

EXPERT INSIGHT

Learning Angular

A practical guide to building web applications
with modern Angular

Forewords by:

Bonnie Brennan

Founder of TechStackNation.com

Pablo Deeleman

Frontend Architect at GitKraken

Fifth Edition

Aristeidis Bampakos

<packt>



Aprendiendo Angular

Quinta edición

Una guía práctica para crear aplicaciones web con Angular moderno

Aristeidis Bampakos



Aprendiendo Angular

Quinta edición

Derechos de autor © 2024 Packt Publishing

Todos los derechos reservados. Ninguna parte de este libro podrá reproducirse, almacenarse en un sistema de recuperación de datos ni transmitirse en ninguna forma ni por ningún medio sin la autorización previa por escrito del editor, excepto en el caso de citas breves incluidas en artículos críticos o reseñas.

En la preparación de este libro se ha hecho todo lo posible para garantizar la precisión de la información presentada. Sin embargo, la información contenida en este libro se vende sin garantía, ni expresa ni implícita. Ni el autor, ni Packt Publishing ni sus distribuidores serán responsables de ningún daño causado o presuntamente causado, directa o indirectamente, por este libro.

Packt Publishing se ha esforzado por proporcionar información sobre las marcas comerciales de todas las empresas y productos mencionados en este libro mediante el uso adecuado de mayúsculas. Sin embargo, Packt Publishing no puede garantizar la exactitud de esta información.

Gerente de productos editoriales: Lucy Wan

Editora de Adquisiciones – Revisiones por Pares: Jane D'Souza

Editora del proyecto: Janice Gonsalves

Editora de desarrollo: Rebecca Youé

Editor de copias: Safis Editing

Editor técnico: Gaurav Gavas

Corrector: Safis Editing

Indexador: Hemangini Bari

Diseñador de presentaciones: Ajay Patule

Ejecutivo de marketing de relaciones con desarrolladores: Deepak Kumar

Primera publicación: abril de 2016

Segunda edición: diciembre de 2017

Tercera edición: septiembre de 2020

Cuarta edición: febrero de 2023

Quinta edición: diciembre de 2024

Referencia de producción: 1271224

Publicado por Packt Publishing Ltd.

Casa Grosvenor

11 Plaza de San Pablo

Birmingham

B3 1RB, Reino Unido.

ISBN 978-1-83508-748-0

www.packt.com

Colaboradores

Acerca del autor

Aristeidis Bampakos tiene más de 20 años de experiencia en la industria del desarrollo de software.

Actualmente trabaja como líder del equipo de desarrollo web en Plex-Earth, especializándose en el desarrollo de aplicaciones web con Angular. Comenzó su carrera como desarrollador de C# .NET, pero vio el potencial del desarrollo web y se dedicó a él a principios de 2011. Empezó a trabajar con AngularJS y en 2020 fue reconocido oficialmente como Google Developer Expert (GDE) para Angular.

A Aristeidis le apasiona ayudar a la comunidad de desarrolladores a aprender y crecer. Su pasión por la enseñanza lo ha llevado a ser un autor galardonado de los exitosos libros "Aprendiendo Angular" y "Proyectos Angular". Disfruta hablando sobre Angular en reuniones, conferencias y podcasts. Actualmente, también lidera el esfuerzo para que Angular sea accesible a la comunidad de desarrollo griega, manteniendo la traducción griega de código abierto de la versión oficial de Angular. documentación.

Este libro está dedicado a todas las personas alrededor del mundo que luchan con problemas de salud mental.

Acerca de los revisores

Thomas Laforgue es un padre casado que vive en los Alpes franceses. Es freelancer de Angular con más de 8 años de experiencia en el mundo frontend, especialmente en Angular. Ha sido Google Developer Expert (GDE) durante más de un año y es reconocido por su proyecto de código abierto, Angular Challenges. Este proyecto incluye más de 50 desafíos diseñados para ayudar a los desarrolladores a mejorar sus habilidades en Angular. Le apasiona la tecnología frontend y los proyectos de código abierto.

Fuera del trabajo, disfruta de los deportes y los juegos de mesa.

Martina Kraus ha estado activa en el mundo del desarrollo web desde sus inicios y, con el tiempo, se ha convertido en una experta en seguridad web. Como ingeniera de seguridad de aplicaciones, se centra en integrar las mejores prácticas de seguridad en todas las fases del desarrollo de software. Como Google Developer Expert (GDE) en Angular , le apasiona difundir conocimientos sobre Angular y seguridad web en congresos nacionales e internacionales. Organiza regularmente eventos ngGirls (talleres gratuitos de Angular para mujeres) y la conferencia alemana de Angular NG-DE. Actualmente trabaja en un libro titulado "Autorización y autenticación para desarrolladores web: una guía práctica", donde comparte sus conocimientos.

Prólogos

Querido lector,

El libro que tienes en tus manos continúa un viaje de conocimiento y descubrimiento que comenzó hace casi una década. Los orígenes de Learning Angular se remontan al verano de 2015. Durante ese tiempo, Packt Publishing, con quien había tenido varias conversaciones a lo largo de los años, me contactó para escribir un libro sobre cualquier tema de mi elección que resultara atractivo para la comunidad de desarrolladores web frontend.

En el verano de 2015, ya era bien sabido que el equipo de Angular en Google estaba trabajando en una nueva versión de su framework. Esta no era simplemente una continuación de lo que AngularJS había sido hasta entonces, sino una reescritura completa desde cero. AngularJS mostraba signos de envejecimiento y recibía críticas por su operatividad y rendimiento. En cambio, bibliotecas como React y Vue ganaban cada vez más aceptación, y su futuro parecía prometedor.

Angular se enfrentó al importante desafío de recuperar el corazón de los desarrolladores en una carrera en la que llegó tarde (quizás ya demasiado tarde).

Con solo esa idea en mente, la tarea de escribir un libro parecía abrumadora, agravada por la falta de documentación disponible. En el verano de 2015, Angular aún estaba en fase alfa, y la única manera de familiarizarse con la mecánica del framework era leer el blog oficial del equipo, que publicaba sus entradas poco a poco, o aplicar ingeniería inversa al código, que cambiaba radicalmente cada semana con cada nueva versión.

Abundaban las dudas: ¿Sería el libro resultante lo suficientemente preciso? ¿Tendría buena acogida entre el público dadas las expectativas creadas? ¿Resistiría el paso del tiempo? A pesar de estas preocupaciones, este era el momento crucial para escribir un libro sobre una tecnología frontend completamente nueva. Finalmente, la primera edición de Learning Angular 2 (que posteriormente eliminó el número de versión para pasar a llamarse simplemente Learning Angular) se publicó en mayo de 2016, tras mucho esfuerzo y más de dos docenas de reescrituras. Sinceramente, pensé que ese viaje terminaría ahí: se venderían como máximo un par de docenas de libros, recibiría algunas críticas positivas y probablemente muchas negativas.

Dudaba que Angular en sí mismo durara mucho más tiempo; a pesar de su arquitectura bellamente diseñada, el marco había llegado tarde a la fiesta y se basaba en principios que la comunidad tenía la intención de enterrar en favor de paradigmas de programación funcional.

Casi diez años después, me complace decir que mi juicio fue erróneo. El esfuerzo colectivo invertido en este libro ha permitido a miles de desarrolladores de todo el mundo crear proyectos maravillosos, contribuyendo a un mundo mejor y más accesible para todos. Aprender Angular 2 fue un éxito, y sus ediciones posteriores no lo han sido menos.

Mientras tanto, Angular ha seguido evolucionando y ha roto paradigmas en su constante búsqueda de la evolución. Desde señales hasta vistas diferibles, renderizado nativo del lado del servidor, herramientas de compilación ultrarrápidas mejoradas , una sintaxis renovada y una API de transición optimizada, junto con cientos de novedades importantes y menores, Angular ha demostrado un compromiso inigualable con la comunidad e influido en el futuro de nuestra industria. Justo después de su creación, Angular era considerado un patito feo en la industria. Ahora, es el nuevo cisne blanco el que, una vez más, marca el ritmo para el resto.

Sin embargo, esto plantea un gran desafío: ¿puede un libro capturar la grandeza de Angular, ayudar a los lectores a iniciarse con confianza en él y seguir siendo accesible y atractivo, todo ello mientras compite con la información completa sobre angular.dev? ¿Su sitio web oficial? La respuesta es sí, siempre y cuando Aristeidis Bampakos lidere esta iniciativa.

Aristeidis ha sido el motor del éxito de esta franquicia y le debo una gratitud infinita. Su perseverancia para satisfacer las expectativas de la comunidad, su enorme habilidad técnica para deconstruir conceptos complejos y su excelente capacidad narrativa son las razones por las que considero que el libro que ahora tienes en tus manos es una llave poderosa que te abrirá las puertas a un futuro fascinante para ti y para muchos otros.

Es un honor escribir este prólogo y un privilegio haber compartido este viaje con Aristeidis Bampakos y el equipo de Packt durante casi una década. El viaje no termina aquí. Ahora te toca a ti, querido lector, dar los siguientes pasos, y este libro será tu mejor guía.

Buen viaje.

Pablo Deeleman

Arquitecto de frontend en GitKraken y anterior autor de Learning Angular

Hola amigos,

Es un honor para mí presentar un libro excepcional escrito por uno de mis expertos favoritos en Angular, Aristeidis Bampakos. Es un autor consolidado de éxitos de ventas, un respetado Google Developer Expert en Angular, arquitecto empresarial principal y autor de código abierto. Con los años, Aris se ha convertido en una figura de confianza en la comunidad Angular, ya que no solo domina el framework, sino que también ha contribuido directamente a las traducciones y otras mejoras. Su dedicación al ecosistema Angular se refleja no solo en sus contribuciones, sino también en su pasión por ayudar a otros a desarrollar sus conocimientos y habilidades.

Para quienes buscan profundizar en su conocimiento de Angular, este libro ofrece una visión general completa y accesible del framework. Aris tiene una capacidad única para desglosar conceptos complejos en contenido digerible, lo que hace que el aprendizaje de Angular sea accesible y ameno para desarrolladores de todos los niveles. Tanto si estás empezando como si estás diseñando una aplicación de producción, esta guía sin duda te ayudará a profundizar en tu comprensión de Angular.

Además de sus contribuciones a Angular, Aris ha sido un líder muy querido e influyente en nuestra comunidad de Tech Stack Nation desde sus inicios. Sus contribuciones van más allá del código; aporta sabiduría, humildad y una auténtica pasión por compartir conocimientos. Los invito a asistir a uno de nuestros eventos en vivo, donde a menudo encontrarán a Aris compartiendo sus ideas, no solo como un brillante profesor y autor que siempre hace preguntas y aprende cosas nuevas, sino también como un amigo cariñoso y comprensivo para todos nosotros.

En el mundo en constante evolución de la tecnología de código abierto, los recursos en los que podemos confiar son cada vez más valiosos, y es aún más valioso contar con alguien como Aris para guiarnos en el cambiante panorama de Angular. Sin duda, este libro les resultará invaluable, como yo he considerado las contribuciones de Aris a nuestra comunidad a lo largo de los años.

¡Disfruta tu experiencia con Angular! Si tienes preguntas o comentarios después de leer, te recomiendo contactar a Aris; es muy amable. También puedes visitar Tech Stack Nation y preguntar por él; ¡seguro que le encantará conocerte!

Kilómetros de sonrisas,

Bonnie Brennan

Fundador de TechStackNation.com, arquitecto empresarial y Angular GDE

Únase a nosotros en Discord

Únase al espacio Discord de nuestra comunidad para discutir con el autor y otros lectores:

<https://packt.link/AprendizajeAngular5e>



Tabla de contenido

Prefacio	XIX
<hr/>	
Capítulo 1: Construyendo su primera aplicación Angular	1
<hr/>	
Requisitos técnicos	2
¿Qué es Angular?	2
¿Por qué elegir Angular?	4
Multiplataforma • 4	
Herramientas • 4	
Incorporación • 5	
El uso de Angular en todo el mundo • 5	
Configuración del espacio de trabajo de Angular CLI ...	
Prerrequisitos • 6	
Node.js • 6	
npm • 7	
Git • 7	
Instalación de Angular CLI • 7	
Comandos CLI • 8	
Creando un nuevo proyecto • 9	
La estructura de una aplicación Angular	12
Componentes • 13	
Arranque • 13	
Sintaxis de plantilla • 14	

Herramientas angulares	16
Herramientas de desarrollo angular • 16	
Depurador de VSCode • 20	
Perfiles de VSCode • 22	
Servicio de lenguaje angular • 22	
Tema de ícono de material • 24	
Configuración del editor • 24	
Resumen	25
Capítulo 2: Introducción a TypeScript	27
Requisitos técnicos	27
Fundamentos de JavaScript	28
Declaración de variables • 28	
Parámetros de función • 31	
Funciones de flecha • 32	
Encadenamiento opcional • 33	
Coalición nula • 34	
Clases • 35	
Módulos • 36	
¿Qué es TypeScript?	37
Introducción a TypeScript	39
Tipos • 41	
Cuerda • 42	
Booleano • 42	
Número • 42	
Matriz • 42	
cualquiera • 43	
Tipos personalizados • 43	
Funciones • 44	
Clases • 45	

Interfaces • 48	
Genéricos • 50	
Tipos de utilidades • 52	
Resumen • 52	
Capítulo 3: Estructuración de interfaces de usuario con componentes	55
Requisitos técnicos	55
Creando nuestro primer componente	56
La estructura de un componente Angular • 56	
Creación de componentes con Angular CLI • 58	
Interactuar con la plantilla	60
Cargando la plantilla del componente • 60	
Visualización de datos de la clase de componente • 62	
Control de la representación de datos • 63	
Vinculación de clases • 71	
Encuadernación de estilo • 72	
Obteniendo datos de la plantilla • 73	
Intercomunicación de componentes	75
Pasar datos mediante un enlace de entrada • 75	
Escucha de eventos mediante un enlace de salida • 77	
Emisión de datos a través de eventos personalizados • 80	
Variables de referencia locales en plantillas • 81	
Encapsulando estilos CSS	82
Decidir una estrategia de detección de cambios	85
Presentación del ciclo de vida de los componentes ... 89	
Realizar la inicialización de componentes • 90	
Limpieza de recursos de componentes • 91	
Detección de cambios en la vinculación de entrada • 93	
Acceso a componentes secundarios • 95	
Resumen	96

Capítulo 4: Enriquecimiento de aplicaciones mediante tuberías y directivas 99

Requisitos técnicos	99
Manipulación de datos con tuberías	99
Tuberías de construcción	106
Ordenar datos mediante tuberías • 106	
Pasando parámetros a las tuberías • 110	
Detección de cambios con tuberías • 112	
Directivas de construcción	113
Visualización de datos dinámicos • 114	
Vinculación de propiedades y respuesta a eventos • 118	
Resumen	120

