblioteca JavaScript dos veces o incluido un archivo sin usar. La herramienta analiza el paquete de nuestra aplicación y muestra todos los artefactos y bibliotecas Angular que usamos en una representación visual.

Para comenzar a usarlo haga lo siguiente:

1. Instale el paquete npm source-map-explorer desde la terminal:

```
npm install source-map-explorer --save-dev
```

2. Cree su aplicación Angular y habilite los mapas de origen:

```
ng build --source-map
```

3. Agregue el siguiente script en el archivo package.json :

```
"guiones": {  
  "ng": "ng",  
  "inicio": "ng servicio",  
  "construir": "ng compilación",  
  "watch": "ng build --watch --configuration desarrollo",  
  "prueba": "prueba ng",  
  "analizar": "explorador de mapas fuente"  
}
```

4. Ejecute el siguiente comando contra el archivo del paquete principal :

```
npm ejecutar analizar dist/mi-aplicación/navegador/main*.js
```

Se abrirá una representación visual del paquete de aplicaciones en el navegador:

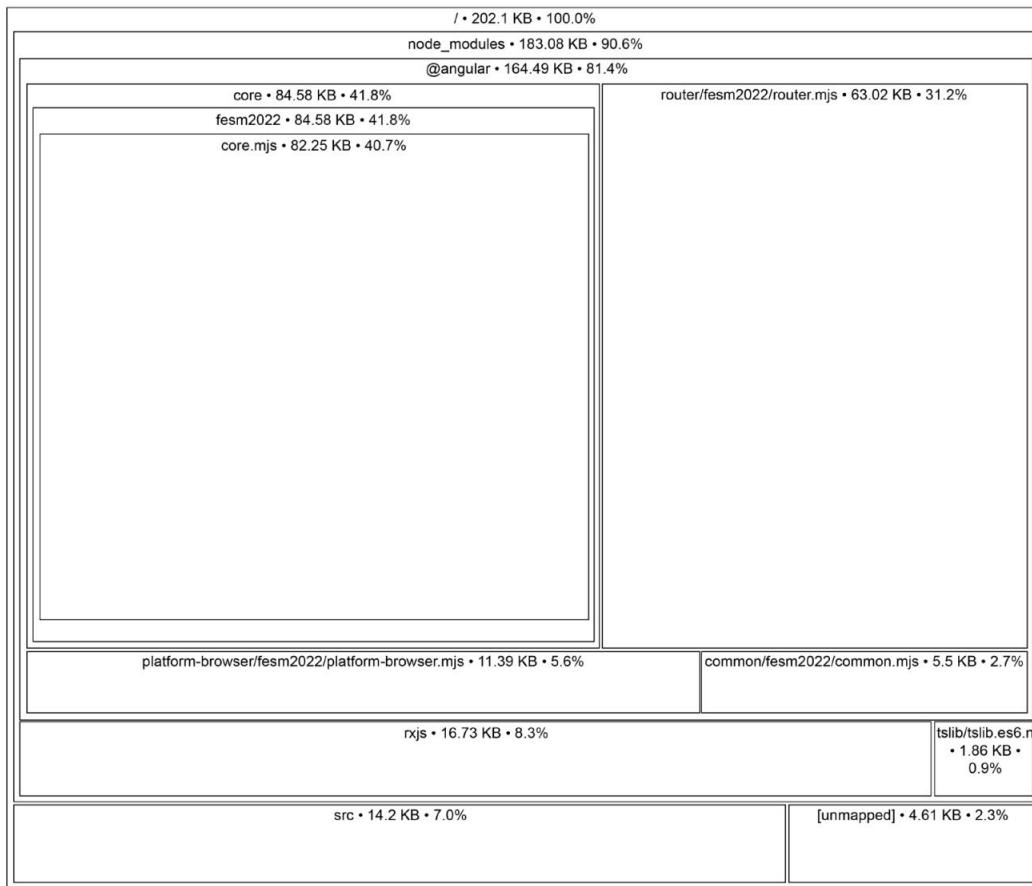


Figura 14.1: Salida del explorador de mapas de origen

Podemos interactuar con él e inspeccionarlo para comprender por qué nuestro paquete sigue siendo demasiado grande.

Algunas causas pueden ser las siguientes:

- Se incluye una biblioteca dos veces en el paquete.
- Se incluye una biblioteca que no se puede modificar, pero no se utiliza actualmente.

El último paso después de construir nuestra aplicación Angular es implementarla en un servidor web, como aprenderemos en la siguiente sección.

## Implementación de una aplicación Angular

Si ya tienes un servidor web que quieras usar para tu aplicación Angular, puedes copiar el contenido de la carpeta de salida a una ruta en ese servidor. Si quieres implementarla en una carpeta distinta a la raíz, puedes cambiar el atributo href de la etiqueta <base> en el archivo HTML principal de las siguientes maneras:

- Pasando la opción --base-href en el comando ng build :

```
ng build --base-href=/miruta/
```

- Establecer la propiedad baseHref en el comando de compilación de la configuración angular.json

archivo:

```
"opciones":  
  { "rutaDeSalida": "dist/mi-aplicación",  
    "índice": "src/index.html",  
    "navegador": "src/main.ts",  
    "baseHref": "/mypath/",  
    "polyfills": [  
      "zona.js"  
    ],  
    "tsConfig": "tsconfig.app.json",  
    "activos": [ {  
  
      "glob": "**/*", "input":  
      "público"  
  
    },  
    "estilos": [ "src/  
      styles.css"  
    ],  
    "guiones": []  
  }
```

Si no desea implementarlo en un servidor personalizado, puede usar las herramientas CLI de Angular para implementarlo en un proveedor de alojamiento compatible, que puede encontrar en <https://angular.dev/tools/cli/ implementación#implementación-automática-con-la-cli>.

## Resumen

La implementación de una aplicación Angular es la parte más sencilla y crucial, ya que permite que tu aplicación esté disponible para el usuario final. Las aplicaciones web se centran en brindar experiencias al usuario final al final del día.

En este capítulo, aprendimos cómo crear una aplicación Angular y prepararla para producción.

También investigamos diferentes formas de optimizar el paquete final y aprendimos cómo implementar una aplicación Angular en un servidor personalizado, de forma manual y automática, para otros proveedores de alojamiento.

En el próximo capítulo, que también es el capítulo final del libro, aprenderemos cómo mejorar el rendimiento de una aplicación Angular.

# 15

## Optimización de aplicaciones Actuación

Como desarrolladores y profesionales técnicos, desempeñamos un papel crucial en la creación e implementación de aplicaciones Angular, garantizando su rendimiento continuo y brindando una experiencia de usuario superior. Nuestros esfuerzos son fundamentales para el éxito de nuestras aplicaciones.

El comportamiento de una aplicación web y su rendimiento durante la ejecución son factores clave para la monitorización y la optimización. Debemos monitorizar y medir el rendimiento de la aplicación en caso de que esta comience a degradarse. Una de las métricas más populares para identificar problemas en aplicaciones web son las Core Web Vitals (CWV).

Tras determinar las causas de la degradación, podemos aplicar diversas técnicas de optimización. El framework Angular ofrece diversas herramientas para optimizar aplicaciones Angular, como la renderización del lado del servidor (SSR), la optimización de imágenes y la carga diferida de vistas. Si sabemos de antemano que la aplicación exigirá un alto rendimiento, es muy recomendable utilizar cualquiera de las herramientas mencionadas en las primeras etapas del desarrollo.

En este capítulo, exploraremos los siguientes conceptos de Angular con respecto a la optimización:

- Presentación de Core Web Vitals
- Representación de aplicaciones SSR
- Optimización de la carga de imágenes
- Aplazamiento de componentes
- Pre-renderizado de aplicaciones SSG

## Requisitos técnicos

Este capítulo contiene varios ejemplos de código que explican el concepto de optimización de aplicaciones Angular.

Puede encontrar el código fuente relacionado en la carpeta ch15 del siguiente repositorio de GitHub:

<https://github.com/PacktPublishing/Aprendizaje-Angular-Quinta-Edición>

## Presentamos Core Web Vitals

CWV es un conjunto de métricas que nos ayuda a medir el rendimiento de una aplicación web. Forma parte de Web Vitals, una iniciativa liderada por Google que unifica diversas guías y herramientas para medir el rendimiento de las páginas web. Cada métrica se centra en un aspecto específico de la experiencia del usuario, como la carga, la interactividad y la estabilidad visual de una página web:

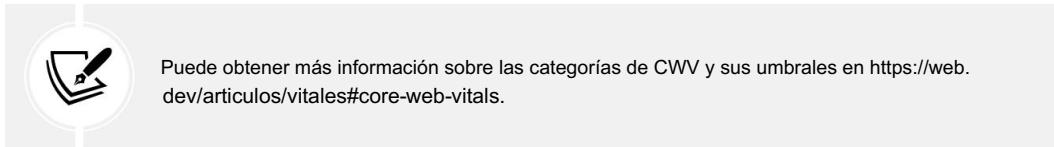
- Largest Contentful Paint (LCP): Mide la velocidad de carga de una página web calculando el tiempo que tarda en renderizarse el elemento más grande. Un valor de LCP rápido indica que la página está disponible para el usuario rápidamente.
- Interacción con la siguiente pintura (INP): Mide la capacidad de respuesta de una página web calculando el tiempo que tarda en responder a las interacciones del usuario y proporcionar retroalimentación visual. Un valor bajo de INP indica que la página responde al usuario rápidamente.
- Desplazamiento de Diseño Acumulado (CLS): Mide la estabilidad de la interfaz de usuario (IU) en una página web calculando la frecuencia con la que se producen cambios de diseño no deseados. Un cambio de diseño suele ocurrir cuando los elementos HTML se mueven en el DOM debido a una carga dinámica o asíncrona. Un valor bajo de CLS indica que la página es visualmente estable.