Capítulo 5: Gestión de tareas complejas con servicios 121

Requisitos técnicos	122
Presentación de Angular DI	122
Creando nuestro primer servicio Angular	124
Inyectando servicios en el constructor • 126	
La palabra clave injectar • 128	
Proporcionar dependencias en toda la aplicación	129
Inyección de servicios en el árbol de componentes	133
Compartir dependencias a través de componentes • 133	
Inyectores de raíz y componentes • 138	
Componentes de sandbox con múltiples instancias • 139	
Restricción de la búsqueda de proveedores • 145	
Anulación de proveedores en la jerarquía de inyectores	147
Implementación del servicio de anulación • 147	
Prestación de servicios de forma condicional • 149	
Transformación de objetos en servicios angulares • 151	
Resumen	153

Capítulo 6: Patrones reactivos en Angular 155

Requisitos técnicos	155
Estrategias para el manejo de información asíncrona	156
Pasando del infierno de las devoluciones de llamadas a las promesas • 156	
Observables en pocas palabras • 160	
Programación reactiva en Angular	162
La biblioteca RxJS	165
Creación de observables • 166	
Transformación de observables • 167	
Suscribirse a observables	169
Darse de baja de observables	172
Destrucción de un componente • 172	
Uso de la tubería asíncrona • 174	

Capítulo 7: Seguimiento del estado de la aplicación con señales 177

Requisitos técnicos	177
Comprendiendo las señales	178
Señales de lectura y escritura	178
Señales calculadas	180
Cooperando con RxJS	182
Resumen	185

Capítulo 8: Comunicación con servicios de datos a través de HTTP 187

Requisitos técnicos	187
Comunicación de datos a través de HTTP	188
Presentamos el cliente HTTP Angular	... 189
Configuración de una API de backend	191
Manejo de datos CRUD en Angular	192
Obtención de datos a través de HTTP • 192	

Modificación de datos a través de HTTP • 202	
Añadiendo nuevos productos • 203	
Actualización del precio del producto • 207	
Eliminar un producto • 210	
Autenticación y autorización con HTTP	214
Autenticación con una API de backend • 214	
Autorización de acceso de usuarios • 216	
Autorización de solicitudes HTTP • 218	
Resumen	222
Capítulo 9: Navegación por aplicaciones con enrutamiento	223
Requisitos técnicos	224
Presentamos el enrutador Angular	224
Especificación de una ruta base • 226	
Habilitación del enrutamiento en aplicaciones Angular • 226	
Configuración del enrutador • 227	
Componentes de renderizado • 228	
Configuración de las rutas principales	228
Organización de rutas de aplicaciones	232
Navegar imperativamente hacia una ruta • 233	
Uso de rutas integradas • 238	
Enlaces de enrutador de estilo • 239	
Pasando parámetros a rutas	240
Creación de una página de detalles utilizando parámetros de ruta • 240	
Reutilización de componentes mediante rutas secundarias • 245	
Tomar una instantánea de los parámetros de ruta • 247	
Filtrado de datos mediante parámetros de consulta • 248	
Vinculación de propiedades de entrada a rutas • 250	
Mejorar la navegación con funciones avanzadas	252
Control de acceso a rutas • 252	
Cómo evitar la navegación fuera de una ruta • 254	

Tabla de contenido

Precarga de datos de ruta • 256	
Carga diferida de partes de la aplicación • 259	
Protección de una ruta de carga lenta • 262	
Resumen	263
Capítulo 10: Recopilación de datos de usuario con formularios	265
Requisitos técnicos	265
Presentación de formularios web	265
Creación de formularios basados en plantillas	267
Construyendo formas reactivas	271
Interactuando con formas reactivas • 271	
Creación de jerarquías de formularios anidados • 276	
Modificar formularios dinámicamente • 278	
Uso de un generador de formularios • 285	
Validación de entradas en formularios	288
Validación global con CSS • 288	
Validación en formularios basados en plantillas • 290	
Validación en formas reactivas • 294	
Creación de validadores personalizados • 297	
Manipulación del estado del formulario	303
Actualizando el estado del formulario • 303	
Reaccionando a los cambios de estado • 304	
Resumen	306
Únete a nosotros en Discord	306
Capítulo 11: Manejo de errores de aplicación	307
Requisitos técnicos	307
Manejo de errores de tiempo de ejecución	308
Detectar errores de solicitud HTTP • 308	
Creación de un controlador de errores global • 312	
Respondiendo al error 401 No autorizado • 315	

Desmitificando los errores del marco	316
Resumen	318
Capítulo 12: Introducción al material angular	319
Requisitos técnicos	319
Presentación de Material Design	320
Introducción a Angular Material	320
Instalación de material angular • 321	
Agregar componentes de interfaz de usuario • 324	
Tematización de componentes de interfaz de usuario • 325	
Integración de componentes de UI	329
Controles de formulario • 330	
Entrada • 330	
Seleccionar • 335	
Papas fritas • 337	
Navegación • 338	
Diseño • 340	
Tarjeta • 341	
Tabla de datos • 344	
Ventanas emergentes y superposiciones • 350	
Creación de un cuadro de diálogo de confirmación • 350	
Configuración de diálogos • 353	
Obtención de datos de los diálogos • 354	
Visualización de notificaciones de usuario • 355	
Resumen	359
Capítulo 13: Pruebas unitarias de aplicaciones angulares	361
Requisitos técnicos	362
¿Por qué necesitamos pruebas unitarias?	362
La anatomía de una prueba unitaria	363

Introducción a las pruebas unitarias en Angular	365
Componentes de prueba	366
Pruebas con dependencias • 370	
Reemplazar la dependencia con un stub • 371	
Espiando el método de dependencia • 375	
Prueba de servicios asincrónicos • 378	
Pruebas con entradas y salidas • 380	
Prueba con un arnés de componentes • 383	
Servicios de pruebas	385
Prueba de métodos sincrónicos/asincrónicos • 386	
Servicios de prueba con dependencias • 387	
Prueba de tuberías	389
Directivas de prueba	390
Formularios de prueba	392
Probando el enrutador	395
Componentes enrutados y de enrutamiento • 395	
Guardias • 398	
Resolvedores • 401	
Resumen	403
Capítulo 14: Llevar aplicaciones a producción	405
Requisitos técnicos	406
Construyendo una aplicación Angular	406
Edificios para diferentes entornos • 408	
Edificio para el objeto ventana • 410	
Limitar el tamaño del paquete de aplicaciones	411
Optimización del paquete de aplicaciones	412
Implementación de una aplicación Angular	415
Resumen	416

<u>Capítulo 15: Optimización del rendimiento de las aplicaciones</u>	417
Requisitos técnicos	418
Presentamos Core Web Vitals	418
Representación de aplicaciones SSR	422
Anulación de SSR en aplicaciones angulares • 425	
Optimización de la carga de imágenes	428
Aplazamiento de componentes	430
Presentamos las vistas diferibles • 430	
Uso de bloques diferibles • 431	
Cargando patrones en bloques @defer • 437	
Prerenderizado de aplicaciones SSG	440
Resumen	441
Otros libros que te pueden gustar	445
<u>Índice</u>	449

Prefacio

A medida que Angular se consolida como uno de los mejores frameworks de JavaScript, cada vez más desarrolladores buscan la mejor manera de comenzar con este framework extraordinariamente flexible y seguro. Aprender Angular, ahora en su quinta edición, te mostrará cómo usar Angular para lograr un alto rendimiento multiplataforma con las últimas técnicas web, una amplia integración con estándares web modernos y entornos de desarrollo integrados (IDE).

Este libro es especialmente útil para quienes se inician en Angular y les ayudará a familiarizarse con los fundamentos del framework necesario para empezar a desarrollar aplicaciones Angular. Aprenderán a desarrollar aplicaciones aprovechando el potencial de la interfaz de línea de comandos (CLI) de Angular, a escribir pruebas unitarias, a diseñar sus aplicaciones siguiendo las directrices de Material Design y, finalmente, a compilarlas para producción.

Actualizada para Angular 19, esta nueva edición incluye numerosas características y prácticas nuevas que abordan los desafíos actuales del desarrollo web frontend. Encontrarás nuevos capítulos dedicados a señales y optimización, así como más información sobre el manejo de errores y la depuración en Angular, y nuevos ejemplos prácticos . Al finalizar este libro, no solo podrás crear aplicaciones Angular con TypeScript desde cero, sino que también mejorarás tus habilidades de programación con las mejores prácticas.

Para quién es este libro

Este libro está dirigido a desarrolladores web que desean iniciarse en el desarrollo frontend y a desarrolladores frontend que desean ampliar sus conocimientos sobre frameworks de JavaScript. Necesitará experiencia previa con JavaScript, conocimientos básicos de la línea de comandos y familiaridad con el uso de IDE para comenzar con este libro.

Qué cubre este libro

Capítulo 1, Construyendo su primera aplicación Angular

En este capítulo, configuraremos el entorno de desarrollo instalando la CLI de Angular y aprenderemos a usar esquemas (comandos) para automatizar tareas como la generación de código y la creación de aplicaciones. Creamos una aplicación sencilla con la CLI de Angular y la compilamos. También conocemos algunas de las herramientas más útiles de Angular disponibles en Visual Studio Code.

Capítulo 2, Introducción a TypeScript

En este capítulo, aprenderemos qué es TypeScript, el lenguaje que se utiliza para crear aplicaciones Angular, y cuáles son los componentes básicos, como los tipos y las clases. Analizaremos algunos de los tipos avanzados disponibles y las características más recientes del lenguaje.

Capítulo 3, Estructuración de interfaces de usuario con componentes

En este capítulo, aprenderemos cómo se conecta un componente a su plantilla y cómo usar un decorador para configurarlo. Analizaremos cómo se comunican los componentes entre sí mediante el intercambio de datos entre ellos mediante enlaces de entrada y salida, y aprenderemos sobre las diferentes estrategias para detectar cambios en un componente. También aprenderemos a ejecutar lógica personalizada durante el ciclo de vida del componente.

Capítulo 4, Enriquecimiento de aplicaciones mediante tuberías y directivas

En este capítulo, analizamos las tuberías integradas de Angular y creamos nuestra propia tubería personalizada. Aprendemos a crear directivas y a aprovecharlas mediante una aplicación Angular que demuestra su uso.

Capítulo 5, Gestión de tareas complejas con servicios

En este capítulo, aprendemos cómo funciona el mecanismo de inyección de dependencia, cómo crear y usar servicios en componentes y cómo crear proveedores en una aplicación Angular.

Capítulo 6, Patrones reactivos en Angular

En este capítulo, aprenderemos qué es la programación reactiva y cómo podemos usar observables en una aplicación Angular mediante la biblioteca RxJS. También repasaremos todos los operadores RxJS comunes que se utilizan en una aplicación Angular.

Capítulo 7, Seguimiento del estado de la aplicación con señales

En este capítulo, aprenderemos los conceptos básicos de la API de Señales y su fundamento. Exploraremos cómo usar señales para monitorizar el estado de una aplicación Angular. También analizaremos la interoperabilidad de las señales con RxJS y cómo pueden funcionar conjuntamente en una aplicación de ejemplo.

Capítulo 8, Comunicación con servicios de datos a través de HTTP

En este capítulo, aprenderemos a interactuar con una API remota de backend y a realizar operaciones CRUD con datos en Angular. También investigaremos cómo configurar encabezados adicionales para una solicitud HTTP e interceptarla antes de enviarla o al finalizarla.

Capítulo 9, Navegación por aplicaciones con enrutamiento

En este capítulo, aprenderemos a usar el enrutador Angular para activar diferentes partes de una aplicación Angular. Descubriremos cómo pasar parámetros a través de la URL y cómo dividir una aplicación en rutas secundarias que puedan cargarse de forma diferida. También aprenderemos a proteger nuestros componentes y a preparar los datos antes de inicializarlos.

Capítulo 10, Recopilación de datos de usuario con formularios

En este capítulo, aprenderemos a usar formularios Angular para integrar formularios HTML en una aplicación y a configurarlos mediante FormGroup y FormControl. Monitoreamos la interacción del usuario en el formulario y validamos los campos de entrada.

Capítulo 11, Manejo de errores de aplicación

En este capítulo, aprendemos cómo manejar diferentes tipos de errores en una aplicación Angular y Aprenda acerca de los errores que provienen del propio marco.

Capítulo 12, Introducción al material angular

En este capítulo, aprenderemos a integrar las directrices de Google Material Design en una aplicación Angular mediante la biblioteca Angular Material, desarrollada por el equipo de Angular. Analizaremos algunos de los componentes principales de la biblioteca y cómo utilizarlos. Analizaremos los temas incluidos en la biblioteca y cómo instalarlos.

Capítulo 13, Pruebas unitarias de aplicaciones angulares

En este capítulo, aprendemos cómo probar artefactos angulares y anularlos en una prueba, cuáles son las diferentes partes de una prueba y qué partes de un componente deben probarse.

Capítulo 14, Llevar aplicaciones a producción

En este capítulo, aprendemos a usar la CLI de Angular para crear e implementar una aplicación Angular. Analizamos cómo pasar variables de entorno durante la compilación y cómo realizar optimizaciones de compilación antes de la implementación.

Capítulo 15, Optimización del rendimiento de las aplicaciones

En este capítulo, aprenderemos qué son las Core Web Vitals (CWV) y cómo afectan el rendimiento de una aplicación Angular. Exploraremos tres maneras diferentes de mejorar las métricas de CWV: cómo renderizar una aplicación del lado del servidor, cómo aprovechar la hidratación y cómo optimizar nuestras imágenes.

Para aprovechar al máximo este libro

Necesitará tener una versión de Angular 19 instalada en su computadora, preferiblemente la más reciente. Todos los ejemplos de código se han probado con Angular 19.0.0 en Windows, pero deberían funcionar también con cualquier versión futura de Angular 19.

Le recomendamos que escriba el código de este libro usted mismo o acceda a él a través del repositorio de GitHub (el enlace se encuentra en la siguiente sección). Esto le ayudará a evitar posibles errores al copiar y pegar código.

Descargue los archivos de código de ejemplo

Puede descargar los archivos de código de ejemplo para este libro desde su cuenta en <http://www.packtpub.com>.

Si compró este libro en otro lugar, puede visitar <http://www.packtpub.com/support> y regístrese para recibir los archivos directamente en su correo electrónico.

Puede descargar los archivos de código siguiendo estos pasos:

1. Inicie sesión o regístrese en <http://www.packtpub.com>.
2. Seleccione la pestaña SOPORTE .
3. Haga clic en Descargas de código y erratas.
4. Ingrese el nombre del libro en el cuadro de búsqueda y siga las instrucciones en pantalla.

Una vez descargado el archivo, asegúrese de descomprimir o extraer la carpeta utilizando la última versión de:

- WinRAR/7-Zip para Windows
- Zipeg/iZip/UnRarX para Mac
- 7-Zip/PeaZip para Linux

El paquete de código del libro también está alojado en GitHub en <https://github.com/PacktPublishing/Aprendiendo-Angular-Quinta-Edition>. También tenemos otros paquetes de códigos de nuestro rico catálogo de libros y videos disponibles en <https://github.com/PacktPublishing/>. ¡Échales un vistazo!

Descargar las imágenes en color

También proporcionamos un archivo PDF que tiene imágenes en color de las capturas de pantalla/diagramas utilizados en este libro.

Puedes descargarlo aquí: <https://packt.link/gbp/9781835087480>.

Convenciones utilizadas

A lo largo de este libro se utilizan varias convenciones textuales.

CodeInText: Indica palabras clave en el texto, nombres de tablas de bases de datos, nombres de carpetas, nombres de archivos, extensiones de archivo, rutas de acceso, URL ficticias, entradas del usuario y perfiles de redes sociales. Por ejemplo: "Monte el archivo de imagen de disco WebStorm-10*.dmg descargado como otro disco en su sistema".

Un bloque de código se establece de la siguiente manera:

```
[por defecto]
extender => s,1,Dial(Zap/1|30)
exten => s,2,Buzón de voz(u100)
exten => s,102,Buzón de voz(b100)
exten => i,1,Buzón de voz(s0)
```

Cuando deseamos llamar su atención sobre una parte particular de un bloque de código, las líneas relevantes o

Los elementos están en negrita:

```
[por defecto]
extender => s,1,Dial(Zap/1|30)
exten => s,2,Buzón de voz(u100)
exten => s,102,Buzón de voz(b100)
exten => i,1,Buzón de voz(s0)
```

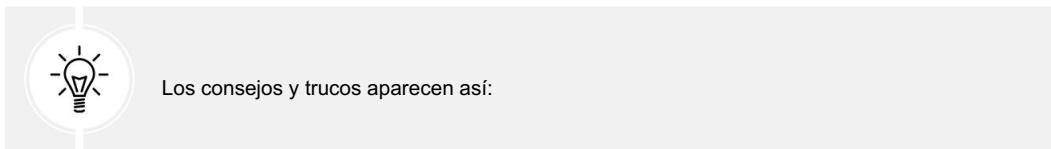
Cualquier entrada o salida de la línea de comandos se escribe de la siguiente manera:

```
# cp /usr/src/asterisk-addons/configs/cdr_mysql.conf.sample
/etc/asterisk/cdr_mysql.conf
```

Negrita: Indica un término nuevo, una palabra importante o palabras que aparecen en pantalla, por ejemplo, en menús o cuadros de diálogo. Por ejemplo: "Seleccionar Información del sistema en el panel de administración".



Las advertencias o notas importantes aparecen así.



Los consejos y trucos aparecen así:

Ponte en contacto con nosotros

Los comentarios de nuestros lectores siempre son bienvenidos.

Comentarios generales: Envíe un correo electrónico a feedback@packtpub.com e indique el título del libro en el asunto . Si tiene alguna pregunta sobre este libro, envíenos un correo electrónico a questions@packtpub.com.

Erratas: Aunque hemos tomado todas las precauciones para garantizar la precisión de nuestro contenido, pueden producirse errores. Si encuentra algún error en este libro, le agradeceríamos que nos lo comunicara. Visite <http://www.packtpub.com/submit-errata>. seleccionando su libro, haciendo clic en el enlace del Formulario de envío de erratas e ingresando los detalles.

Piratería: Si encuentra copias ilegales de nuestras obras en internet, en cualquier formato, le agradeceríamos que nos proporcionara la dirección o el nombre del sitio web. Por favor, contáctenos en copyright@packtpub.com con el enlace al material.

Si está interesado en convertirse en autor: si hay un tema en el que es experto y está interesado en escribir o contribuir a un libro, visite <http://authors.packtpub.com>.

Reseñas

Una vez que haya leído y usado este libro, ¿por qué no deja una reseña en el sitio donde lo compró? Los lectores potenciales podrán ver y usar su opinión imparcial para tomar decisiones de compra, en Packt podemos entender su opinión sobre nuestros productos y nuestros autores pueden ver sus comentarios sobre su libro. ¡Gracias!

Para obtener más información sobre Packt, visita packtpub.com.

Comparte tus pensamientos

Una vez que hayas leído "Aprendiendo Angular, quinta edición", ¡nos encantaría conocer tu opinión! Haz clic aquí para ir directamente a la página de reseñas de Amazon. para este libro y comparte tus comentarios.

Su reseña es importante para nosotros y para la comunidad tecnológica y nos ayudará a garantizar que ofrecemos contenido de excelente calidad.

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.

1

Construyendo tu primer Angular Solicitud

El desarrollo web ha experimentado un enorme crecimiento durante la última década. Han surgido frameworks, bibliotecas y herramientas que permiten a los desarrolladores crear excelentes aplicaciones web. Angular ha allanado el camino al crear un framework centrado en el rendimiento de las aplicaciones, la ergonomía del desarrollo y las técnicas web modernas.

Antes de desarrollar aplicaciones Angular, necesitamos aprender algunos aspectos básicos pero esenciales para tener una buena experiencia con el framework Angular. Uno de los aspectos principales que debemos saber es qué es Angular y por qué deberíamos usarlo para el desarrollo web. También faremos un recorrido por la historia de Angular para comprender su evolución.

Otro tema introductorio importante, aunque a veces desafiante, es la configuración de nuestro entorno de desarrollo. Debe realizarse al inicio del proyecto, y una correcta implementación temprana puede reducir la fricción a medida que nuestra aplicación crece. Por lo tanto, gran parte de este capítulo está dedicada a la CLI de Angular , una herramienta desarrollada por el equipo de Angular que proporciona herramientas de andamiaje y automatización en una aplicación Angular, eliminando el código repetitivo de configuración y permitiendo a los desarrolladores centrarse en el proceso de codificación. Usaremos la CLI de Angular para crear nuestra primera aplicación desde cero, familiarizarnos con la anatomía de una aplicación Angular y echar un vistazo a su funcionamiento. bajo el capó.

Trabajar en un proyecto Angular sin la ayuda de herramientas de desarrollo, como un Entorno de Desarrollo Integrado (IDE), puede ser tedioso. Nuestro editor de código favorito puede proporcionar un flujo de trabajo de desarrollo ágil que incluye compilación en tiempo de ejecución, verificación de tipos estáticos, introspección, completado de código y asistencia visual para depurar y compilar nuestra aplicación. En este capítulo, destacaremos algunas de las herramientas más populares del ecosistema Angular, como Angular DevTools y Visual Studio Code (VSCode).

En resumen, estos son los principales temas que exploraremos en este capítulo:

- ¿Qué es Angular?
- ¿Por qué elegir Angular?
- Configuración del espacio de trabajo de Angular CLI
- La estructura de una aplicación Angular
- Herramientas angulares

Requisitos técnicos

- GitHub: <https://github.com/PacktPublishing/Learning-Angular-Fifth-Edition/árbol/principal/ch01>
- Node.js: <https://nodejs.org>
- Git: <https://git-scm.com>
- VSCode: <https://code.visualstudio.com>
- Herramientas de desarrollo angular: <https://angular.dev/tools/devtools>

¿Qué es Angular?

Angular es un marco web escrito en el lenguaje TypeScript e incluye una CLI, un servicio de lenguaje, una herramienta de depuración y una amplia colección de bibliotecas propias.



Las bibliotecas incluidas en el marco Angular que se proporciona de manera predeterminada se denominan bibliotecas de origen.

Angular permite a los desarrolladores crear aplicaciones web escalables con TypeScript, un superconjunto sintáctico estricto de JavaScript, que aprenderemos en el Capítulo 2, Introducción a TypeScript.

La documentación oficial de Angular se puede encontrar en <https://angular.dev>.



La documentación oficial de Angular es el recurso más actualizado para el desarrollo con Angular. Es preferible usarla a otros recursos externos al desarrollar con Angular.

Google creó Angular. La primera versión, la 1.0, se lanzó en 2012 y se llamó AngularJS. AngularJS era un framework de JavaScript, y las aplicaciones web desarrolladas con él se escribían en JavaScript.

En 2016, el equipo de Angular decidió implementar un cambio revolucionario en AngularJS. Colaboraron con el equipo de TypeScript de Microsoft e introdujeron el lenguaje TypeScript en el framework. La siguiente versión del framework, la 2.0, se escribió en TypeScript y se renombró como Angular , con un logotipo diferente al de AngularJS.

En 2022, Angular entró en una nueva era de avances evolutivos conocida como el Renacimiento Angular. Durante ese período, el framework cobró impulso en el desarrollo web al introducir importantes innovaciones enfocadas en mejorar la Experiencia del Desarrollador (DX) y optimizar el rendimiento de las aplicaciones, como:

- Un enfoque simple y moderno para la creación de aplicaciones Angular
- Patrones de reactividad mejorados para gestionar el estado de la aplicación de manera eficiente
- La integración de técnicas de representación del lado del servidor (SSR) para mejorar el rendimiento

Un hito importante en la era del Renacimiento Angular fue Angular 17, cuando el equipo de Angular decidió cambiar la marca del marco con un nuevo logotipo y colores, reflejando los cambios y la configuración recientes. La visión para los avances futuros.



En este libro, abordaremos Angular 19, la última versión estable principal del framework Angular. AngularJS finalizó su ciclo de vida en 2022 y el equipo de Angular ya no le proporciona soporte ni mantenimiento.

Angular se basa en los estándares web más modernos y es compatible con todos los navegadores actuales.

Puede encontrar más detalles sobre la versión específica compatible con cada navegador en <https://angular.dev/reference/versiones#browser-support>.

En la siguiente sección, aprenderemos los beneficios de elegir Angular para el desarrollo web.

¿Por qué elegir Angular?

El poder del framework Angular se basa en la combinación de las siguientes características:

- Los pilares principales del marco:
 - Multiplataforma
 - Herramientas increíbles
 - Fácil incorporación
- El uso de Angular en todo el mundo:
 - Una comunidad increíble
 - Probado en batalla contra productos de Google

En las siguientes secciones examinaremos cada característica con más detalle.

Multiplataforma

Las aplicaciones Angular pueden ejecutarse en diferentes plataformas: web, servidor, escritorio y móvil. Angular solo puede ejecutarse de forma nativa en la web, ya que es un framework web; sin embargo, es de código abierto y cuenta con herramientas increíbles que permiten su ejecución en las tres plataformas restantes mediante las siguientes herramientas:

- Angular SSR: renderiza aplicaciones Angular del lado del servidor
- Trabajador de servicio angular: permite que las aplicaciones angulares se ejecuten como aplicaciones web progresivas (PWA) que pueden ejecutarse en entornos de escritorio y móviles nativos.
- Ionic/NativeScript: Nos permite crear aplicaciones móviles utilizando Angular

El siguiente pilar del marco describe las herramientas disponibles en el ecosistema Angular.

Estampación

El equipo de Angular ha creado dos excelentes herramientas que hacen que el desarrollo de Angular sea fácil y divertido:

- Angular CLI: Una interfaz de línea de comandos que nos permite trabajar con proyectos Angular,
 - Desde el andamiaje hasta las pruebas y la implementación
- Angular DevTools: una extensión del navegador que nos permite inspeccionar y perfilar Angular aplicaciones desde la comodidad de nuestro navegador

La CLI de Angular es la solución ideal para trabajar con aplicaciones Angular. Permite al desarrollador centrarse en escribir el código de la aplicación, eliminando las tareas repetitivas de configuración, como el andamiaje, la compilación, las pruebas y la implementación de una aplicación Angular.

Incorporación

Es simple y fácil comenzar con el desarrollo de Angular porque cuando instalamos Angular, también obtenemos una rica colección de bibliotecas propias listas para usar, que incluyen:

- Un cliente HTTP Angular para comunicarse con recursos externos a través de HTTP
- Formularios angulares para crear formularios HTML para recopilar información y datos de los usuarios.
- Un enrutador angular para realizar navegaciones dentro de la aplicación

Las bibliotecas anteriores se instalan por defecto al crear una nueva aplicación Angular mediante la CLI de Angular. Sin embargo, solo se utilizan en nuestra aplicación si las importamos explícitamente a nuestro proyecto.

El uso de Angular en todo el mundo

Muchas empresas usan Angular para sus sitios web y aplicaciones. El sitio web <https://www.hechoconangular.com> Contiene una lista extensa de dichas empresas, incluidas algunas populares.

Además, Angular se utiliza en miles de proyectos de Google y por millones de desarrolladores en todo el mundo. El hecho de que Angular ya se utilice internamente en Google es un factor crucial para la fiabilidad del framework. Cada nueva versión de Angular se prueba exhaustivamente en dichos proyectos antes de estar disponible para el público. El proceso de pruebas ayuda al equipo de Angular a detectar errores de forma temprana y a ofrecer un framework de alta calidad al resto de la comunidad de desarrolladores.

Angular cuenta con el respaldo y el apoyo de una próspera comunidad de desarrolladores. Los desarrolladores pueden acceder a numerosas comunidades disponibles en todo el mundo, en línea o localmente, para obtener ayuda y orientación sobre el framework Angular. Por otro lado, las comunidades contribuyen al progreso del framework Angular compartiendo comentarios sobre nuevas funciones, probando nuevas ideas e informando sobre problemas. Algunas de las comunidades en línea más populares son:

- Tech Stack Nation: El grupo de estudio de Angular más amigable del mundo, que reúne a desarrolladores de Angular apasionados por desarrollar su confianza en la creación de increíbles aplicaciones Angular. Tech Stack Nation es una comunidad donde los desarrolladores de Angular pueden colaborar, aprender de la experiencia de los demás y superar los límites de lo que Angular puede lograr. Puedes unirte a Tech Stack Nation en <https://techstacknation.com>.