Web Vitals contiene métricas adicionales que contribuyen al conjunto CWV existente al medir un área más amplia o más específica de UX, como First Contentful Paint (FCP) y Time to First Byte (TTFB).

El valor de cada métrica CWV se divide en las siguientes categorías:

- BUENO (verde)
- NECESITA MEJORAR (naranja)
- POBRE (rojo)



Puede obtener más información sobre las categorías de CWV y sus umbrales en <https://web.dev/articulos/vitales#core-web-vitals>.

Podemos medir el CWV de las siguientes maneras:

- En el campo: Podemos utilizar herramientas como PageSpeed Insights y Chrome User Experience Informe mientras la aplicación web se ejecuta en producción.
- Programáticamente en JavaScript: Podemos utilizar API web estándar o bibliotecas de terceros como los web vitals.
- En el laboratorio: podemos utilizar herramientas como Chrome DevTools y Lighthouse mientras construimos la aplicación web durante el desarrollo.

En este capítulo, aprenderemos cómo usar Chrome DevTools para medir el rendimiento de nuestra aplicación de tienda electrónica:

1. Copie el código fuente del Capítulo 12, Introducción a Angular Material, en una nueva carpeta.
2. Ejecute el siguiente comando dentro de la nueva carpeta para instalar las dependencias del paquete:

```
instalación de npm
```

3. Ejecute el siguiente comando para iniciar la aplicación Angular:

```
ng servir
```

4. Abra Google Chrome y navegue a <http://localhost:4200>.

5. Active las herramientas de desarrollador y seleccione la pestaña Lighthouse . Lighthouse es una herramienta para medir diversos aspectos del rendimiento de una página web, incluyendo CWV. Google Chrome tiene una versión integrada de Lighthouse que podemos usar para comparar nuestra aplicación:

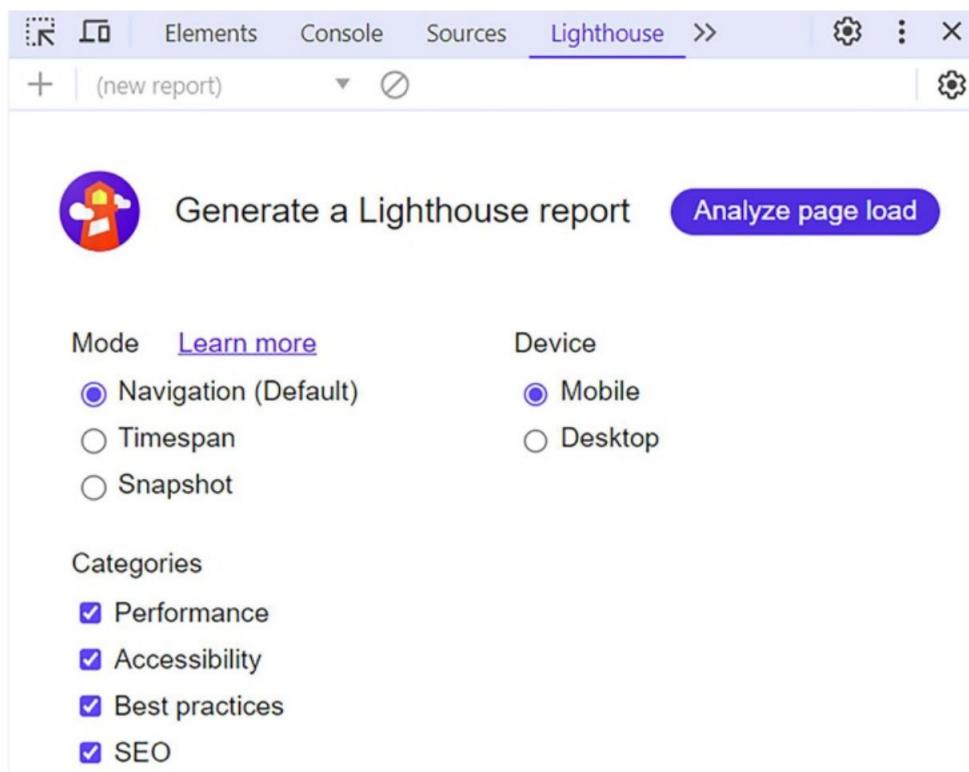


Figura 15.1: Pestaña Faro

En la pantalla que se muestra en la imagen anterior, podemos generar un informe de rendimiento de Lighthouse seleccionando varias opciones, incluyendo las secciones Dispositivo y Categorías . La sección Dispositivo nos permite especificar el entorno en el que queremos medir nuestra aplicación. La sección Categorías nos permite evaluar diferentes métricas, incluyendo el Rendimiento, relacionadas con CWV.

6. Seleccione la opción Escritorio en la sección Dispositivo , marque solo la opción Rendimiento en la sección Categorías y haga clic en el botón Analizar carga de página :

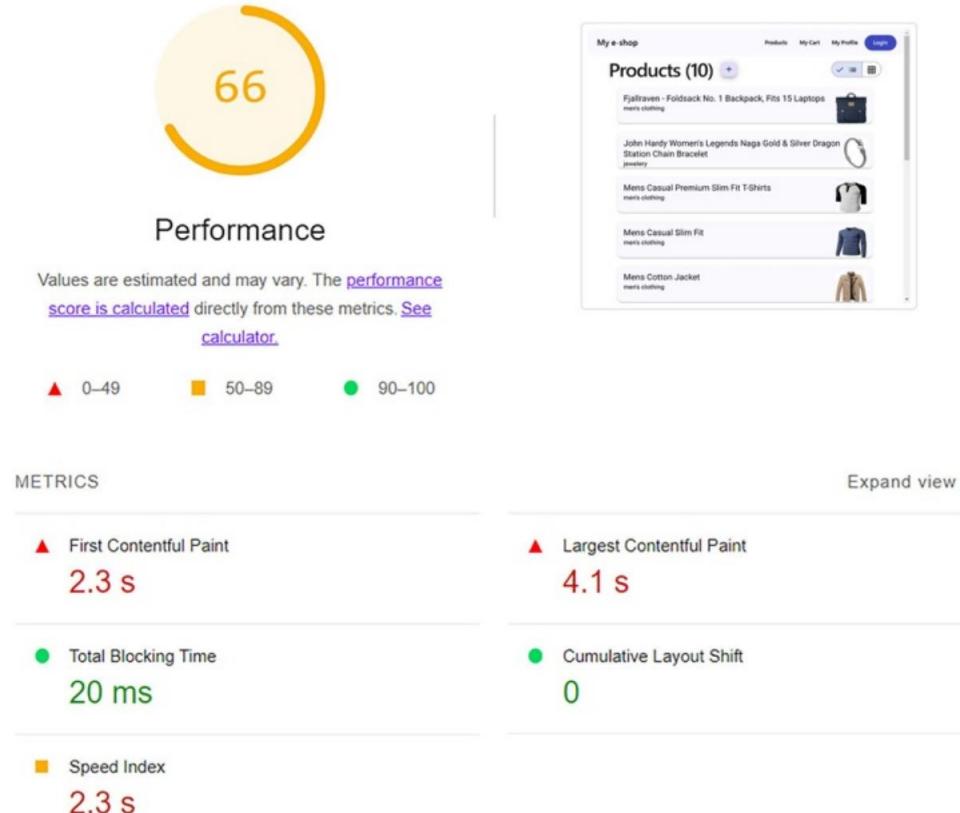
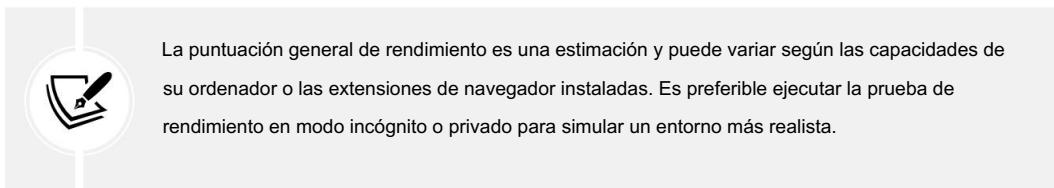


Figura 15.2: Informe de Lighthouse

En la imagen anterior, podemos ver la puntuación individual de las métricas CWV y la puntuación de rendimiento general.



En las siguientes secciones, exploraremos maneras de mejorar el rendimiento aplicando las mejores prácticas de Angular. Comenzaremos con SSR.

## Representación de aplicaciones SSR

SSR es una técnica de desarrollo web que mejora el rendimiento y la seguridad de las aplicaciones de las siguientes maneras:

- Mejora el rendimiento de carga al renderizar la aplicación en el servidor y eliminar el contenido HTML inicial entregado al cliente. El servidor entrega el HTML inicial al cliente, que puede analizarlo y cargarlo mientras espera el contenido JavaScript ser descargado.
- Mejora la optimización para motores de búsqueda (SEO) al permitir que los rastreadores web detecten e indexen la aplicación. El SEO proporciona contenido relevante cuando se comparte en aplicaciones de terceros, como plataformas de redes sociales.
- Mejora las métricas de CWV relacionadas con la velocidad de carga y la estabilidad de la interfaz de usuario, como LCP, FCP y CLS.
- Mejora la seguridad al agregar nonces CSP a las aplicaciones Angular.