- Discord de la Comunidad Angular: El servidor oficial de Discord de Angular que reúne a la increíble comunidad Angular. Todos pueden unirse a la comunidad con un solo clic . Es el punto de encuentro ideal para miembros del equipo Angular, Google Developer Experts (GDE), autores de bibliotecas, grupos de Meetup y cualquier persona interesada en aprender el framework. Puedes unirte al servidor de Discord de la Comunidad Angular en <https://discord.gg/angular>.
- Angularlove: una plataforma comunitaria para entusiastas de Angular, respaldada por House of Angular , para facilitar el crecimiento de los desarrolladores de Angular a través de iniciativas de intercambio de conocimientos. Comenzó como un blog donde expertos publicaban artículos sobre noticias, características y mejores prácticas de Angular. Ahora, Angular.love también organiza reuniones presenciales y en línea, con la participación frecuente de desarrolladores de Angular. Puedes unirte a Angular.love en <https://angular.love>.

Ahora que hemos visto qué es Angular y por qué alguien debería elegirlo para el desarrollo web, aprenderemos a usarlo y a crear excelentes aplicaciones web.

Configuración del espacio de trabajo de Angular CLI

Configurar un proyecto con Angular puede ser complicado. Es necesario saber qué bibliotecas importar y asegurarse de que los archivos se procesen en el orden correcto, lo que nos lleva al tema del andamiaje.

El andamiaje es una herramienta para automatizar tareas, como generar un proyecto desde cero, y se vuelve necesario a medida que crece la complejidad y cada hora cuenta para producir valor comercial, en lugar de gastarse luchando contra problemas de configuración.

La principal motivación para crear la CLI de Angular fue ayudar a los desarrolladores a centrarse en la creación de aplicaciones, eliminando la configuración repetitiva. En esencia, con un simple comando, debería ser posible inicializar una aplicación, agregar nuevos artefactos, ejecutar pruebas, actualizar aplicaciones y crear un paquete de producción.

La CLI de Angular facilita todo esto mediante comandos especiales.

Mandos llamados esquemas.

Prerrequisitos

Antes de comenzar, debemos asegurarnos de que nuestro entorno de desarrollo incluya herramientas de software esenciales para el flujo de trabajo de desarrollo de Angular.

Node.js

Node.js es un entorno de ejecución de JavaScript basado en el motor JavaScript v8 de Chrome. Angular requiere una versión de soporte de larga duración (LTS) activa o en mantenimiento . Si ya lo tienes instalado, puedes ejecutar node -v en la línea de comandos para comprobar qué versión estás ejecutando.



Si necesitas trabajar con aplicaciones que usan diferentes versiones de Node.js o no puedes instalar el entorno de ejecución debido a permisos restringidos, use nvm, un administrador de versiones para Node.js diseñado para que cada usuario lo instale. Puede obtener más información en <https://github.com/nvm-sh/nvm>.

npm

npm es un gestor de paquetes de software incluido por defecto en Node.js. Puedes comprobarlo ejecutando npm -v en la línea de comandos. Una aplicación Angular consta de varias bibliotecas, llamadas paquetes, que se encuentran en un lugar central llamado registro npm. El cliente npm descarga e instala las bibliotecas necesarias para ejecutar la aplicación desde el registro npm a tu ordenador local.

Git

Git es un cliente que nos permite conectarnos a sistemas de control de versiones distribuidos, como GitHub, Bitbucket y GitLab. Es opcional desde la perspectiva de la CLI de Angular. Debes instalarlo si quieres subir tu proyecto de Angular a un repositorio de Git, lo cual podría ser conveniente.

Instalación de Angular CLI

La CLI de Angular forma parte del ecosistema Angular y se puede descargar desde el registro de paquetes de npm. Dado que se utiliza para crear proyectos Angular, debemos instalarla globalmente en nuestro sistema. Abra una ventana de terminal y ejecute el siguiente comando:

```
npm install -g @angular/cli
```



Es posible que necesite permisos elevados en algunos sistemas Windows, por lo que debería ejecutar su terminal como administrador. Ejecute el comando anterior en sistemas Linux/MacOS añadiendo la palabra clave sudo como prefijo para ejecutarlo con privilegios administrativos.

El comando que usamos para instalar Angular CLI usa el cliente npm , seguido de un conjunto de argumentos de tiempo de ejecución:

- instalar o i: Indica la instalación de un paquete
- -g o --global: Indica que el paquete se instalará en el sistema globalmente
- @angular/cli: El nombre del paquete a instalar

La CLI de Angular usa la misma versión que el framework Angular, que en este libro es la 19. El comando anterior instalará la última versión estable de la CLI de Angular. Puede comprobar qué versión tiene instalada ejecutando `ng version` o `ng v` en la línea de comandos. Si tiene una versión diferente a la 19 después de instalarla, puede ejecutar el siguiente comando:

```
npm install -g @angular/cli@19
```

El comando anterior buscará e instalará la última versión de Angular CLI 19.

Comandos CLI

La CLI de Angular es una herramienta de interfaz de línea de comandos que automatiza tareas específicas durante el desarrollo, como servir, compilar, agrupar, actualizar y probar un proyecto Angular. Como su nombre indica, utiliza la línea de comandos para invocar el archivo ejecutable `ng` y ejecutar comandos con la siguiente sintaxis:

```
ng [comando] [opciones]
```

Aquí, `[comando]` es el nombre del comando que se ejecutará y `[opciones]` indica los parámetros adicionales que se pueden pasar a cada comando. Para ver todos los comandos disponibles, puede ejecutar lo siguiente:

```
ng ayuda
```

Algunos comandos pueden invocarse usando un alias en lugar del nombre. En este libro, explicamos los más comunes (el alias de cada comando se muestra entre paréntesis):

- `nuevo (n)`: crea un nuevo espacio de trabajo de Angular CLI desde cero
- `build (b)`: compila una aplicación Angular y genera archivos generados en un formato predefinido. carpeta
- `generar (g)`: crea nuevos archivos que componen una aplicación Angular
- `servir (dev)`: crea una aplicación Angular y la sirve utilizando un servidor web preconfigurado
- `prueba (t)`: ejecuta las pruebas unitarias de una aplicación Angular
- agregar: Instala una biblioteca Angular en una aplicación Angular
- actualización: actualiza una aplicación Angular a la última versión de Angular

Puede encontrar más comandos CLI de Angular en <https://angular.dev/cli>.

Actualizar una aplicación Angular es una de las tareas más importantes de la lista anterior. Nos ayuda a mantenernos al día actualizando nuestras aplicaciones Angular a la última versión.



Intente mantener sus proyectos Angular actualizados porque cada nueva versión de Angular viene repleta de nuevas características interesantes, mejoras de rendimiento y correcciones de errores.

Además, puedes utilizar la guía de actualización de Angular, que contiene consejos e instrucciones paso a paso sobre cómo actualizar tus aplicaciones, en <https://angular.dev/update-guide>.

Creando un nuevo proyecto

Ahora que hemos preparado nuestro entorno de desarrollo, podemos empezar a crear nuestra primera aplicación Angular. Usaremos el comando `ng new` de la CLI de Angular y pasaremos el nombre de la aplicación que queremos crear como opción:

1. Abra una ventana de terminal, navegue a la carpeta que desee y ejecute el comando `ng new my-app`. Crear una nueva aplicación Angular es un proceso sencillo. La CLI de Angular solicitará detalles sobre la aplicación que queremos crear para que pueda estructurar el proyecto Angular de la mejor manera posible.
2. Inicialmente, nos preguntará si queremos habilitar el análisis de Angular:

¿Desea compartir datos de uso seudónimos sobre este proyecto con el equipo de Angular de Google según la Política de Privacidad de Google en <https://policies.google.com/privacy>? Para obtener más información y saber cómo cambiar esta configuración, consulte <https://angular.dev/cli/analytics>. (sí/no)

La CLI de Angular hará esta pregunta una vez al crear el primer proyecto de Angular y aplicarla globalmente en nuestro sistema. Sin embargo, podemos cambiar la configuración posteriormente en un espacio de trabajo específico de Angular.

3. La siguiente pregunta está relacionada con el estilo de nuestra aplicación:

¿Qué formato de hoja de estilo le gustaría utilizar?

Es común usar CSS para dar estilo a las aplicaciones Angular. Sin embargo, podemos usar preprocesadores como SCSS o Less para enriquecer nuestro flujo de trabajo de desarrollo. En este libro, trabajamos directamente con CSS, así que acepte la opción predeterminada, CSS, y presione Enter.

4. Finalmente, la CLI de Angular nos preguntará si queremos habilitar SSR y la generación de sitios estáticos (SSG) en nuestra aplicación:

¿Desea habilitar la representación del lado del servidor (SSR) y la generación de sitios estáticos (SSG/Prerenderización)? (sí/no)

SSR y SSG se encargan de mejorar el rendimiento de inicio y carga de una aplicación Angular.

Aprenderemos más sobre ellos en el Capítulo 15, Optimización del Rendimiento de Aplicaciones. Por ahora, acepte la opción predeterminada, "No", pulsando Intro.

El proceso puede tardar un tiempo, dependiendo de tu conexión a internet. Durante este tiempo, la CLI de Angular descarga e instala todos los paquetes necesarios y crea los archivos predeterminados para tu aplicación Angular. Al finalizar, se creará una carpeta llamada my-app. Esta carpeta representa un espacio de trabajo de la CLI de Angular que contiene una única aplicación Angular llamada my-app en el nivel raíz.

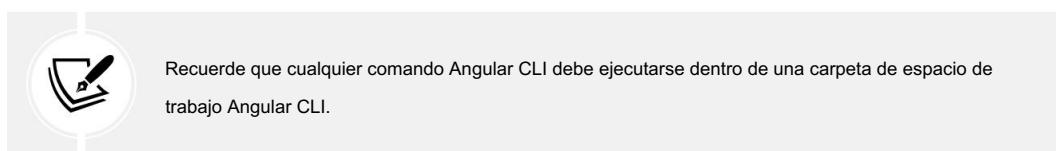
El espacio de trabajo contiene varias carpetas y archivos de configuración que la CLI de Angular necesita para compilar y probar la aplicación Angular:

- .vscode: incluye archivos de configuración de VSCode
- node_modules: incluye paquetes npm instalados que son necesarios para desarrollar y ejecutar el Aplicación angular
- público: contiene recursos estáticos como fuentes, imágenes e íconos
 - src: Contiene los archivos fuente de la aplicación
- .editorconfig: define estilos de codificación para el editor predeterminado
- .gitignore: especifica los archivos y carpetas que Git no debe rastrear
- angular.json: El archivo de configuración principal del espacio de trabajo de Angular CLI
- package.json y package-lock.json: proporcionan definiciones de paquetes npm, junto con sus versiones exactas, que son necesarias para desarrollar, probar y ejecutar la aplicación Angular
- README.md: un archivo README que se genera automáticamente desde la CLI de Angular
- tsconfig.app.json: una configuración de TypeScript específica para la aplicación Angular
- tsconfig.json: una configuración de TypeScript específica del espacio de trabajo de Angular CLI
- tsconfig.spec.json: una configuración de TypeScript específica para las pruebas unitarias de la aplicación Angular

Como desarrolladores, solo deberíamos preocuparnos por escribir el código fuente que implementa las funcionalidades de nuestra aplicación. Sin embargo, tener conocimientos básicos de cómo se orquesta y configura la aplicación nos ayuda a comprender mejor su mecánica y cómo intervenir si es necesario.

Navegue a la carpeta recién creada e inicie su aplicación con el siguiente comando:

```
ng servir
```



La CLI de Angular compila el proyecto de Angular e inicia un servidor web que monitorea los cambios en los archivos del proyecto. De esta forma, cada vez que se modifica el código de la aplicación, el servidor web reconstruye el proyecto para reflejar los nuevos cambios.

Una vez completada con éxito la compilación, puede obtener una vista previa de la aplicación abriendo su navegador y navegando a <http://localhost:4200>:

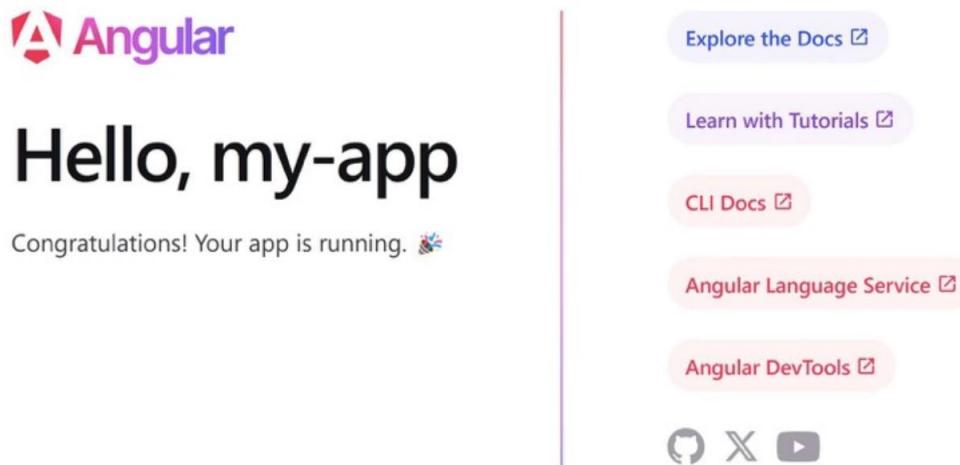


Figura 1.1: Página de inicio de la aplicación Angular

¡Felicitaciones! Has creado tu primer espacio de trabajo de Angular CLI. Angular CLI creó una página web de ejemplo que podemos usar como referencia para crear nuestro proyecto. En la siguiente sección, exploraremos las partes principales de nuestra aplicación y aprenderemos a modificar esta página.

La estructura de una aplicación Angular

Daremos los primeros pasos intrépidos al examinar nuestra aplicación Angular. La CLI de Angular ya ha estructurado nuestro proyecto y ha hecho gran parte del trabajo pesado por nosotros. Solo necesitamos iniciar nuestro IDE favorito y empezar a trabajar con el proyecto Angular. Usaremos VSCode en este libro, pero siéntete libre de elegir cualquier editor con el que te sientas cómodo:

1. Abra VSCode y seleccione Archivo | Abrir carpeta... en el menú principal.
2. Dirígete a la carpeta my-app y seleccionala. VSCode cargará la CLI de Angular asociada.
espacio de trabajo.
3. Expanda la carpeta src .

Al desarrollar una aplicación Angular, es probable que interactuemos con la carpeta src . Es donde escribimos el código y las pruebas de nuestra aplicación. Contiene lo siguiente:

- app: Todos los archivos de la aplicación relacionados con Angular. Interactúas con esta carpeta la mayor parte del tiempo durante el desarrollo.
- index.html: La página HTML principal de la aplicación Angular.
- main.ts: el punto de entrada principal de la aplicación Angular.
- style.css: Estilos CSS que se aplican globalmente a la aplicación Angular. La extensión de este archivo depende del formato de hoja de estilos seleccionado al crear la aplicación.