Como vimos en el Capítulo 1, Creación de su primera aplicación Angular, cuando creamos una nueva aplicación usando la CLI de Angular, nos solicitó habilitar SSR:

¿Desea habilitar la representación del lado del servidor (SSR) y la generación de sitios estáticos (SSG/Prerenderización)? (sí/no)

En nuestro caso, ya hemos creado una aplicación Angular con la CLI de Angular. Para añadir SSR a una aplicación Angular existente, ejecute el siguiente comando en una ventana de terminal dentro del espacio de trabajo de la CLI de Angular:

```
ng agregar @angular/ssr
```

El comando anterior nos hará la siguiente pregunta:

¿Le gustaría utilizar las API de enrutamiento de servidor y App Engine (versión preliminar para desarrolladores) para esta aplicación de servidor? (sí/no)

Acepte el valor predeterminado, No, presionando Enter y la CLI de Angular nos solicitará que instalamos el paquete npm `@angular/ssr`.



Una característica en Developer Preview significa que aún no está lista para producción, pero puedes probarla en tu entorno de desarrollo.

Una vez completada la instalación, la CLI de Angular crea los siguientes archivos:

- main.server.ts: Esto se utiliza para iniciar la aplicación en el servidor utilizando un configuración.
- app.config.server.ts: Contiene la configuración de la aplicación renderizada en el servidor. Exporta una variable de configuración que contiene una versión fusionada de los archivos de configuración de la aplicación cliente y del servidor.
- server.ts: Configura e inicia un servidor Node.js Express que renderiza la aplicación Angular en el servidor. Utiliza la clase CommonEngine del paquete @angular/ssr para iniciar la aplicación Angular.

Además, el comando realizará las siguientes modificaciones en el espacio de trabajo de Angular CLI:

- Agregará las opciones necesarias en la sección de compilación del archivo angular.json para ejecutar la aplicación Angular en SSR y SSG.
- Agregará las entradas necesarias en las propiedades de archivos y tipos de tsconfig.app. archivo json para que el compilador de TypeScript pueda identificar los archivos creados para el servidor.
- Agregará los scripts y dependencias necesarios en el archivo package.json .
- Se añadirá provideClientHydration al archivo src\app\app.config.ts para habilitar la hidratación en la aplicación Angular. La hidratación es el proceso de restaurar la aplicación renderizada del lado del servidor al cliente. Aprenderemos más sobre la hidratación más adelante en este capítulo.

Ahora que hemos instalado Angular SSR en nuestra aplicación, veamos cómo usarlo:

1. Abra el archivo app.config.ts y modifique la declaración de importación de @angular/common/ espacio de nombres http como sigue:

```
importar { provideHttpClient, withFetch } desde '@angular/common/http';
```

El método withFetch se utiliza para configurar el cliente HTTP Angular para que utilice la API de búsqueda nativa para realizar solicitudes.

Se recomienda encarecidamente habilitar la búsqueda para las aplicaciones que usan SSR para lograr un mejor rendimiento y compatibilidad.

2. Pase el método withFetch como parámetro en el método provideHttpClient :

```
proporcionarHttpClient(withFetch())
```

3. Ejecute el siguiente comando para compilar la aplicación Angular:

```
construcción ng
```

El comando anterior genera paquetes de navegador y servidor dentro de dist\my-app  
Carpeta y rutas estáticas de prerenderizado. Aprenderemos más sobre prerenderizado en la sección  
"Prerenderizado de aplicaciones SSG".

- Ejecute el siguiente comando para ejecutar la aplicación SSR:

```
npm run serve:ssr:mi-aplicación
```

El comando anterior iniciará el servidor Express localmente en el puerto 4000 y servirá la aplicación SSR.

- Abra Google Chrome y navegue a <http://localhost:4000>. Debería ver la tienda online.  
aplicación en la página web.

- Repita el proceso aprendido en la sección anterior para ejecutar una prueba de rendimiento con Lighthouse.  
La puntuación general y las métricas de CWV deberían haber mejorado considerablemente:

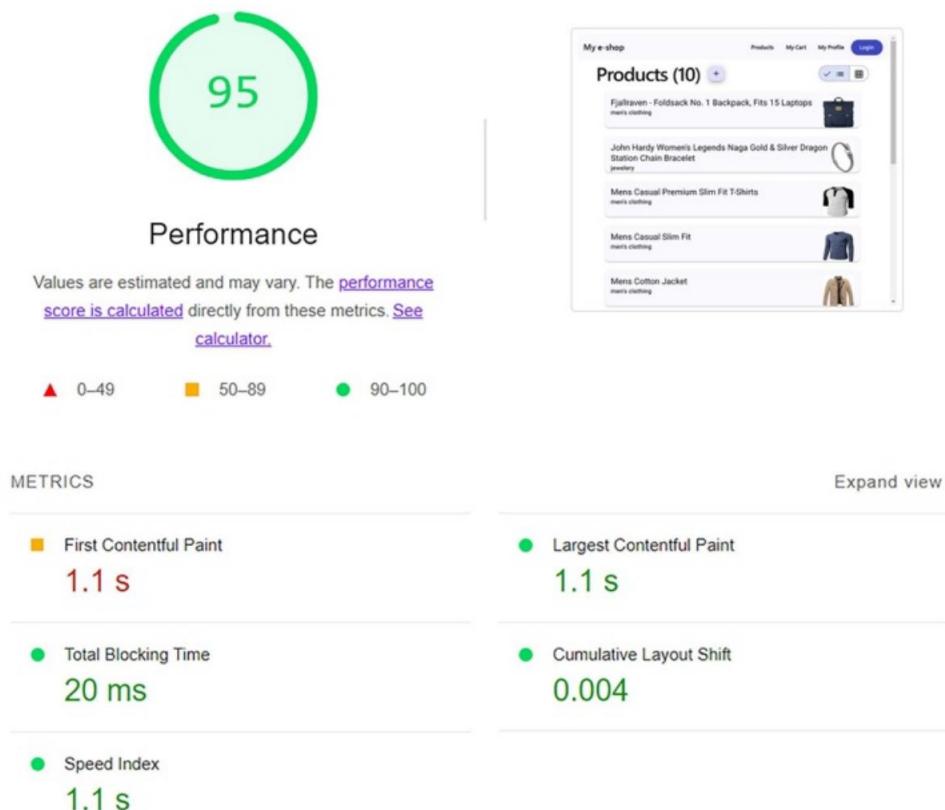


Figura 15.3: Informe Lighthouse (SSR)

El rendimiento de nuestra aplicación ha mejorado en más de un 20 % con solo instalar SSR en nuestra aplicación Angular. Como veremos más adelante en este capítulo, podemos aplicar diversas técnicas de Angular para mejorar aún más el rendimiento.

Angular SSR es una buena opción cuando necesitamos obtener datos del servidor y mostrarlos estáticamente en un sitio web. Sin embargo, hay casos en los que SSR no es beneficioso, como cuando una aplicación se basa en la entrada de datos y tiene muchas entradas de usuario.

En la siguiente sección, aprenderemos cómo anular SSR u omitirlo por completo para ciertas partes de una aplicación Angular.

## Anulación de SSR en aplicaciones Angular

La hidratación es una función importante habilitada por defecto en las aplicaciones Angular SSR. Mejora el rendimiento general de la aplicación al gestionar eficientemente la creación del DOM en el cliente. El cliente puede reutilizar la estructura del DOM de la aplicación renderizada en el servidor en lugar de crearla desde cero y forzar un parpadeo de la interfaz de usuario, lo que afecta a las métricas de CWV como LCP y CLS. El proceso de hidratación fallará en los siguientes casos:

- Cuando intentamos manipular el DOM a través de una API nativa del navegador, como una ventana o documento ya sea directamente o utilizando una biblioteca de terceros
- Cuando nuestras plantillas de componentes no tienen una sintaxis HTML válida

Podemos superar los problemas anteriores aplicando las siguientes prácticas recomendadas:

- Utilice las API de Angular para detectar la plataforma en la que se ejecuta nuestra aplicación antes de interactuar con el DOM
- Omitir la hidratación para componentes angulares específicos

Veamos cómo utilizar ambos con un ejemplo:

1. Ejecute la versión SSR de la aplicación Angular, como se muestra en la sección anterior.
2. Observe el texto que se muestra en el pie de página de la aplicación:

- v1 0

La información de derechos de autor no se muestra correctamente.

3. Abra el archivo `copyright.directive.ts` y concéntrese en el código del constructor :

```
constructor(el: ElementRef) {  
  const currentYear = nueva Fecha().getFullYear();  
  constante targetEl: HTMLElement = el.nativeElement;
```

```

targetEl.classList.add('derecho de autor');
targetEl.textContent = `Copyright ©${currentYear} Todos los derechos reservados`;

}

```

El código anterior usa la propiedad nativeElement para manipular el DOM añadiendo una clase CSS y estableciendotextContent del elemento HTML. Sin embargo, como se mencionó, el código causa problemas en nuestra aplicación porque no hay DOM en el servidor. ¡Vamos a solucionarlo!

4. Abra el archivo app.component.html y agregue el atributo ngSkipHydration en el <mat-toolbar> de la etiqueta HTML <footer> :

```

<pie de página>

<mat-toolbar ngSkipHydration>
  <mat-toolbar-row>
    <span appCopyright> - v${ settings.version }
  </mat-toolbar-row>
</mat-toolbar>

</pie de página>

```



ngSkipHydration es un atributo HTML, no una directiva de Angular. Solo se puede usar en otros componentes de Angular, no en elementos HTML nativos. No funcionaría si lo hubiéramos añadido en la etiqueta <footer> .

En el fragmento anterior, el componente <mat-toolbar> y sus componentes secundarios no se hidratarán. Esto significa que Angular los creará desde cero cuando la versión SSR de la aplicación esté lista.

5. Ejecute el paso 1 nuevamente y observe el resultado en el pie de página de la aplicación:

Copyright ©2024 Todos los derechos reservados - v1 0



Omitir la hidratación debería considerarse una solución alternativa. La usamos temporalmente cuando no se puede habilitar. Se recomienda refactorizar el código para que la aplicación pueda aprovechar las capacidades de hidratación.

Un enfoque alternativo y mejor es refactorizar nuestro código para que ejecute el código del cliente de manera condicional:

1. Modifique las declaraciones de importación en el archivo copyright.directive.ts de la siguiente manera:

```
importar { isPlatformBrowser } desde '@angular/common';
importar { Directiva, ElementRef, injectar, OnInit, PLATFORM_ID } desde
'@angular/core';
```

PLATFORM\_ID es un InjectionToken que indica el tipo de plataforma en la que se ejecuta nuestra aplicación . La función isPlatformBrowser comprueba si un ID de plataforma determinado corresponde al navegador.

Agregue la interfaz OnInit a la lista de interfaces implementadas de la Directiva de Derechos de Autor clase:

La clase de exportación CopyrightDirective implementa OnInit

2. Agregue las siguientes propiedades de clase:

```
plataforma privada = injectar(PLATFORM_ID);
privado el = injectar(ElementRef);
```

3. Elimine el constructor y agregue el siguiente método ngOnInit :

```
ngOnInit(): vacío {
    si (isPlatformBrowser(this.platform)) {
        const currentYear = nueva Fecha().getFullYear();
        const targetEl: HTMLElement = this.el.nativeElement;
        targetEl.classList.add('derecho de autor');
        targetEl.textContent = `Copyright ©${currentYear} Todos los derechos
reservados ${targetEl.textContent}`;
    }
}
```

La función isPlatformBrowser acepta el ID de la plataforma como parámetro.



Angular también proporciona la función isPlatformServer , una contraparte de la función isPlatformBrowser , que verifica si la plataforma actual es el servidor.

4. Cree y ejecute la aplicación en modo del lado del servidor para verificar que el mensaje de derechos de autor Todavía es visible.

En resumen, se recomienda usar Angular SSR en toda la aplicación y refactorizar las partes del código que deben ejecutarse en el navegador. Esto le permitirá aprovechar al máximo las ventajas de una aplicación renderizada del lado del servidor.

En la sección anterior, demostramos que añadir SSR a una aplicación Angular mejora drásticamente su rendimiento general. Como veremos en la siguiente sección, podemos mejorarlo aún más aplicando técnicas de optimización a las imágenes de producto.

## Optimización de la carga de imágenes

La lista de productos, que es el componente de aterrizaje de nuestra aplicación, muestra una imagen de cada producto. La forma en que se cargan las imágenes en una aplicación Angular puede afectar las métricas de CWV, como LCP y CLS. Actualmente, nuestra aplicación carga las imágenes tal como las recibe de la API de Fake Store. Sin embargo, podemos usar artefactos Angular específicos para aplicar las mejores prácticas al cargar imágenes.

El framework Angular nos proporciona la directiva NgOptimizedImage , que podemos adjuntar a elementos HTML <img> :

1. Abra el archivo product-list.component.ts e importe la clase NgOptimizedImage desde el paquete npm

@angular/common :

```
importar { AsyncPipe, CurrencyPipe, NgOptimizedImage } desde '@angular/
común';
```

2. Agregue la clase NgOptimizedImage en la matriz de importaciones del decorador @Component :

```
@Component({
  selector: 'lista-de-productos-de-aplicaciones',
  importaciones: [
    SortPipe,
    Tubería asíncrona,
    CurrencyPipe,
    Enlace de enrutador,
    Botón MatMiniFab,
    MatIcon,
    Módulo MatCard,
    MatTableModule,
    Botón de tapete para alternar,
    MatButtonToggleGroup,
    Imagen optimizada de Ng
  ],
  templateUrl: './lista-de-productos.component.html',
  styleUrls: ['./lista-de-productos.component.css']
})
```

3. Abra el archivo product-list.component.html y reemplace el enlace de la propiedad src con la directiva ngSrc :

```
<mat-card-title-group>
  <mat-card-title>{{ producto.título }}</mat-card-title>
  <mat-card-subtitle>{{ producto.categoría }}</mat-card-subtitle>
  <img mat-card-sm-image [ngSrc]="producto.imagen" />
  Grupo de títulos de tarjetas de tapete
```

La directiva ngSrc no es suficiente para evitar cambios de diseño al cargar la imagen. También debemos definir el tamaño de la imagen mediante los atributos de ancho, alto o relleno . En este caso, usaremos este último porque el tamaño de cada imagen no es el mismo para todos los productos.

```
<img mat-card-sm-image [ngSrc]="producto.imagen" fill />
```

4. Abra el archivo product-list.component.css y agregue los siguientes estilos CSS a la posición

La imagen en la parte superior derecha del contenedor:

```
imagen {
  objeto-apto: contener;
  posición del objeto: derecha 5px arriba 0;
}
```

5. Ejecute el siguiente comando para iniciar la aplicación:

```
ng servir
```

6. Navegue a <http://localhost:4200> y verifique que la lista de productos se muestre correctamente.

Los beneficios obtenidos al usar la directiva NgOptimizedImage no se aprecian inmediatamente en la interfaz de usuario. Esta directiva funciona en segundo plano y mejora automáticamente la métrica LCP de CWV mediante:

- Establecer la prioridad de búsqueda en el elemento HTML <img>
- Carga diferida de imágenes
- Configuración de etiquetas de enlace de preconexión y sugerencias de precarga en el caso de SSR
- Generación de atributos srcset para imágenes responsivas

Además, ayuda a los desarrolladores a seguir las mejores prácticas con respecto a la carga de imágenes, como:

- Establecer el tamaño de la imagen si se conoce de antemano
- Carga de imágenes a través de una CDN
- Visualización de advertencias apropiadas en la ventana de la consola para diferentes métricas

La directiva NgOptimizedImage contiene muchas otras funciones que podemos habilitar para lograr mejoras de rendimiento significativas, como configurar cargadores de imágenes, usar marcadores de posición y definir la prioridad de carga de imágenes. Puede encontrar más información en <https://angular.dev/guide/image-optimization>.

Ya hemos aprendido sobre diversas herramientas para mejorar el rendimiento de las aplicaciones. Una de las más eficaces son las vistas diferibles, que veremos en la siguiente sección.

## Aplazamiento de componentes

La introducción de la nueva sintaxis de flujo de control permitió a Angular integrar nuevas primitivas en el framework, mejorando la ergonomía, la experiencia de usuario (DX) y el rendimiento de las aplicaciones Angular. Una de estas primitivas son las vistas diferibles, que permiten la carga diferida de un componente Angular y sus dependencias.

## Presentamos las vistas diferibles

Ya hemos aprendido a usar el enrutador Angular para la carga diferida de un componente según una ruta específica. Las vistas diferibles proporcionan una nueva API que complementa la anterior. Su combinación con el enrutamiento de carga diferida garantiza el desarrollo de aplicaciones web potentes y de alto rendimiento. Las vistas diferibles permiten la carga diferida de un componente según un evento o su estado, y presentan las siguientes características:

- Son fáciles de usar y de razonar sobre el código adjunto.
- Los definimos de forma declarativa
- Minimizan la carga inicial de la aplicación y el tamaño final del paquete, mejorando las métricas de CWV como LCP y TTFB

Cada vista diferible se divide en un fragmento independiente, similar a los archivos de fragmentos individuales generados por las rutas de carga diferida. Constan de los siguientes bloques HTML:

- `@defer`: Indica el contenido HTML que se cargará.
- `@placeholder`: indica el contenido HTML que se muestra antes de que el bloque `@defer` comience a cargarse. Es especialmente útil cuando la aplicación se carga en una red lenta o cuando queremos evitar el parpadeo de la interfaz de usuario.
- `@loading`: Indica el contenido HTML que será visible mientras se carga el bloque `@defer`.
- `@error`: Indica el contenido HTML que se muestra si ocurre un error mientras el bloque `@defer` está activo. cargando.

Aprenderemos cómo utilizar cada bloque en la siguiente sección.