La carpeta de la aplicación contiene el código fuente que escribimos para nuestra aplicación. Los desarrolladores pasan la mayor parte del tiempo en ella. La aplicación Angular, creada automáticamente desde la CLI de Angular, contiene los siguientes archivos:

- app.component.css: Contiene estilos CSS específicos de la página de ejemplo. La extensión de este archivo depende del formato de hoja de estilo que elija al crear la aplicación.
- app.component.html: contiene el contenido HTML de la página de muestra.
- app.component.spec.ts: contiene pruebas unitarias para la página de muestra.
- app.component.ts: define la lógica de presentación de la página de muestra.
- app.config.ts: define la configuración de la aplicación Angular.
- app.routes.ts: define la configuración de enrutamiento de la aplicación Angular.



La extensión de nombre de archivo .ts se refiere a archivos TypeScript.

En las siguientes secciones, aprenderemos cómo Angular orquesta algunos de esos archivos para mostrar la página de muestra de la aplicación.

Componentes

Los archivos cuyos nombres empiezan por app.component constituyen un componente de Angular. Un componente en Angular controla parte de una página web orquestando la interacción de la lógica de presentación con el contenido HTML de la página, denominado plantilla.

Cada aplicación Angular tiene un archivo HTML principal, llamado index.html, que existe dentro del src carpeta y contiene el siguiente elemento HTML <body> :

```
<cuerpo>
  <app-root></app-root>
</cuerpo>
```

La etiqueta <app-root> se utiliza para identificar el componente principal de la aplicación y actúa como contenedor para mostrar su contenido HTML. Indica a Angular que renderice la plantilla del componente principal dentro de esa etiqueta. Aprenderemos cómo funciona en el Capítulo 3, "Estructura de interfaces de usuario con componentes".

Cuando la CLI de Angular crea una aplicación Angular, analiza el archivo index.html e identifica las etiquetas HTML dentro del elemento <body> . Una aplicación Angular siempre se renderiza dentro del elemento <body> y se compone de un árbol de componentes. Cuando la CLI de Angular encuentra una etiqueta que no es un elemento HTML conocido, como <app-root>, comienza a buscar en los componentes del árbol de la aplicación. Pero ¿cómo sabe por dónde empezar?

Arranque

El método de inicio de una aplicación Angular se llama bootstrapping y se define en el Archivo main.ts dentro de la carpeta src :

```
importar { bootstrapApplication } desde '@angular/platform-browser';
importar { appConfig } desde './app/app.config';
importar { AppComponent } desde './app/app.component';
```

```
bootstrapApplication(Componente de la aplicación, Configuración de la aplicación)
    .catch((err) => console.error(err));
```

La función principal del archivo de arranque es definir el componente que se cargará al iniciar la aplicación. Llama al método bootstrapApplication y pasa AppComponent como parámetro para especificar el componente inicial de la aplicación. También pasa el objeto appConfig como segundo parámetro para especificar la configuración que se utilizará al iniciar la aplicación. La configuración de la aplicación se describe 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: [provideZoneChangeDetection({ eventCoalescing: true }),
    proporcionarRouter(rutas)]
};
```

El objeto appConfig contiene una propiedad "proveedores" para definir los servicios proporcionados en toda la aplicación Angular. Aprenderemos más sobre los servicios en el Capítulo 5, "Gestión de tareas complejas". con Servicios.

Una nueva aplicación Angular CLI proporciona servicios de enrutamiento por defecto. El enrutamiento está relacionado con la navegación dentro de la aplicación entre diferentes componentes mediante la URL del navegador. Se activa mediante el método provideRouter , pasando un objeto de rutas , llamado configuración de ruta, como parámetro. La configuración de ruta de la aplicación se define en el archivo app.routes.ts :

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

exportar const rutas: Rutas = [];
```

Nuestra aplicación aún no tiene una configuración de ruta, como lo indica la matriz de rutas vacía . Aprenderemos cómo configurar el enrutamiento y configurarlo en el Capítulo 9, Navegación por aplicaciones con enrutamiento.

Sintaxis de la plantilla

Ahora que hemos realizado una breve descripción general de nuestra aplicación de muestra, es hora de comenzar a interactuar. con el código fuente:

- Ejecute el siguiente comando en una ventana de terminal para iniciar la aplicación si no está ya corriendo:

```
ng servir
```



Si está trabajando con VSCode, es preferible utilizar su terminal integrada, a la que se puede acceder desde la opción Terminal | Nueva terminal en el Menú principal.

- Abra la aplicación con su navegador en `http://localhost:4200` y observe el texto debajo del logotipo de Angular: "Hola, mi-aplicación". La palabra "mi-aplicación", que corresponde al nombre de la aplicación, proviene de una variable declarada en el archivo TypeScript del componente principal. Abra el archivo `app.component.ts` y localice la variable del título :

```
importar { Componente } desde '@angular/core';
importar { RouterOutlet } desde '@angular/router';

@Component({
  selector: 'app-root',
  importaciones: [RouterOutlet],
  URL de plantilla: './app.component.html',
  URL de estilo: './app.component.css'
})
clase de exportación AppComponent {
  título = 'mi-aplicación';
}
```

La variable de título es una propiedad del componente que se utiliza en la plantilla del componente.

- Abra el archivo `app.component.html` y vaya a la línea 228:

```
Hola, {{ título }}
```

La propiedad `title` está entre llaves, una sintaxis llamada interpolación, que forma parte de la sintaxis de plantilla de Angular. En resumen, la interpolación convierte el valor de la propiedad `title` en texto y lo imprime en la página.

Angular utiliza una sintaxis de plantilla específica para ampliar y enriquecer la sintaxis HTML estándar en la plantilla de la aplicación. Aprenderemos más sobre la sintaxis de plantilla de Angular en el Capítulo 3, "Estructura de interfaces de usuario con componentes".

4. Cambie el valor de la propiedad de título en la clase AppComponent a World, guarde los cambios, espere a que la aplicación se vuelva a cargar y examine la salida en el navegador:



Figura 1.2: Título de la página de destino

¡Felicitaciones! Has interactuado con éxito con el código fuente de tu aplicación.

A estas alturas, ya deberías tener una comprensión básica de cómo funciona Angular y cuáles son los componentes básicos de una aplicación Angular. Como lector, has tenido que absorber mucha información hasta ahora. Sin embargo, tendrás la oportunidad de familiarizarte con los componentes en los próximos capítulos. Por ahora, el objetivo es ponerlo en funcionamiento, brindándole una herramienta poderosa como Angular CLI y mostrándole cómo solo se necesitan unos pocos pasos para mostrar una aplicación en la pantalla.

Herramientas angulares

Una de las razones por las que el framework Angular es popular entre los desarrolladores es su amplio ecosistema de herramientas disponibles. La comunidad Angular ha creado herramientas increíbles para completar y automatizar diversas tareas, como la depuración, la inspección y la creación de aplicaciones Angular:

- Herramientas de desarrollo angulares
- Depurador de VSCode
- Perfiles de VSCode

Aprenderemos cómo utilizar cada uno en las siguientes secciones, comenzando con Angular DevTools.

Herramientas de desarrollo angulares

Angular DevTools es una extensión de navegador creada y mantenida por el equipo de Angular. Permite inspeccionar y perfilar aplicaciones Angular directamente en el navegador. Actualmente es compatible con Google Chrome y Mozilla Firefox y se puede descargar desde las siguientes tiendas de navegadores:

- Google Chrome: <https://chrome.google.com/webstore/detail/angular-developer-herramientas/ienfalfjfdbdbebioblackkekamfmbnh>
- Mozilla Firefox: <https://addons.mozilla.org/firefox/addon/angular-devtools>

Para abrir la extensión, abra las herramientas para desarrolladores del navegador y seleccione la pestaña Angular . Contiene tres pestañas adicionales:

- Componentes: muestra el árbol de componentes de la aplicación Angular
- Profiler: Nos permite perfilar e inspeccionar la aplicación Angular
- Árbol de inyectores: muestra los servicios proporcionados por la aplicación Angular

En este capítulo, exploraremos cómo usar la pestaña Componentes . Aprenderemos a usar la pestaña Perfilador en el Capítulo 3, Estructura de interfaces de usuario con componentes, y la pestaña Árbol de inyectores en el Capítulo 5, Gestión de tareas complejas con servicios.

La pestaña Componentes permite previsualizar los componentes y directivas de una aplicación Angular e interactuar con ellos. Si seleccionamos un componente de la representación en árbol, podemos ver sus propiedades y metadatos:

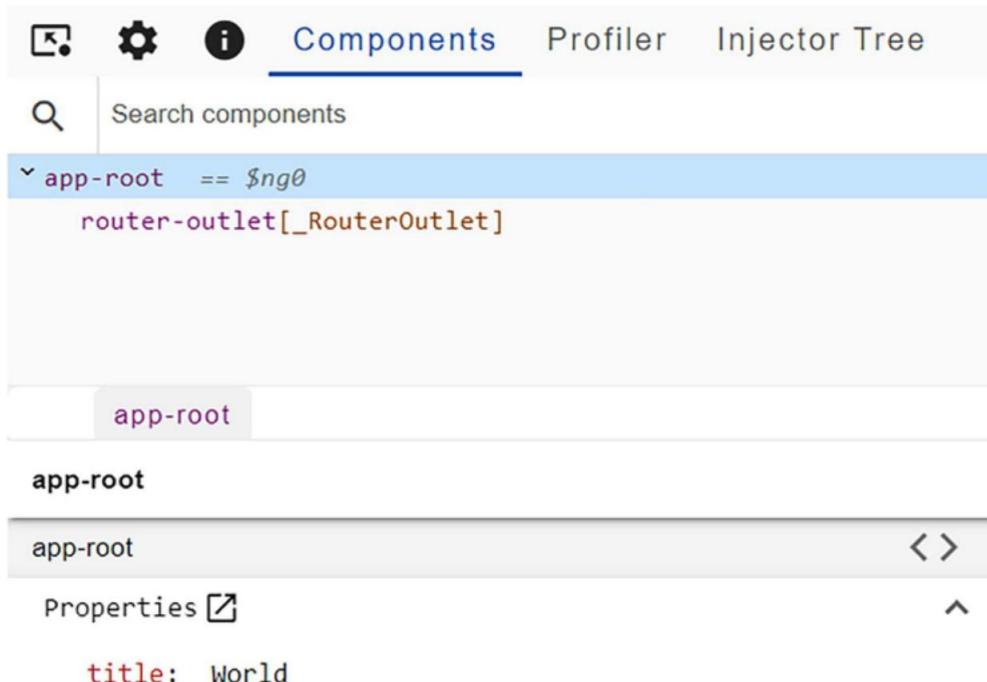
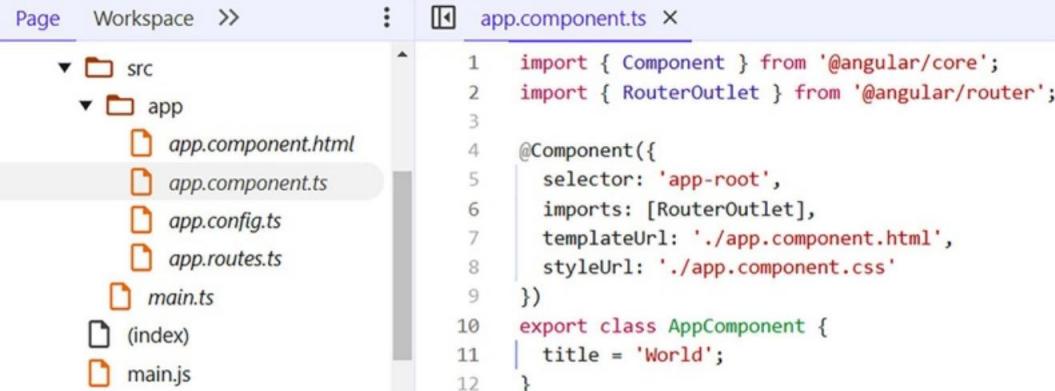


Figura 1.3: Vista previa del componente

Desde la pestaña Componentes , también podemos buscar el elemento HTML correspondiente en el DOM o acceder al código fuente del componente o directiva. Al hacer clic en el botón < > accederemos al archivo TypeScript del componente actual:



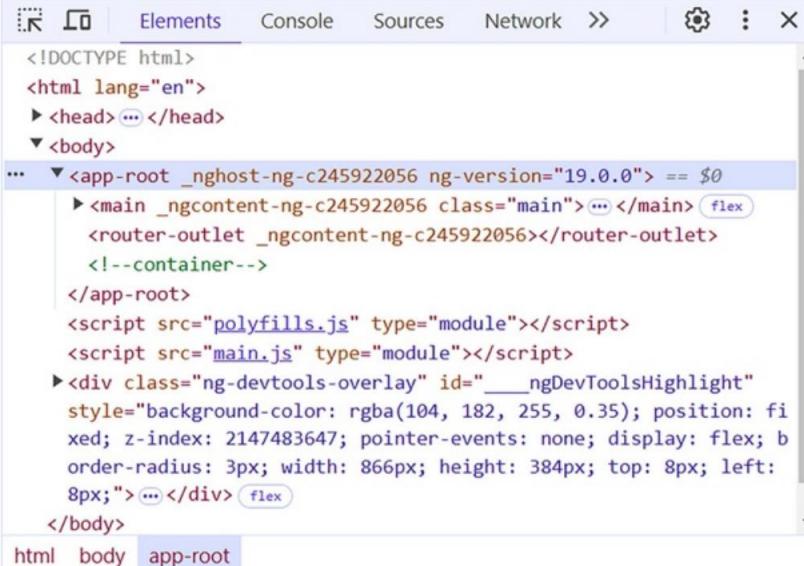
```

Page Workspace >> app.component.ts ×
src
  app
    app.component.html
    app.component.ts
    app.config.ts
    app.routes.ts
    main.ts
  (index)
  main.js
1 import { Component } from '@angular/core';
2 import { RouterOutlet } from '@angular/router';
3
4 @Component({
5   selector: 'app-root',
6   imports: [RouterOutlet],
7   templateUrl: './app.component.html',
8   styleUrls: ['./app.component.css']
9 })
10 export class AppComponent {
11   title = 'World';
12 }

```

Figura 1.4: Archivo fuente de TypeScript

Al hacer doble clic en un selector desde la representación de árbol de la pestaña Componentes , accederemos al DOM de la página principal y resaltaremos el elemento HTML individual:



```

<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    <app-root _ngcontent-ng-c245922056 ng-version="19.0.0"> == $0
      <main _ngcontent-ng-c245922056 class="main">...</main> (flex)
      <router-outlet _ngcontent-ng-c245922056></router-outlet>
      <!--container-->
    </app-root>
    <script src="polyfills.js" type="module"></script>
    <script src="main.js" type="module"></script>
    <div class="ng-devtools-overlay" id="__ngDevToolsHighlight"
        style="background-color: rgba(104, 182, 255, 0.35); position: fixed; z-index: 2147483647; pointer-events: none; display: flex; border-radius: 3px; width: 866px; height: 384px; top: 8px; left: 8px;">...</div> (flex)
  </body>
<html> body app-root