Uso de bloques diferibles. Integraremos vistas diferibles en nuestra aplicación de tienda online mediante la creación de un componente que muestre un producto destacado de la API de Tienda Falsa que no esté actualmente en la lista de productos. Comencemos:

1. Ejecute el siguiente comando para crear el nuevo componente:

```
ng generar componente destacado
```

2. Abra el archivo products.service.ts y agregue el siguiente método, que obtiene un valor específico.

Producto con ID 20 de la API de Fake Store:

```
getFeatured(): Observable<Producto> {
  devuelve este.http.get<Product>(this.productsUrl + '/20');
}
```

3. Abra el archivo featured.component.ts y modifique las declaraciones de importación de la siguiente manera:

```
importar { Component, OnInit } desde '@angular/core'; importar
{ CommonModule } desde '@angular/common'; importar
{ MatButton } desde '@angular/material/button';
importar { MatCardModule } de '@angular/material/card'; importar
{ Observable } de 'rxjs';
importar { Producto } de '../producto'; importar
{ ProductosServicio } de '../productos.servicio';
```

4. Modifique la matriz de importaciones del decorador @Component de la siguiente manera:

```
@Component({
  selector: 'con funciones de la aplicación',
  importaciones: [CommonModule, MatButton, MatCardModule],
  templateUrl: './featured.component.html',
  styleUrls: ['./featured.component.css']
})
```

5. Modifique la clase FeaturedComponent de la siguiente manera:

```
clase de exportación FeaturedComponent implementa OnInit
{ producto$: Observable<Product> | indefinido;

  constructor(productoServicioprivado: ProductosServicio) {}

  ngOnInit() {
```

```
        este.producto$ = este.productoService.getFeatured();
    }
}
```

En la clase TypeScript anterior, declaramos el observable `product$` y lo asignamos al valor devuelto del método `getFeatured` de la clase `ProductsService`.

6. Abra el archivo `featured.component.html` y reemplace su contenido con el siguiente HTML código:

```
@if (producto$ | async; como producto) {
  <mat-card>
    <mat-card-header>
      <mat-card-title>MEGA OFERTA</mat-card-title>
      <mat-card-subtitle>{{ producto.título }}</mat-card-subtitle>
    encabezado de tarjeta mat
    <img mat-card-image [src]="producto.imagen" />
    <mat-card-actions>
      Comprar ahora
    </mat-card-actions>
  </mat-card>
}
```

En el fragmento anterior, usamos la tubería asíncrona para suscribirnos al observable `product\$` dentro del bloque `@if`. El contenido HTML del bloque muestra los detalles del producto como un componente de tarjeta de Angular Material.

7. Abra el archivo `featured.component.css` y agregue los siguientes estilos CSS para los componentes de la tarjeta y del botón:

```
tarjeta de tapete {
  ancho máximo: 350px;
}

botón {
  ancho: 100%;
}
```

El nuevo componente Angular ya está instalado. Debemos agregarlo al componente principal de la aplicación y usar un bloque @defer para cargarlo:

1. Abra el archivo app.component.ts y agregue la siguiente declaración de importación :

```
importar { FeaturedComponent } desde './featured/featured.component';
```

2. Agregue la clase FeaturedComponent en la matriz de importaciones del decorador @Component :

```
@Component({
  selector: 'app-root',
  importaciones: [
    Salida de enrutador,
    Enlace de enrutador,
    Directiva sobre derechos de autor,
    Componente de autenticación,
    MatToolbarRow,
    MatToolbar,
    Botón Mat,
    MatBadge,
    Componente destacado
  ],
  URL de plantilla: './app.component.html', URL de
  estilo: './app.component.css' })
```

3. Abra el archivo app.component.html y agregue el componente <app-featured> dentro del

Eiqueta HTML <main> :

```
<main class="principal">
  <div class="contenido">
    <enrutador-de-salida />
  </div>
  @defer()
  { <app-featured />
  }
</principal>
```

En el fragmento anterior, utilizamos el bloque @defer para declarar el componente <app-featured> utilizando la sintaxis de etiqueta autoencapsulada.

4. Ejecute el comando `ng serve` para iniciar la aplicación y observar los archivos del fragmento Lazy.  
sección en la ventana de terminal:

```
Archivos de fragmentos | Nombres | Tamaño en bruto  
diferidos chunk-OP24QI45.mjs | componente destacado | 2,88 kB | chunk-4T4L5V7V.mjs |  
rutas de usuario | 1,19 kB |
```

El código fuente del componente destacado se divide en un archivo fragmentado.

5. Navegue a `http://localhost:4200` y observe el nuevo componente en el lado derecho de la lista de productos:



Figura 15.4: Producto destacado

Intente volver a cargar el navegador y notará que la interfaz de usuario parpadea mientras se carga el producto destacado. Usaremos el bloque `@placeholder` para mostrar una imagen de contorno antes de que el componente destacado comience a cargarse:

1. Copie la imagen `placeholder.png` de la carpeta pública del repositorio de GitHub descrita en la sección Requisitos técnicos a la carpeta correspondiente de su espacio de trabajo.
2. Agregue un bloque `@placeholder` después del bloque `@defer` de la siguiente manera:

```
@defer() {  
  <app-featured />  
} @placeholder(mínimo 1s) {  
    
}
```

El bloque `@placeholder` acepta un parámetro opcional que define el tiempo mínimo que el marcador de posición estará visible. En este caso, hemos definido el tiempo mínimo en 1 segundo.

3. Ejecute la aplicación usando el comando `ng serve` y verifique que el siguiente marcador de posición

La imagen es visible durante 1 segundo antes de que se cargue el contenido real:

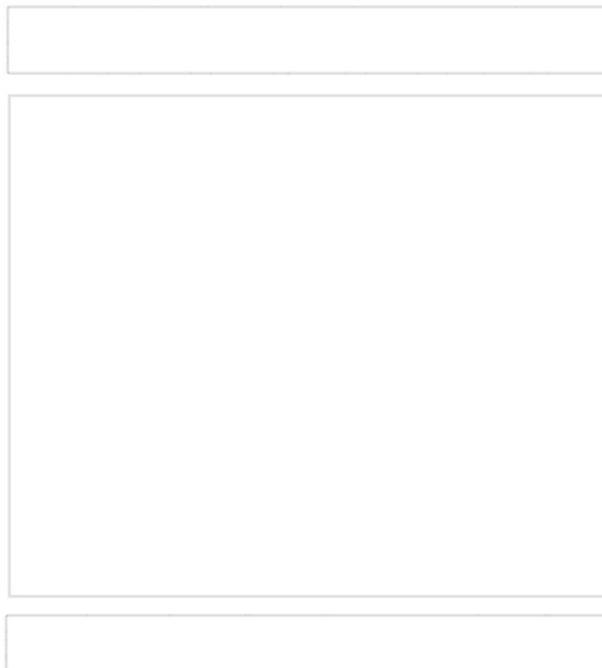


Figura 15.5: Imagen de marcador de posición

Un enfoque alternativo sería utilizar el bloque @loading y mostrar un indicador de carga, como un spinner, mientras se carga el componente destacado:

1. Abra el archivo app.component.ts y agregue la siguiente declaración de importación :

```
importar { MatProgressSpinner } desde '@angular/material/progress-spinner';
```

La clase MatProgressSpinner es un componente giratorio de la biblioteca Angular Material.

2. Agregue la clase MatProgressSpinner en la matriz de importaciones del decorador @Component :

```
@Componente({  
    selector: 'app-root',  
    importaciones: [  
        Salida de enrutador,  
        Enlace de enrutador,  
        Directiva sobre derechos de autor,  
        Componente de autenticación,  
        MatToolbarRow,  
        MatToolbar,  
        Botón Mat,  
        MatBadge,  
        Componente destacado,  
        MatProgressSpinner ],
```

```
URL de plantilla: './app.component.html', URL de  
estilo: './app.component.css' })
```

3. Agregue el bloque @loading en el archivo app.component.html de la siguiente manera:

```
@defer() {  
    <app-featured /> }  
    @loading(mínimo 1s) {  
        <mat-spinner ngSkipHydration></mat-spinner>  
    }
```

El bloque @loading acepta los mismos parámetros opcionales que el bloque @placeholder . En este caso, mostramos el componente giratorio durante al menos 1 segundo.



Agregamos el atributo `ngSkipHydration` porque el componente spinner interactúa con el DOM del navegador y no se puede hidratar.

4. Si ejecutamos la aplicación usando el comando `ng serve`, deberíamos ver una indicación de giro durante 1 segundo mientras se carga el componente destacado.

El bloque `@error` en las vistas diferibles funciona de manera similar a los bloques `@placeholder` y `@loading`.

El contenido HTML dentro de él será visible cuando ocurra un error al cargar el bloque `@defer` contenido:

```
@defer() {  
  <app-featured />  
}  
} @placeholder(mínimo 1s) {  
    
}  
} @error() {  
  Se produjo un error al cargar el producto destacado.  
}
```

Como hemos visto, el contenido de un bloque `@defer` empieza a cargarse inmediatamente al renderizarse el componente al que pertenece. Sin embargo, la API de vistas diferibles nos proporciona herramientas ergonómicas para controlar cuándo se cargará el bloque, como veremos en la siguiente sección.

## Cargando patrones en bloques `@defer`

Usando activadores y mecanismos de precarga, podemos controlar cómo y cuándo se cargará un bloque `@defer`:

- Los activadores definen cuándo empiezan a cargarse los contenidos del bloque.
- Prefetch define si Angular buscará el contenido de antemano para que sea disponible cuando sea necesario

Podemos definir un disparador como un parámetro opcional dentro del bloque `@defer` usando la palabra clave `on` y el nombre del disparador:

```
@defer(en la ventana gráfica) {  
  <app-featured />  
}  
} @placeholder(mínimo 1s) {  
    
}  
} @error()
```

Se produjo un error al cargar el producto destacado.  
}

El marco Angular contiene los siguientes activadores integrados:

- ventana gráfica: Esto activará el bloqueo cuando el contenido ingrese a la ventana gráfica del navegador, lo que es la parte del navegador que está visible actualmente.



Puede obtener más información sobre la ventana gráfica en <https://developer.mozilla.org/docs/Glosario/Viewport>.

- Interacción: Esto activará el bloqueo cuando el usuario interactúe con el contenido.
- pasar el cursor: Esto activará el bloqueo cuando los usuarios pasen el cursor sobre el área cubierta por el contenido con su ratón.
- Inactivo: Esto activará el bloqueo cuando el navegador entre en estado inactivo , que es el comportamiento predeterminado de las vistas diferibles. El estado inactivo del navegador se activa mediante la API nativa `requestIdleCallback` .



Puede obtener más información sobre el estado inactivo en <https://developer.mozilla.org/docs/Web/API/Window/requestIdleCallback>.

- inmediato: esto activará el bloqueo cuando el cliente represente la página.



**La diferencia entre no usar el bloqueo y usarlo con la `inmediatez`**  
El desencadenante es que nos beneficiamos de las funciones de división de código de las vistas diferibles y entregamos menos JavaScript al cliente.

- Temporizador: Activará el bloqueo después de un tiempo especificado. La duración es obligatoria. parámetro de la función del temporizador :

```
@defer(en el temporizador(2s)) {
  <app-featured />
}
```

- El fragmento anterior comenzará a cargar el componente destacado después de 2 segundos.

Podemos lograr una mejor granularidad de carga combinando activadores:

```
@defer(en temporizador(2s); en inactivo) {  
  <app-featured />  
}
```

El fragmento anterior cargará el componente destacado cuando el navegador esté inactivo o después de 2 segundos.

Además de los activadores incorporados, podemos crear activadores personalizados nosotros mismos usando el `when` Palabra clave. La palabra clave `when` va seguida de una expresión que evalúa un valor booleano:

```
@defer(cuando isActive === verdadero) {  
  <app-featured />  
}
```

En el fragmento anterior, el componente destacado se cargará cuando la propiedad del componente `isActive` sea verdadera.

Los activadores en vistas diferibles son herramientas potentes y ergonómicas que ofrecen resultados increíbles en velocidad y rendimiento. Al combinarse con la precarga, pueden lograr importantes mejoras de rendimiento en aplicaciones Angular. La precarga permite especificar la condición en la que se puede precargar una vista diferible para que esté lista cuando se necesite. La precarga es compatible con todos los activadores integrados de vistas diferibles:

```
@defer(on timer(2s); precarga en inactivo) {  
  <app-featured />  
}
```

El fragmento anterior buscará previamente el contenido cuando el navegador esté inactivo y lo cargará después de 2 segundos. También puede definir cuándo se precargará el contenido usando la palabra clave " `when` " o crear activadores personalizados.

Los disparadores y la precarga nos permiten crear escenarios sofisticados y complejos para cargar vistas diferibles. La versatilidad de la API de vistas diferibles la convierte en una herramienta muy útil para desarrollar aplicaciones Angular altamente sofisticadas y de alto rendimiento.



Las vistas diferibles no se deben utilizar para contenido que debe mostrarse de inmediato.

En la siguiente sección, concluiremos nuestro viaje hacia la optimización del rendimiento de la aplicación con Angular SSG.

## Pre-renderizado de aplicaciones SSG

SSG o prerenderizado en tiempo de compilación es el proceso de crear archivos HTML estáticos para una aplicación Angular. Se realiza por defecto al compilar una aplicación SSR Angular con el comando ng build de la CLI de Angular.

La principal ventaja de una aplicación SSG es que no requiere tiempos de ida y vuelta entre el servidor y el cliente para cada solicitud. En su lugar, cada página se sirve como contenido estático, lo que elimina el tiempo de carga de la aplicación, medido por la métrica TTFB CWV.

En la sección Representación de aplicaciones SSR , la salida del comando de compilación de Angular CLI incluía el siguiente mensaje:

```
Se pre-renderizaron 4 rutas estáticas.
```

Veamos cómo funciona SSG y qué significa el resultado anterior:

1. Ejecute el siguiente comando para compilar la aplicación Angular:

```
construcción ng
```

2. El comando ng build creará la carpeta dist\my-app\browser .



La carpeta anterior no debe confundirse con la carpeta del navegador generada al crear una aplicación Angular que no sea SSR.

3. Navegue a la carpeta dist\my-app y abra el archivo prerendered-routes.json :

```
{  
  "rutas": [  
    "/carro",  
    "/productos",  
    "/productos/nuevos",  
    "/usuario"  
  ]  
}
```

Enumera las rutas de la aplicación que Angular SSG prerenderizó. También creó una carpeta .y el archivo index.html para cada ruta dentro de la carpeta del navegador.

4. Abra el archivo products\index.html y verá que Angular ha agregado todos los archivos CSS y HTML, e incluso ha renderizado los datos del producto tal como se obtuvieron de la API de Fake Store.
5. Para obtener una vista previa del funcionamiento de SSG, ejecute el comando ng serve para iniciar la aplicación y navegue a <http://localhost:4200/products>. La lista de productos se carga al instante, sin esperar a que la aplicación obtenga datos de la API de la Tienda Falsa.



El comando ng serve sirve la versión SSG de nuestra aplicación porque ejecuta el comando ng build internamente. Para deshabilitar SSG, abra el archivo angular.json. archivo y establezca la propiedad prerender en falso dentro de la sección de compilación .

SSG está habilitado por defecto en las aplicaciones SSR de Angular y puede mejorar drásticamente el tiempo de carga y el rendimiento en tiempo de ejecución. Resulta especialmente útil para dispositivos de gama baja con bajo rendimiento.

#### Resumen En este

capítulo, aprendimos diferentes maneras de optimizar y mejorar el rendimiento de una aplicación Angular. Presentamos el concepto de CWV y cómo puede afectar a una aplicación web.

Exploramos cómo medir y mejorar las métricas de CWV mediante SSR e hidratación en aplicaciones Angular. También investigamos diferentes aspectos de la optimización del rendimiento, como la directiva NgOptimizedImage y las vistas diferibles. Finalmente, vimos una descripción general de SSG en aplicaciones Angular.

Nuestro viaje con el framework Angular termina con este capítulo. Sin embargo, las posibilidades son infinitas. El framework Angular se actualiza con nuevas funciones en cada versión, ofreciendo a los desarrolladores web una potente herramienta para el desarrollo diario. Nos encantó contar con tu apoyo y esperamos que este libro te haya ayudado a ampliar tus ideas sobre lo que puedes lograr.

¡con una herramienta tan excelente!

## Únase a nosotros en Discord

Únase al espacio Discord de nuestra comunidad para discutir con el autor y otros lectores:

<https://packt.link/AprendizajeAngular5e>





[packt.com](http://packt.com)

Suscríbete a nuestra biblioteca digital en línea para acceder a más de 7000 libros y videos, además de herramientas líderes en la industria que te ayudarán a planificar tu desarrollo personal y a impulsar tu carrera profesional. Para más información, visita nuestro sitio web.

## ¿Por qué suscribirse?

- Dedique menos tiempo a aprender y más tiempo a codificar con libros electrónicos y videos prácticos de más de 4000 profesionales de la industria.
- Mejora tu aprendizaje con planes de habilidades diseñados especialmente para ti
- Obtenga un libro electrónico o un video gratuito cada mes
- Completamente buscable para un fácil acceso a información vital
- Copiar y pegar, imprimir y marcar contenido

En [www.packt.com](http://www.packt.com), También puede leer una colección de artículos técnicos gratuitos, suscribirse a una variedad de boletines gratuitos y recibir descuentos y ofertas exclusivas en libros y libros electrónicos de Packt.

Machine Translated by Google

## Otros libros

Puede que te guste

Si te ha gustado este libro, puede que te interesen estos otros libros de Packt:

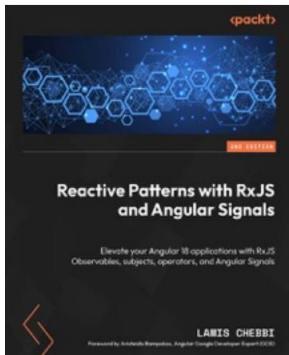


Angular efectivo

Roberto Heckers

ISBN: 978-1-80512-553-2

- Cree monorepositorios Nx listos para manejar cientos de aplicaciones Angular
- Reduzca la complejidad en Angular con la API independiente, la función de inyección, el flujo de control y Señales
  - Gestione eficazmente el estado de la aplicación utilizando señales, RxJS y NgRx
- Construir componentes dinámicos con proyección, TemplateRef y bloques de aplazamiento
- Realice pruebas unitarias y de extremo a extremo en Angular con Cypress y Jest
- Optimice el rendimiento de Angular, evite malas prácticas y automatice las implementaciones