```

Figura 1.5: DOM de la página principal

Finalmente, una de las características más útiles del árbol de componentes es que podemos alterar el valor de una propiedad del componente e inspeccionar cómo se comporta la plantilla del componente:

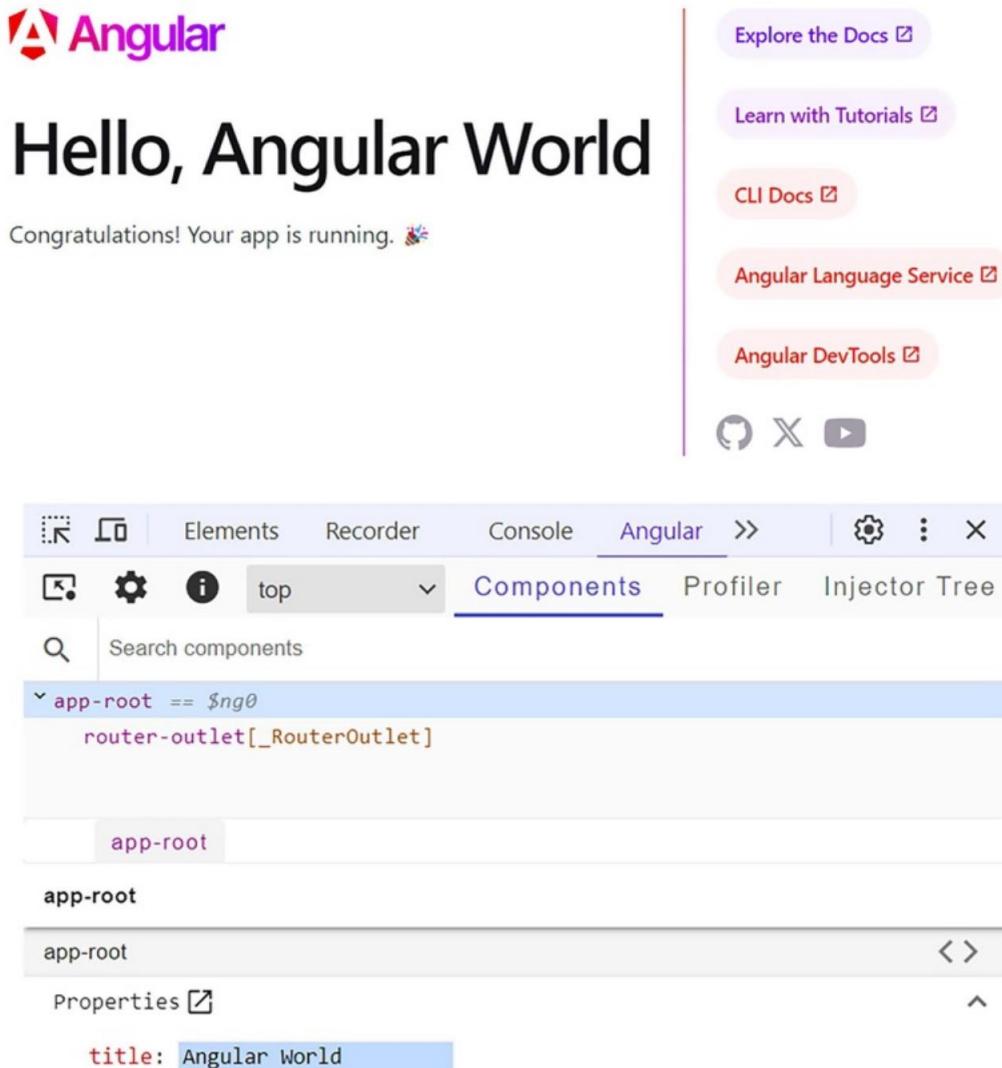


Figura 1.6: Cambiar el estado del componente

En la imagen anterior, puedes ver que cuando cambiamos el valor de la propiedad del título a Angular World, el cambio también se reflejó en la plantilla del componente.

Depurador de VSCode

Podemos depurar una aplicación Angular utilizando técnicas de depuración estándar para aplicaciones web o las herramientas que VSCode proporciona de manera predeterminada.

El objeto de consola es la API web más utilizada para la depuración. Ofrece una forma muy rápida de imprimir datos e inspeccionar valores en la consola del navegador. Para inspeccionar el valor de un objeto en un componente Angular, podemos usar el método `debug` o `log`, pasando el objeto que queremos inspeccionar como parámetro. Sin embargo, se considera un enfoque anticuado, y un código fuente con muchos métodos `console.log` resulta difícil de leer. Una alternativa es usar puntos de interrupción dentro del código fuente mediante el menú de depuración de VSCode.

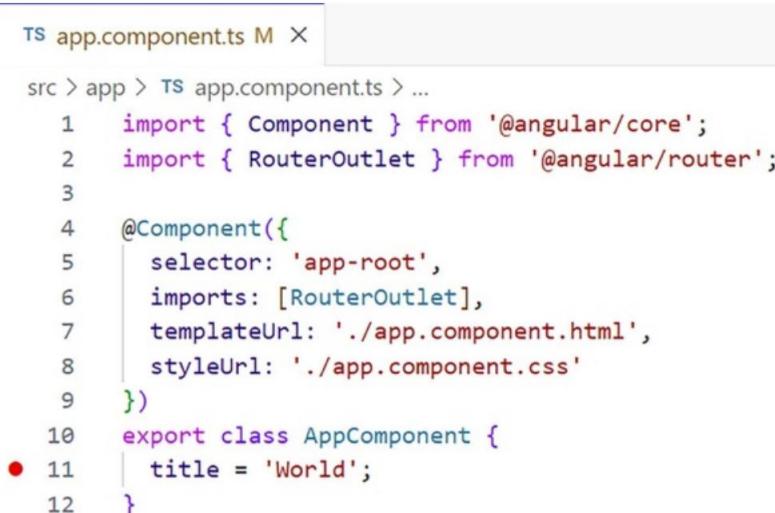


VSCode es un editor de código abierto respaldado por Microsoft. Es muy popular en la comunidad Angular, principalmente por su sólida compatibilidad con TypeScript. TypeScript ha sido, en gran medida, un proyecto impulsado por Microsoft, por lo que es lógico que uno de sus editores más populares se haya concebido con compatibilidad integrada con este lenguaje. Contiene una amplia gama de funciones útiles, como sintaxis, resaltado de errores, compilaciones automáticas y depuración.

VSCode contiene una herramienta de depuración incorporada que utiliza puntos de interrupción para depurar aplicaciones Angular. Podemos agregar puntos de interrupción dentro del código fuente desde VSCode e inspeccionar el estado de una aplicación Angular. Cuando una aplicación Angular se ejecuta y alcanza un punto de interrupción, se pausa y espera. Durante ese tiempo, podemos investigar e inspeccionar varios valores involucrados en el contexto de ejecución actual.

Veamos cómo agregar puntos de interrupción a nuestra aplicación de muestra:

1. Abra el archivo `app.component.ts` y haga clic a la izquierda de la línea 11 para agregar un punto de interrupción. Un punto rojo indica los puntos de interrupción:



```

TS app.component.ts M X
src > app > TS app.component.ts > ...
1 import { Component } from '@angular/core';
2 import { RouterOutlet } from '@angular/router';
3
4 @Component({
5   selector: 'app-root',
6   imports: [RouterOutlet],
7   templateUrl: './app.component.html',
8   styleUrls: ['./app.component.css']
9 })
10 export class AppComponent {
● 11   title = 'World';
12 }

```

Figura 1.7: Agregar un punto de interrupción

2. Haga clic en el botón Ejecutar y depurar en la barra lateral izquierda de VSCode.
3. Haga clic en el botón de reproducción para iniciar la aplicación usando el comando ng serve :

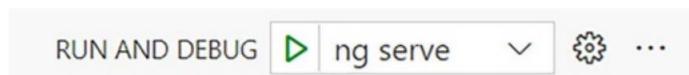
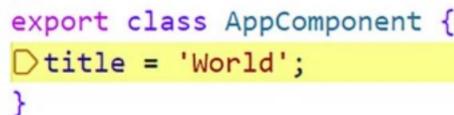


Figura 1.8: Menú Ejecutar y depurar

VSCode creará nuestra aplicación, abrirá el navegador web predeterminado y alcanzará el punto de interrupción.

Dentro del editor:



```

export class AppComponent {
Dtitle = 'World';
}

```

Figura 1.9: Alcanzando un punto de quiebre

Ahora podemos inspeccionar varios aspectos de nuestro componente y utilizar los botones en la barra de herramientas del depurador para controlar la sesión de depuración.

Otra característica poderosa de VSCode son los Perfiles de VSCode, que ayudan a los desarrolladores a personalizar VSCode según sus necesidades de desarrollo.

Perfiles de VSCode

Los perfiles de VSCode nos permiten personalizar los siguientes aspectos del editor de VSCode:

- Configuración: Los ajustes de configuración de VSCode
- Atajos de teclado: Atajos para ejecutar comandos de VSCode con el teclado
- Fragmentos: fragmentos de código de plantilla reutilizables
- Tareas: Tareas que automatizan la ejecución de scripts y herramientas directamente desde VSCode
- Extensiones: Herramientas que nos permiten agregar nuevas capacidades en VSCode, como idiomas, depuradores y linters

Los perfiles también se pueden compartir, lo que nos ayuda a mantener una configuración de desarrollo y un flujo de trabajo consistentes en todo el equipo. VSCode contiene un conjunto de perfiles integrados, incluyendo uno para Angular, que podemos personalizar según nuestras necesidades de desarrollo. Para instalar el perfil de Angular:

1. Haga clic en el botón Administrar representado por el ícono de engranaje en la parte inferior de la barra lateral izquierda en VSCode y seleccione la opción Perfiles .
2. Haga clic en la flecha del botón Nuevo perfil y seleccione Desde plantilla | Angular opción.
3. Haga clic en el botón de engranaje si desea seleccionar un ícono personalizado para su perfil.
4. Haga clic en el botón Crear para crear su perfil.

VSCode aplicará automáticamente el nuevo perfil después de que se haya creado correctamente.

En las siguientes secciones, exploraremos algunas de las extensiones del perfil Angular de VSCode.

Servicio de lenguaje angular

La extensión Angular Language Service , desarrollada y mantenida por el equipo de Angular, proporciona autocompletado de código, navegación y detección de errores dentro de las plantillas de Angular. Enriquece VSCode con las siguientes funciones:

- Completado de código
- Una definición de referencia
- Información rápida
- Mensajes de diagnóstico

Para comprender mejor sus potentes capacidades, veamos la función de completado de código.

Supongamos que queremos mostrar una nueva propiedad llamada descripción en la plantilla del componente principal.

Podemos configurar esto siguiendo los siguientes pasos:

1. Defina la nueva propiedad en el archivo app.component.ts :

```
clase de exportación AppComponent {
  título = 'mi-aplicación';
  descripción = 'Hola mundo';
}
```

2. Abra el archivo app.component.html y agregue el nombre de la propiedad en la plantilla usando la sintaxis de interpolación de Angular. El Servicio de Lenguaje Angular lo encontrará y lo sugerirá automáticamente:



Figura 1.10: Servicio de lenguaje Angular

La propiedad `description` es pública . Podemos omitir la palabra clave "public" al usar propiedades y métodos públicos. El autocompletado de código no funciona con propiedades y métodos privados. Si la propiedad se hubiera declarado privada, Angular Language Service y la plantilla no la habrían reconocido.

Quizás hayas notado que apareció una línea roja inmediatamente debajo del elemento HTML mientras escribías. El Servicio de Lenguaje Angular no reconoció la propiedad hasta que la escribiste correctamente y te indicó correctamente esta falta de reconocimiento. Si pasas el cursor sobre la indicación roja, se muestra un mensaje con información completa sobre el problema:

```
Property 'descr' does not exist on type 'AppComponent'. ngtsc(2339)
app.component.ts(1, 23): Error occurs in the template of component AppComponent.
any
View Problem (Alt+F8) Quick Fix... (Ctrl+.)
```

Figura 1.11: Manejo de errores en la plantilla

El mensaje de información anterior proviene de la función de mensajes de diagnóstico. El Servicio de Lenguaje Angular admite diversos mensajes según el caso de uso. Encontrará más mensajes de este tipo a medida que trabaje con Angular.

Tema de ícono de material

VSCode cuenta con un conjunto integrado de iconos para mostrar diferentes tipos de archivos en un proyecto. La extensión Material Icon Theme proporciona iconos adicionales que cumplen con las directrices de Material Design de Google; un subconjunto de esta colección se centra en los artefactos basados en Angular:

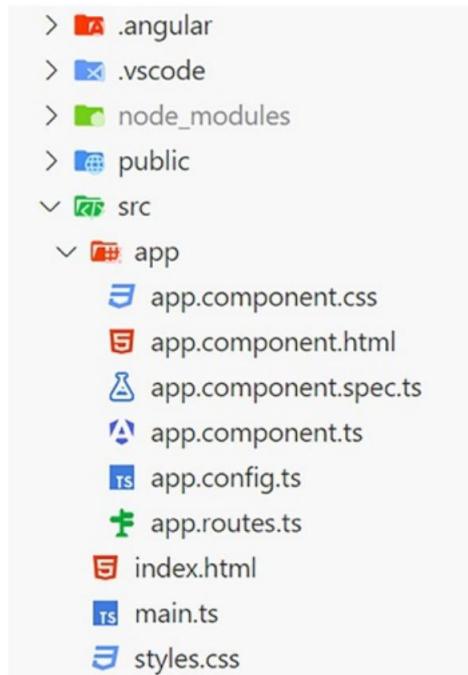


Figura 1.12: Tema del ícono de material

Con esta extensión, puede detectar fácilmente el tipo de archivos Angular en un proyecto, como componentes, y aumentar la productividad del desarrollador, especialmente en proyectos grandes con muchos archivos.