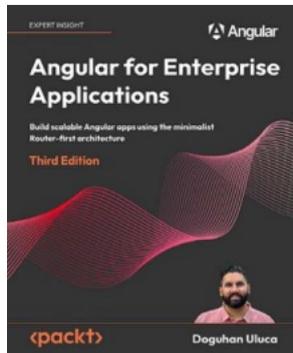


Patrones reactivos con RxJS y señales angulares

Lamis Chebbi

ISBN: 978-1-83508-770-1

- Familiarícese con los conceptos básicos de RxJS, como observables, sujetos y operadores.
  - Utilice el diagrama de canicas en patrones reactivos •
- Profundice en la manipulación de corrientes, incluida su transformación y combinación
- Comprender los problemas de fuga de memoria utilizando RxJS y las mejores prácticas para evitarlos
  - Construya patrones reactivos utilizando señales angulares y RxJS
  - Explorar diferentes estrategias de prueba para aplicaciones RxJS
  - Descubra la multidifusión en RxJS y cómo puede resolver problemas complejos
  - Cree una aplicación Angular completa de forma reactiva utilizando las últimas funciones de RxJS y Angular



Angular para aplicaciones empresariales, tercera edición

Doguhan Uluca

ISBN: 978-1-80512-712-3

- Mejores prácticas para la arquitectura y dirección de proyectos empresariales
- Enfoque minimalista que prioriza el valor para la entrega de aplicaciones web
- Cómo funcionan los componentes independientes, los servicios, los proveedores, los módulos, la carga diferida y las directivas trabajar en Angular
- Administre la reactividad de los datos de su aplicación mediante señales o RxJS
- Gestión de estados para sus aplicaciones Angular con NgRx
- Ecosistema angular para crear y entregar aplicaciones empresariales
- Pruebas automatizadas y CI/CD para ofrecer aplicaciones de alta calidad
- Autenticación y autorización
- Creación de control de acceso basado en roles con REST y GraphQL

## Packt está buscando autores como tú

Si está interesado en convertirse en autor de Packt, visite [authors.packtpub.com](http://authors.packtpub.com) Postúlate hoy mismo.

Hemos trabajado con miles de desarrolladores y profesionales de la tecnología, como tú, para ayudarles a compartir sus conocimientos con la comunidad tecnológica global. Puedes presentar una solicitud general , postularte para un tema de actualidad específico para el que buscamos autores o enviar tu propia idea.

## Comparte tus pensamientos

Ahora que terminaste de leer "Aprendiendo Angular, Quinta Edición", ¡ nos encantaría conocer tu opinión! Si compraste el libro en Amazon, haz clic aquí para ir directamente a la reseña de Amazon. página para este libro y comparte tus comentarios o deja una reseña en el sitio donde lo compraste.

Su reseña es importante para nosotros y para la comunidad tecnológica y nos ayudará a garantizar que ofrecemos contenido de excelente calidad.

# Índice

## Símbolos

401 Error no autorizado

respondiendo a 315, 316

Decoradores de @Host

enlace de referencia 146

@Autodecoradores

enlace de referencia 146

## A

Compilación de Ahead of Time (AOT) 408

Aplicación angular

bootstrap 13, 14

construcción 406-408

construcción, para diferentes entornos 408-410

edificio, para objeto ventana 410, 411

componentes 13

despliegue 415

Estructura 12

sintaxis de plantilla 14-16

Angular CDK 321

Características de Angular

multiplataforma 4

incorporación 5

herramientas

4, 5 en todo el mundo, uso 5, 6

Angular CLI 1

comandos 8, 9

instalación 7, 8

nuevo proyecto, creación 9-11

requisitos previos

6 espacio de trabajo, configuración 6

Componente angular 13

creando 56

creando, con Angular CLI 58-60

intercomunicación 75

estructura 56-58

Componentes angulares, intercomunicación

datos, emitiendo a través de eventos personalizados 80,

81 datos, pasando con enlace de entrada 75-77

eventos, escuchando con enlace de salida 77-80

variables de referencia de plantilla 81, 82

Decorador angular 57

Inyección de dependencia angular (DI) 121-123

dependencias 129-132

Herramientas de desarrollo angular 2, 16-20

Directiva angular 65

Forma angular

reaccionando a cambios de estado 304, 305

estado, manipulando 303

estado, actualizando 303

Marco angular

comunidades en línea 5

- Cliente HTTP angular 189-191 datos,  
obtención 192-201 datos,  
modificación 202 utilizado  
para comunicar servicios de datos 188,  
189 utilizado para  
manejar datos CRUD 192
- Autenticación de cliente HTTP angular 214 con API de  
backend 214, 215
- Autorización de cliente HTTP angular 214
- Solicitudes HTTP 218-221
  - acceso de usuario 216-218
- Cliente HTTP Angular, modificación de datos de un  
nuevo producto, adición de precio de  
producto 203-207, actualización de producto  
207-210, eliminación de 210-213
- AngularJS 3
- Servicio de lenguaje angular 22, 23
  - Material angular 320, 321 instalación  
321-324
    - Componentes de la interfaz de usuario, añadiendo 324, 325
    - Componentes de interfaz de usuario, integrando 329
    - Componentes de interfaz de usuario, temas 325-329
- Enrutador angular 224, 225
- ruta base, especificando 226
  - componentes, renderizando 228
  - configurando 227, 228
  - habilitando, en aplicaciones Angular 226 rutas  
principales, configurando 228-232
- Servicio angular
- creando 124, 125
  - inyectando, en el constructor 126-128
  - inyectar palabra clave 128, 129
- Utilidades de prueba angular 365
- cualquier tipo 43
  - paquete de aplicaciones
    - Optimización del tamaño  
412-414, limitación 411, 412
- rutas de aplicación rutas
- integradas, usando 238, 239 navegar,  
imperativamente a 233-237 organizar 232, 233
  - enlaces de enrutador, estilo  
239, 240
- Método 112 de Array.push
- Método Array.slice 147
- tipo de matriz 42
- funciones de flecha 32, 33
- afirmación 364
- información asincrónica
- El infierno de las devoluciones de llamadas, el cambio de  
promesas 156-160, estrategias de  
manejo 156, observables 160-162
- Pruebas de métodos asincrónicos  
386
- Componentes de servicios  
asincrónicos , pruebas 378-380
- tubería
- asíncrona usando 174, 175
- B**
- componente de insignia 355
- insignias
- aplicando 355, 356
- tipo booleano 42
- Bootstrap 410,
- arranque 13
- Presupuestos
- 411 rutas integradas  
utilizando 238, 239
- do
- devolución de llamada infernal 158
  - patrón de devolución de llamada 156

- Estrategia de detección de cambios  
que decide entre 85 y 89
- rutas infantiles  
usado, para reutilizar componentes 245-247
- Herramientas de desarrollo de Chrome 419  
utilizado para medir el rendimiento de la aplicación de tienda electrónica 419, 420
- Sintaxis del proveedor de clase 419 del informe de experiencia del usuario de Chrome 147
- declaración de clase  
elementos 35
- arnés de componentes 383, 384 ciclo de vida de componentes 89, 90  
componentes secundarios, acceso 95, 96  
inicialización, realización 90, 91 cambios de enlace de entrada, detección 93-95 recursos, limpieza 91-93
- Componentes  
que aplazan 430  
pruebas 366-370  
pruebas, con servicios asincrónicos 378-380  
pruebas, con arnés de componentes 383-385 pruebas, con dependencias 370 pruebas, con entradas y salidas 380-383  
enlace de clase de plantilla de componente 71, 72 datos, visualización condicional 63-66 datos, visualización desde la clase de componente 62, 63 datos, obtención de 73, 74 representación de datos, control 63  
interacción con 60 iteración, a través de datos 66-69 carga 60, 61 enlace de estilo 72
- estilo 71  
comutación, a través de 69-71
- señales calculadas 113, 178-180 trabajo 180, 181
- patrón de inyección de constructor 127
- Métricas web básicas (CWV) 417, 418  
Disposición acumulada Shi (CLS) 418  
Interacción con la siguiente pintura (INP) 418  
Pintura con contenido más grande (LCP) 418  
medición, formas 419 enlace de referencia 419
- Crear Leer Actualizar Eliminar (CRUD) 191  
Manejo de datos en el cliente HTTP Angular 192
- Clases CSS  
validación global con 288-290
- Estilo CSS  
encapsulando 82-85  
tipos personalizados 43, 44  
validadores personalizados  
edificio 297-302
- D**
- datos  
manipulando, con tuberías 99 ordenando, con tuberías 106-109
- servicios de datos  
comunicándose con Angular  
Cliente HTTP 188, 189
- ruta predeterminada 238
- bloques diferibles  
usando 431-437
- vistas diferibles 263, 430
- Operación DELETE 189
- dependencias 122  
componentes, pruebas 370

- Reemplazando, con los servicios stub 371-375, probando 387-389
- método de dependencia que espionaje en 375-378
- Experiencia de desarrollador (DX) 3
- componente de diálogo configuración del diálogo de confirmación 353, 354, creación de datos 350-353, obtención de notificaciones de usuario 354, 355, visualización de 355
- directivas 99
- directivas de atributos 114
  - edificio 113, 114
  - componentes 113 datos dinámicos, visualización 114-118 enlace de propiedades 118-120 directivas estructurales 113
  - prueba 390-392
- mi
- EditorConfig 24, 25
- inyectores de elementos 132
- Referencia del elemento 116
- método de emisión 80
- Pruebas de extremo a extremo (E2E) 368
- Inyectores de entorno 132 enlace de eventos 73
- F**
- API 191 de tienda falsa
- enlace de referencia 191
  - ruta de reserva 228
- API de búsqueda
- enlace de referencia 188
- método de filtrado 162, 167
- Primeros formularios Contentful Paint (FCP) 418, en aplicaciones Angular validadores personalizados, creación 297-302 validación global, con CSS 288-290 entrada, validando 288 pruebas 392-394 validación, en formularios reactivos 294-297 validación, en formularios basados en plantillas 290-294
- errores de marco desmitificando 316-318
- parámetros de función 31, 32
- GRAMO
- genéricos 50, 51
- Operación GET 189
- Repositorio Git 7
- controlador de errores global creando 312-315
- H**
- API de backend HTTP estableciendo 191, 192
- Módulo de cliente HTTP 190
- Encabezados HTTP 218
- Clase 196 de HttpParams
- Errores de solicitud HTTP que detectan hidratación 308-312 423-425
- I**
- Enlace de referencia de la función de inyección 428-430 para optimizar la carga de imágenes 129