Configuración del editor

La configuración del editor de VSCode, como la sangría o el espaciado, se puede configurar a nivel de usuario o de proyecto. EditorConfig puede anular esta configuración mediante el archivo de configuración `.editorconfig`, que se encuentra en la carpeta raíz de un proyecto de Angular CLI:

```
# Configuración del editor, consulte https://editorconfig.org
raíz = verdadero

[*]
juego de caracteres = utf-8
```

```
estilo_de_sangría = espacio
tamaño de sangría = 2
insertar_nueva_línea_final = verdadero
recortar_espacio_en_blanco_final = verdadero
```

[*.ts]

```
tipo_de_cita = single
ij_typescript_use_double_quotes = falso
```

[*.Maryland]

```
longitud_máxima_de_línea = desactivada
recortar_espacio_en_blanco_final = falso
```

Puede definir configuraciones únicas en este archivo para garantizar la coherencia del estilo de codificación. tu equipo.

Resumen

¡Listo! Tu aventura en el mundo de Angular acaba de comenzar. Repasemos las características que has aprendido hasta ahora. Aprendimos qué es Angular, repasamos su breve historia y examinamos sus beneficios para el desarrollo web.

Vimos cómo configurar nuestro espacio de trabajo de desarrollo y encontrar las herramientas para integrar TypeScript en el juego. Presentamos la herramienta Angular CLI, la herramienta esencial para Angular, que automatiza tareas de desarrollo específicas. Usamos algunos de los comandos más comunes para el andamiaje de nuestra primera aplicación Angular. También examinamos la estructura de nuestra aplicación y aprendimos a... Interactuar con él.

Nuestra primera aplicación nos brindó una comprensión básica de cómo funciona Angular internamente para renderizar nuestra aplicación en una página web. Empezamos nuestro recorrido con el archivo HTML principal de una aplicación Angular. Vimos cómo Angular analiza ese archivo y comienza a buscar en el árbol de componentes para cargar el componente principal. Aprendimos el proceso de arranque de Angular y cómo se utiliza para cargar la configuración de la aplicación.

Finalmente, conocimos algunas de las herramientas Angular más importantes que pueden ayudarte como desarrollador de software. Exploramos cómo usar Angular DevTools para inspeccionar aplicaciones Angular y VSCode Debugger para la depuración. También examinamos VSCode Profiles y cómo puede ayudarnos a mantener un entorno de desarrollo consistente en todo nuestro equipo.

En el siguiente capítulo, aprenderás algunos conceptos básicos del lenguaje TypeScript. Este capítulo abordará los problemas que se pueden resolver introduciendo los tipos y el lenguaje en sí. TypeScript, como superconjunto de JavaScript, contiene muchos conceptos potentes y se integra a la perfección con el framework Angular, como descubrirás a continuación.

Únase a nosotros en Discord

Únase al espacio Discord de nuestra comunidad para discutir con el autor y otros lectores:

<https://packt.link/AprendizajeAngular5e>



2

Introducción a TypeScript

Como aprendimos en el capítulo anterior, cuando creamos nuestra primera aplicación Angular, el código de un proyecto Angular está escrito en TypeScript. Escribir en TypeScript y aprovechar su tipado estático nos brinda una ventaja notable sobre otros lenguajes de scripting. Este capítulo no ofrece una descripción general completa del lenguaje TypeScript. En cambio, nos centraremos en los elementos esenciales que serán útiles para este libro. Como veremos muy pronto, un conocimiento sólido de estos mecanismos es fundamental para comprender cómo funciona la inyección de dependencias en Angular.

En este capítulo cubriremos los siguientes temas principales:

- Fundamentos de JavaScript
- ¿Qué es TypeScript?
- Introducción a TypeScript

Primero, refrescaremos nuestros conocimientos de JavaScript repasando algunas características esenciales de TypeScript, como las funciones y las clases. Después, investigaremos los antecedentes de TypeScript y la lógica detrás de su creación. También aprenderemos a codificar y ejecutar código TypeScript. Haremos hincapié en el sistema de tipificación, que es la principal ventaja de TypeScript, y aprenderemos a utilizarlo para crear tipos e interfaces básicas.

Requisitos técnicos

- GitHub: [https://github.com/PacktPublishing/Learning-Angular-Fifth-Edition/
árbol/principal/ch02](https://github.com/PacktPublishing/Learning-Angular-Fifth-Edition/árbol/principal/ch02)
- Node.js: <https://nodejs.org>
- Git: <https://git-scm.com>
- VSCode: <https://code.visualstudio.com>

Fundamentos de JavaScript

JavaScript es un lenguaje de programación que contiene muchas características para crear aplicaciones web.

En esta sección, repasaremos y refresharemos algunos de los más básicos, ya que están directamente relacionados con el desarrollo en TypeScript y Angular. TypeScript es un superconjunto sintáctico de JavaScript, lo que significa que añade características como tipos, interfaces y genéricos. Analizaremos las siguientes características de JavaScript con más detalle:

- Declaración de variables
- Parámetros de función
- Funciones de flecha
- Encadenamiento opcional
- Coalescencia nula
- Clases
- Módulos



Puede ejecutar todos los ejemplos de código de esta sección de las siguientes maneras:

- Ingrese el código en una ventana de la consola del navegador
- Escriba el código en un archivo JavaScript y use Node.js para ejecutarlo

Si se siente cómodo con estas funciones, puede pasar directamente a la sección [¿Qué es TypeScript?](#)

Declaración de variables

Tradicionalmente, los desarrolladores de JavaScript han usado la palabra clave `var` para declarar objetos, variables y otros artefactos. Esto se debía a que la semántica antigua del lenguaje solo tenía un ámbito de función donde las variables eran únicas dentro de su contexto:

```
función myFunc() {  
    var x = 0;  
}
```

En la función anterior, ninguna otra variable puede declararse como `x` dentro de su cuerpo. Si se declara una, se redefine. Sin embargo, hay casos en los que no se aplica el alcance, como en los bucles:

```
var x = 20;
```

```
para (var x = 0; x < 10; x++) {  
}
```

En el fragmento anterior, la variable x fuera del bucle no afectará a la variable x dentro, ya que tienen un alcance diferente. Para superar esta limitación, JavaScript introdujo la palabra clave let :

```
función myFunc() {  
    sea x = 0;  
    x = 10;  
}
```

La palabra clave let nos permite cambiar la referencia de una variable varias veces en el código.

Otra forma de definir variables en JavaScript es la palabra clave const , que indica que una variable nunca debe cambiar. A medida que crece la base de código, pueden producirse cambios por error, lo cual puede ser costoso. La palabra clave const puede evitar este tipo de errores. Considere el siguiente fragmento de código:

```
precio constante = 100;  
precio = 50;
```

Si intentamos ejecutarlo arrojará el siguiente mensaje de error:

```
TypeError: Asignación a variable constante.
```

El error anterior solo aparecerá en el nivel superior. Debe tenerlo en cuenta si declara objetos como constantes, como en este ejemplo:

```
const producto = { precio: 100 };  
producto.precio = 50;
```

Declarar la variable producto como constante no impide que se edite todo el objeto, sino su referencia. Por lo tanto, el código anterior es válido. Si intentamos cambiar la referencia de la variable, obtendremos el mismo tipo de error que antes:

```
const producto = { precio: 100 };  
producto = { precio: 50 };
```

Es preferible utilizar la palabra clave const cuando estamos seguros de que las propiedades de un objeto no cambiarán durante su vida útil porque evita que el objeto cambie accidentalmente.

Para combinar variables, podemos usar la sintaxis de parámetro de expansión . Un parámetro de expansión usa puntos suspensivos (...) para expandir los valores de una variable:

```
const category = 'Computación';
const categorías = ['Juegos', 'Multimedia'];
const productCategories = [...categorías, categoría];
```

En el fragmento anterior, combinamos la matriz de categorías y el elemento de categoría para crear una nueva matriz. La matriz de categorías aún contiene dos elementos, mientras que la nueva matriz contiene tres. El comportamiento actual se llama inmutabilidad, lo que significa no cambiar una variable sino crear una nueva que provenga de la original.



Un objeto no es inmutable si sus propiedades se pueden cambiar o sus propiedades son un objeto cuyas propiedades se pueden cambiar.

También podemos utilizar un parámetro de propagación en los objetos:

```
producto constante = {
  nombre: 'Teclado',
  precio: 75
};
const nuevoProducto = {
  ...producto,
  precio: 100,
  Categoría: 'Informática'
};
```

En el fragmento anterior, no modificamos el objeto de producto original , sino que creamos una fusión entre ambos. El valor del objeto newProduct será:

```
{
  nombre: 'Teclado',
  precio: 100,
  Categoría: 'Informática'
}
```

El objeto newProduct toma las propiedades del objeto producto , agrega nuevos valores sobre él y reemplaza los existentes.

Parámetros de función

Las funciones en JavaScript son las máquinas de procesamiento que utilizamos para analizar la entrada, procesar la información y aplicar las transformaciones necesarias a los datos. Utilizan parámetros para proporcionar datos que transformen el estado de nuestra aplicación o devuelvan una salida que se utilizará para configurar la lógica de negocio o la interactividad del usuario de nuestra aplicación.

Podemos declarar una función para aceptar parámetros predeterminados de modo que la función asuma un valor predeterminado cuando no se pase explícitamente durante la ejecución:

```
función addtoCart(productId, cantidad = 1) {  
    producto constante = {  
        id: id del producto,  
        cantidad: cantidad  
    };  
}
```

Si no pasamos un valor para el parámetro de cantidad al llamar a la función, obtendremos un objeto de producto con la cantidad establecida en 1.



Los parámetros predeterminados deben definirse después de todos los parámetros obligatorios en la firma de la función.

Una ventaja significativa de la flexibilidad de JavaScript al definir funciones es que acepta una matriz ilimitada y no declarada de parámetros, denominada "parámetros rest". En esencia, podemos definir un parámetro adicional al final de la lista de argumentos, precedido por puntos suspensivos (...):

```
función addProduct(nombre, ...categorías) {  
    producto constante = {  
        nombre,  
        categorías: categorías.join(',')  
    };  
}
```

En la función anterior, usamos el método join para crear una cadena separada por comas a partir del parámetro de categorías . Pasamos cada parámetro por separado al llamar a la función:

```
addProduct('Teclado', 'Computación', 'Periféricos');
```

Los parámetros Rest son útiles cuando no sabemos cuántos argumentos se pasarán como parámetros. La propiedad name también se configura mediante otra característica útil de JavaScript.

En lugar de establecer la propiedad explícitamente en el objeto de producto , usamos directamente su nombre. El siguiente fragmento equivale a la declaración inicial de la función addProduct :

```
función addProduct(nombre, ...categorías) {  
    producto constante = {  
        nombre: nombre,  
        categorías: categorías.join(',')  
    };  
}
```

La sintaxis abreviada para asignar valores de propiedad solo se puede utilizar cuando el nombre del parámetro coincide con el nombre de la propiedad de un objeto.

Funciones de flecha

En JavaScript, podemos crear funciones de una forma alternativa, denominadas funciones de flecha. El propósito de una función de flecha es simplificar la sintaxis general de la función y proporcionar una forma infalible de gestionar el alcance de la función, que tradicionalmente se gestiona mediante el objeto `this` . Considere el siguiente ejemplo, que calcula el descuento de un producto a partir de su precio:

```
descuento constante = (precio) => {  
    retorno (precio / 100) * 10 ;  
};
```

El código anterior no tiene una palabra clave de función , y el cuerpo de la función se define con una flecha ($=>$).

Las funciones de flecha se pueden simplificar aún más siguiendo las siguientes prácticas recomendadas:

- Omite los paréntesis en los parámetros de la función cuando la firma contenga un paréntesis.
Sólo parámetro.
- Omite las llaves en el cuerpo de la función y la palabra clave return si la función tiene
Sólo una afirmación.

La función resultante se verá mucho más simple y fácil de leer:

```
descuento constante = precio => (precio / 100) * 10;
```

Expliquemos ahora cómo se relacionan las funciones de flecha con la gestión del ámbito. El valor de este objeto puede apuntar a un contexto diferente, dependiendo de dónde ejecutemos la función. Al usarlo dentro de una devolución de llamada, perdemos el contexto superior, lo que suele llevarnos a usar convenciones como asignar su valor a una variable externa. Considere la siguiente función, que registra el nombre de un producto mediante la función nativa setTimeout :

```
función crearProducto(nombre) {  
    este.nombre = nombre;  
    esto.getName = función() {  
        setTimeout(función() {  
            console.log(' El nombre del producto es:', this.name);  
        });  
    }  
}
```

Ejecute la función getName utilizando el siguiente fragmento y observe la salida de la consola:

```
const producto = new createProduct('Monitor');  
producto.getName();
```

El fragmento anterior no imprimirá el nombre del producto Monitor como se esperaba porque nuestro código modifica el alcance de este objeto al evaluar la función dentro de la devolución de llamada setTimeout . Para solucionarlo, convierta la función setTimeout para utilizar una función de flecha en su lugar:

```
establecerTiempo de espera() => {  
    console.log(' El nombre del producto es:', this.name);  
});
```

Nuestro código ahora es más simple y podemos usar el alcance de la función de forma segura.

Encadenamiento opcional

El encadenamiento opcional es una función potente que nos ayuda a refactorizar y simplificar nuestro código. En resumen, puede guiar nuestro código para que ignore la ejecución de una sentencia a menos que se haya proporcionado un valor en alguna parte de ella.

Veamos el encadenamiento opcional con un ejemplo:

```
const obtenerOrden = () => {  
    devolver {  
        producto: {  
            nombre: 'Teclado'  
        }  
    }  
}
```

```
    };
}
```

En el fragmento anterior, definimos una función `getOrder` que devuelve el producto de un pedido específico. A continuación, obtenemos el valor de la propiedad `product` , asegurándonos de que exista un pedido antes de leerlo:

```
const orden = obtenerOrden();
si (orden != indefinido) {
    const producto = pedido.producto;
}
```

El fragmento anterior es una medida de precaución en caso de que nuestro objeto haya sido modificado. Si no verificamos el objeto y este queda indefinido, JavaScript generará un error. Sin embargo, podemos usar encadenamiento opcional para mejorar la declaración anterior:

```
const orden = obtenerOrden();
const producto = pedido?.producto;
```

El carácter ? después del objeto de pedido garantiza que solo se acceda a la propiedad del producto si el objeto tiene un valor. El encadenamiento opcional también funciona en casos más complejos, como:

```
const nombre = pedido?.producto?.nombre;
```

En el fragmento anterior, también verificamos si el objeto del producto tiene un valor antes de acceder a su propiedad de nombre .

Coalección nula

La fusión nula se relaciona con proporcionar un valor predeterminado cuando una variable no está definida. Considere el siguiente ejemplo, que asigna un valor a la variable cantidad solo si la variable qty existe:

```
cantidad constante = cantidad ? cantidad : 1;
```

La declaración anterior se llama operador ternario y funciona como una declaración condicional.