Injector 122	K
objetos de jerarquía de	
inyectores, transformándose en	Karma 365
Servicios angulares 151-153	
implementación de servicios, anulación 147-149	Registrador
servicios, provisión 149-151 usado, para	de teclas que implementa la
anular proveedores 147	función de evento keyUp 162-165
servicios de inyección 133	
dependencias, compartir a través de	Yo
componentes 133-137	
búsqueda de proveedor, restringir 145, 146	Pintura con contenido más grande (LCP) 418
inyectores raíz y de componentes 138, 139	ruta de carga lenta
componentes de espacio aislado, con múltiples	protegiendo 262, 263
instancias 139-144	carga diferida 259
enlace de entrada	Faro 419
usado, para pasar datos 75-77	Soporte de largo plazo (LTS) 6
Entorno de Desarrollo Integrado	METRO
(IDE) 2	
Interacción con la siguiente pintura (INP) 418	rutas principales
interceptores 219	configuración del método
interfaces 48	de mapa 228-232 162-167
interpolación 15	operador de mapa 165
Yo	función de comparación 364
Jazmín 363-365	Diseño de materiales 320
Funciones de JavaScript 28	Tema del ícono de material 24
funciones de flecha 32, 33	parche de mono 85
clases 35, 36	norte
parámetros de función 31, 32	navegación, con funciones avanzadas que
módulos 36, 37	mejoran la carga
coalescencia nula 34	diferida 252 259-262
encadenamiento opcional 33,	evitando, fuera de la ruta 254, 255 acceso a la
34 declaración de variable 28-30	ruta, controlando 252-254 datos de ruta,
jQuery 410	precarga 256-259 Node.js 6

- Coalescencia nula 34 tipo de número 42
- Oh
- observables
- creando 166
  - suscribiéndose 169-172
  - transformando 167-169
- corrientes observables 155
- observables, anulación de suscripción 172
  - tubería asíncrona, usando 174, 175 componente, destruyendo 172-174
- métodos de observación
- completa 189
  - error 189
  - próximo 189
- patrón de observador 160
- método onKeyPress de enlace unidireccional 267 119
- operadores 163
- encadenamiento opcional 33, 34
- enlace de salida
  - utilizado para escuchar eventos 77-80
- PAG
- Patrones de carga de PageSpeed
- Insights 419, en bloques @defer 437-439
- tuberías 99
  - construcción 106 tipos integrados 100-106
  - mecanismo de detección de cambios 112, 113
  - parámetros, pasar a 110-112 sintaxis 100 prueba 389
- usado, para manipular datos 99 usado, para ordenar datos 106-109
- Operación POST 189
- función predicada 383
- Aplicaciones web progresivas (PWA) 4 promesas 158 parámetro
  - rechazado 158 parámetro de resolución 158

enlace de propiedad 62

método provideHttpClient 190

Proporcionar sintaxis literal de objeto 147

proveedores
  - que anulan, en la jerarquía del inyector 147

pushState 226

Operación PUT 189

**Q**

parámetros de consulta
  - utilizados para filtrar datos 248-250

**R**

formas reactivas 266

Construyendo 271 constructor de formularios, usando 285-287 interactuando con 271-275 modificando, dinámicamente 278-285 anidando jerarquías de formularios, creando 276-278 validación 294-297

Programación reactiva 155 en Angular 162-165

Biblioteca ReactiveX 165

parámetros de reposo 31

composición rica 165

acceso a la ruta
  - controlando 252-254

- configuración de ruta 14  
datos de ruta  
  precarga 256-259  
pruebas de componentes  
  enrutados 395, 396  
parámetros de ruta 240  
  componentes, reutilización  
    con rutas secundarias 245-247  
  propiedades de entrada, vinculación a 250, 251  
parámetros de consulta, utilizados  
  para filtrar datos 248-250  
instantánea, toma de 247  
utilizados, para crear la página de detalles 240-245  
enrutador  
  Prueba de 395  
protectores de  
  enrutador Prueba de 398-400  
enlaces de enrutador  
  estilo 239, 240  
resolutores de enrutador  
  Pruebas 401-403  
Prueba de componentes de  
  enrutamiento  
397 errores de tiempo de ejecución, en el controlador  
  de errores globales de la aplicación Angular, creación  
  312-315 manejo 308  
  Errores de solicitud HTTP, capturando 308-312 en  
  respuesta, a error 401 no autorizado 315, 316  
  
paquete rxjs-interop 182  
Biblioteca RxJS 155, 165  
  cooperando con 182-184 observables,  
  creando 166 observables,  
  transformando 167-169
- S**
- esquemas 6  
Optimización de motores de búsqueda (SEO) 422  
Separación de preocupaciones (SoC) 121  
Representación del lado del servidor (SSR) 3, 10, 422  
  anulación, en aplicaciones angulares 425-427  
servicio dentro de un servicio 140  
servicios  
  pruebas 385, 386  
  pruebas, con dependencias 387-389  
Limitación del alcance del servicio  
  138 configuración funcionalidad 364  
Sombra DOM 82  
componentes basados en señales 178  
señales 177, 178  
Solicitudes de una sola página (SPA) 225  
Patrón de Responsabilidad Única (SRP) Tubería de 36  
rebanadas 100  
componente snackbar 357 aplicando  
  357, 358  
método de ordenación 108  
parámetro de propagación 30  
espionaje 370  
Prerenderizado de  
  aplicaciones SSG 440, 441  
Aplicaciones de SSR  
  representación 422-425  
Generación de sitios estáticos (SSG) 10  
tipo de cadena 42  
talón  
  dependencia, reemplazando con 371-375  
stumping 370  
suscriptores 160

sentencia switch 69

métodos sincrónicos

prueba 387

## T

evento objetivo 74

propiedad objetivo 62

funcionalidad de desmontaje 364

formularios basados en plantillas 266

construcción 267-270

validación 290-294

expresión de plantilla 62

variable de entrada de plantilla 66

variables de referencia de plantilla 81, 82

declaración modelo 74

operador ternario 34

especificación de prueba 363

conjunto de pruebas 363

Tiempo hasta el primer byte (TTFB) 418

método de transformación

argumentos

107 valor 106

árbol sacudiendo 130

desencadenar 439

enlace bidireccional 267

fundición de tipos 50

Interfaces de TypeScript

37-39 48-50

tipos 41

trabajando con 39-41

Lenguaje TypeScript, interfaces, genéricos

50, 51, tipos de

utilidad 52

Lenguaje TypeScript, tipos cualquier

tipo 43 tipo array

42 tipo booleano

42 tipos de clases

45-48 tipos personalizados

43, 44 tipo de función 44,

45 tipo de número 42 tipo

de cadena 42

## Tú

Componentes de interfaz de usuario, tarjeta

Angular Material 341-343

fichas 337, 338

tabla de datos 344-350

controles de formulario 330

entrada 330-335

diseño 340

navegación 338-340

superposiciones

350 diálogo emergente

350 seleccionar 335-337

Pruebas unitarias 362, 365

Anatomía 363-365

Necesidad de 362, 363

Especificación de

prueba 363 Conjunto de pruebas 363

tipos de utilidad 52

## V

declaración de variables 28-30

enlace de referencia

de la ventana gráfica 438 438

vistas 56

Código de Visual Studio (VSCode) 2

Depurador de VSCode 20, 21

Perfiles de VSCode 22  
Servicio de lenguaje angular 22, 23  
EditorConfig 24, 25  
Tema del ícono de material 24

## O

modo reloj 370  
formularios web 265-267  
Web-Vitals 419  
ruta comodín 228, 238  
señales grabables 178-180  
trabajando 179

## Z

Biblioteca Zone.js 85, 313  
aplicaciones sin zona 178

## Descargue una copia gratuita en PDF de este libro

¡Gracias por comprar este libro!

¿Te gusta leer mientras viajas pero no puedes llevar tus libros impresos a todas partes?

¿Tu compra de libro electrónico no es compatible con el dispositivo que eliges?

No te preocunes, ahora con cada libro de Packt obtendrás una versión PDF de ese libro sin DRM sin costo.

Lee en cualquier lugar, en cualquier dispositivo. Busca, copia y pega código de tus libros técnicos favoritos directamente en tu aplicación.

Los beneficios no terminan ahí, puedes obtener acceso exclusivo a descuentos, boletines informativos y excelente contenido gratuito en tu bandeja de entrada todos los días.

Siga estos sencillos pasos para obtener los beneficios:

1. Escanea el código QR o visita el siguiente enlace:



<https://packt.link/libro-e-gratuito/9781835087480>

2. Envíe su comprobante de compra.

3. ¡Listo! Te enviaremos tu PDF gratuito y otros beneficios directamente a tu correo electrónico.