Si la variable cantidad no tiene un valor, la variable cantidad se inicializará con el valor predeterminado de 1. Podemos reescribir la expresión anterior usando coalescencia nula como:

```
cantidad constante = cantidad ?? 1;
```

La coalescencia nula nos ayuda a hacer que nuestro código sea legible y más pequeño.

Clases

Las clases de JavaScript nos permiten estructurar el código de nuestra aplicación y crear instancias de cada clase.

Una clase puede tener miembros de propiedad, un constructor, métodos y descriptores de acceso a propiedades. El siguiente fragmento de código ilustra el aspecto de una clase:

```
clase Usuario {  
    nombre = "";  
    apellido = "";  
    #isActive = falso;  
  
    constructor(nombre, apellido, isActive = true) {  
        este.firstName = firstName;  
        este.apellido = apellido;  
        esto.#isActive = isActive;  
    }  
  
    obtenerNombreCompleto() {  
        devuelve `${este.nombre} ${este.apellido}`;  
    }  
  
    ponerse activo() {  
        devuelve esto.#isActive;  
    }  
}
```

La declaración de clase envuelve varios elementos que podemos desglosar:

- Miembro: La clase Usuario contiene los miembros firstName, lastName y #isActive . Los miembros de la clase solo serán accesibles desde la propia clase. Las instancias de la clase Usuario solo tendrán acceso a las propiedades públicas firstName y lastName. #isActive La propiedad no estará disponible porque es privada, como lo indica el carácter # delante del nombre de la propiedad.
- Constructor: El constructor se ejecuta cuando creamos una instancia de la clase.
Se suele usar para inicializar los miembros de la clase con los parámetros proporcionados en la firma. También podemos proporcionar valores predeterminados para parámetros como isActive. parámetro.

- **Método:** Un método representa una función y puede devolver un valor, como `getFullscreen` método, que construye el nombre completo de un usuario. También puede definirse como privado, similar a los miembros de la clase.
- **Accesor de propiedad:** un accesor de propiedad se define anteponiendo a un método el conjunto La palabra clave ``para que sea escribible`` y la palabra clave ``get`` para que sea legible, seguida del nombre de la propiedad que queremos exponer. El método `active` es un descriptor de acceso a propiedades que devuelve el valor del miembro `#isActive` .

Una clase también puede extender miembros y funcionalidades de otras clases. Podemos hacer que una clase herede de otra añadiendo la palabra clave "extends" a la definición de la clase, seguida de la clase. queremos heredar:

```
clase Cliente extiende Usuario {  
    numeroDelImpuesto = "";  
  
    constructor(nombre, apellido) {  
        super(nombre, apellido);  
    }  
}
```

En el fragmento anterior, la clase Cliente extiende la clase Usuario , que expone `firstName` y las propiedades `lastName` . Cualquier instancia de la clase Cliente puede usar esas propiedades de forma predeterminada. También podemos anular métodos de la clase Usuario agregando un método con el mismo nombre. Se requiere que el constructor llame al método `super` , que apunta al constructor de Clase de usuario .

Módulos

A medida que nuestras aplicaciones escalan y crecen, llegará un momento en que necesitaremos organizar mejor nuestro código y hacerlo sostenible y reutilizable. Los módulos son una excelente manera de realizar estas tareas, así que veamos cómo funcionan y cómo podemos implementarlos en nuestra aplicación.

En la sección anterior, aprendimos a trabajar con clases. Tener ambas clases en el mismo archivo no es escalable y su mantenimiento no será fácil. Imagine la cantidad de código que debe procesar para realizar un cambio simple en una de las clases. Los módulos nos permiten separar el código de nuestra aplicación en archivos individuales, aplicando el Patrón de Responsabilidad Única (PRU). Cada archivo es un módulo diferente encargado de una característica o funcionalidad específica.



Una buena indicación para dividir un módulo en varios archivos es cuando este empieza a ocupar dominios diferentes. Por ejemplo, un módulo de productos no puede contener lógica. Para los clientes.

Refactoricemos el código descrito en la sección anterior para que las clases Usuario y Cliente pertenezcan a módulos separados:

1. Abra VSCode y cree un nuevo archivo JavaScript llamado user.js.
2. Ingrese el contenido de la clase Usuario y agregue la palabra clave export en la definición de la clase.

La palabra clave de exportación hace que el módulo esté disponible para otros módulos y forma el público API del módulo.
3. Cree un nuevo archivo JavaScript llamado customer.js y agregue el contenido de Customer.
4. Importe la clase Usuario al archivo customer.js agregando la siguiente declaración en el

parte superior del archivo:

```
importar { Usuario } desde './usuario';
```

Usamos la palabra clave " import " y la ruta relativa del archivo del módulo sin la extensión para importar la clase Usuario . Si un módulo exporta más de un artefacto, los colocamos entre llaves, separados por una coma, como en el ejemplo siguiente:

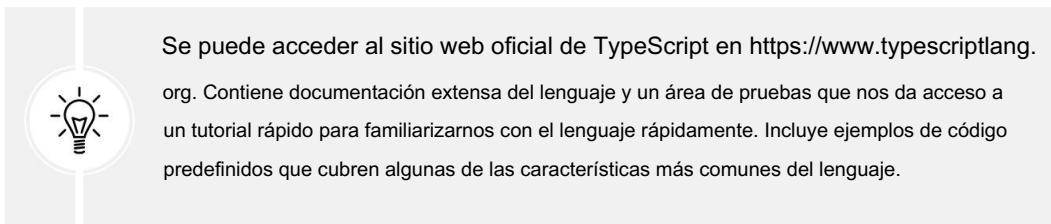
```
importar { Usuario, PreferenciasDeUsuario } desde './usuario';
```

La exploración de los módulos concluye nuestro recorrido por los fundamentos de JavaScript. En la siguiente sección, aprenderemos sobre TypeScript y cómo nos ayuda a crear aplicaciones web.

¿Qué es TypeScript?

Transformar pequeñas aplicaciones web en clientes monolíticos y densos era imposible debido a las limitaciones de las versiones anteriores de JavaScript. En resumen, las aplicaciones JavaScript a gran escala sufrían graves problemas de mantenimiento y escalabilidad a medida que aumentaban de tamaño y complejidad. Este problema se agravó a medida que nuevas bibliotecas y módulos requerían una integración fluida en nuestras aplicaciones. La falta de mecanismos adecuados de interoperabilidad dio lugar a soluciones engorrosas.

Para superar estas dificultades, Microsoft creó un superconjunto del lenguaje JavaScript que permitía crear aplicaciones empresariales con menor huella de errores mediante la comprobación de tipos estáticos, mejores herramientas y análisis de código. TypeScript 1.0 se presentó en 2014. Se adelantaba a JavaScript, implementaba las mismas características y proporcionaba un entorno estable para la creación de aplicaciones a gran escala. Introdujo la tipificación estática opcional mediante anotaciones de tipos, lo que garantizaba la comprobación de tipos en tiempo de compilación y la detección de errores en las primeras etapas del proceso de desarrollo. Su compatibilidad con archivos de declaración también permitía a los desarrolladores describir la interfaz de sus módulos para que otros desarrolladores pudieran integrarlos mejor en su flujo de trabajo y herramientas de código.



Se puede acceder al sitio web oficial de TypeScript en <https://www.typescriptlang.org>. Contiene documentación extensa del lenguaje y un área de pruebas que nos da acceso a un tutorial rápido para familiarizarnos con el lenguaje rápidamente. Incluye ejemplos de código predefinidos que cubren algunas de las características más comunes del lenguaje.

Como superconjunto de JavaScript, una de las principales ventajas de adoptar TypeScript en tu próximo proyecto es la facilidad de acceso. Si conoces JavaScript, prácticamente tienes todo listo, ya que todas las funciones adicionales de TypeScript son opcionales. Puedes elegir e incorporar cualquiera de ellas para lograr tu objetivo. En resumen, hay una larga lista de argumentos sólidos para usar TypeScript en tu próximo proyecto, y todos son aplicables a Angular.

A continuación se presenta un breve resumen de algunas de las ventajas:

- Anotar su código con tipos garantiza la integración consistente de sus diferentes códigos. unidades y mejora la legibilidad y comprensión del código.
- El verificador de tipos integrado analiza su código en tiempo de compilación y le ayuda a evitar errores. antes de ejecutar su código.

El uso de tipos garantiza la coherencia en toda la aplicación. En combinación con los dos métodos anteriores, se minimiza la huella de errores de código a largo plazo.

- Las interfaces garantizan la integración fluida y perfecta de sus bibliotecas en otros sistemas y bases de código.
- La compatibilidad de idiomas entre diferentes IDE es sorprendente y usted puede beneficiarse de características como resaltado de código, verificación de tipos en tiempo real y compilación automática sin costo.
- La sintaxis es familiar para los desarrolladores de otros entornos basados en OOP, como Java, C# y C++.

En la siguiente sección, aprenderemos a desarrollar y ejecutar una aplicación TypeScript. En las aplicaciones Angular, no es necesario ejecutar código TypeScript manualmente, ya que la CLI de Angular lo gestiona automáticamente; sin embargo, conviene saber cómo funciona internamente.

Introducción a TypeScript

El lenguaje TypeScript es un paquete npm que se puede instalar desde el registro npm usando el siguiente comando:

```
npm install -g typescript
```

En el comando anterior, decidimos instalar TypeScript globalmente en nuestro sistema para poder usarlo desde cualquier ruta en nuestro entorno de desarrollo. Veamos cómo podemos usar TypeScript con un ejemplo sencillo:

1. Abra VSCode y seleccione Archivo | Nuevo archivo... en las opciones del menú principal.
2. Ingrese app.ts en el cuadro de diálogo Nuevo archivo... y presione Entrar.

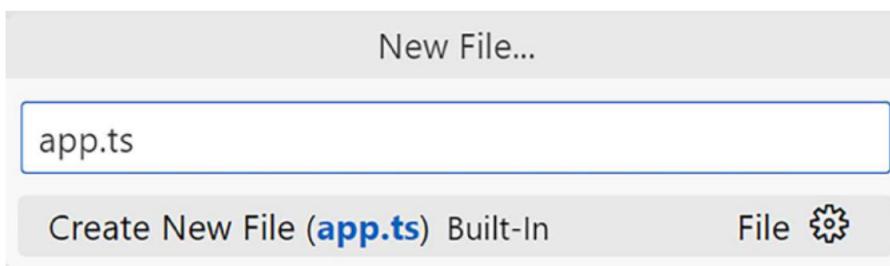


Figura 2.1: Cuadro de diálogo Nuevo archivo...

Como ya hemos aprendido, los archivos TypeScript tienen una extensión .ts .

3. Seleccione la ruta donde desea crear el nuevo archivo. VSCode lo abrirá dentro del editor.
4. Escriba el siguiente fragmento en el archivo app.ts :

```
const title = '¡Hola TypeScript!';
```

Aunque hemos creado un archivo TypeScript, el fragmento anterior es un código JavaScript válido.

Recordemos que TypeScript es un superconjunto de JavaScript que proporciona una sintaxis más compleja mediante su sistema de tipado. Sin embargo, escribir código JavaScript simple con TypeScript no nos ofrece ninguna ventaja clara.

5. Abra una ventana de terminal y ejecute el siguiente comando para compilar el archivo TypeScript en JavaScript:

```
 aplicación tsc.ts
```

El comando anterior inicia un proceso llamado transpilación realizado por el tsc Ejecutable, un compilador fundamental para el lenguaje TypeScript. Necesitamos compilar código TypeScript en JavaScript porque los navegadores actualmente no lo admiten. Fuera de la caja.



Angular utiliza un compilador que utiliza el compilador TypeScript para crear aplicaciones Angular.

El compilador de TypeScript admite opciones de configuración adicionales que podemos pasar al ejecutable tsc mediante la ventana de terminal o un archivo de configuración. La lista completa de opciones de compilación disponibles se encuentra en <https://www.typescriptlang.org/docs/manual/options-del-compilador.html>.

6. El proceso de transpilación creará un archivo app.js en la misma carpeta que el TypeScript archivo. El nuevo archivo contendrá el siguiente código:

```
 var title = '¡Hola TypeScript!';
```

Como todavía no hemos utilizado ninguna característica específica de TypeScript, el fragmento anterior parece casi idéntico al original, excepto por la declaración de la variable.

7. El proceso de transpilación reemplazó la palabra clave const por la palabra clave var porque el compilador de TypeScript usa una versión antigua de JavaScript por defecto. Podemos cambiar esto especificando un destino en el comando tsc :

```
tsc app.ts --target es2022
```

En el comando anterior, especificamos es2022, que representa la versión más reciente de JavaScript al momento de escribir este libro. Las aplicaciones Angular que crearemos a lo largo de este libro también usan la misma versión de JavaScript por defecto.

8. Dado que usaremos la última versión de JavaScript en el resto de este capítulo, definamos la opción de destino mediante un archivo de configuración de TypeScript. Cree un archivo llamado tsconfig.json. en la carpeta actual y agregue el siguiente contenido:

```
{  
  "opcionesdelcompilador": {  
    "objetivo": "ES2022"  
  }  
}
```

Puede encontrar más opciones para el archivo de configuración de TypeScript en <https://www.typescriptlang.org/tsconfig>.

Ejecute el comando tsc en una ventana de terminal para verificar que el archivo JavaScript de salida permanezca sin cambios.



Cuando ejecutamos el comando tsc sin opciones, compilará todos los archivos TypeScript en la carpeta actual utilizando las opciones del archivo de configuración.

El código TypeScript que hemos escrito hasta ahora no utiliza características específicas de TypeScript. En la siguiente sección, aprenderemos a usar el sistema de tipado, la característica más potente y esencial del lenguaje TypeScript.

Tipos

Trabajar con TypeScript o cualquier otro lenguaje de programación implica trabajar con datos, y estos datos pueden representar diferentes tipos de contenido . Los tipos se utilizan para representar que los datos pueden ser texto, un valor entero o una matriz de estos tipos de valores, entre otros.



Los tipos desaparecen durante la transpilación y no se incluyen en el código JavaScript final.

Es posible que ya hayas encontrado tipos en JavaScript, ya que siempre hemos trabajado implícitamente con ellos. En JavaScript, cualquier variable dada puede asumir (o devolver, en el caso de las funciones) cualquier valor. A veces, esto genera errores y excepciones en nuestro código debido a colisiones de tipos entre lo que nuestro código devolvía y lo que esperábamos devolver. Sin embargo, tipificar estáticamente nuestras variables nos proporciona a nuestro IDE y a nosotros mismos una buena visión del tipo de datos que deberíamos encontrar en cada instancia de código. Se convierte en una herramienta invaluable para ayudar a depurar nuestras aplicaciones en tiempo de compilación. antes de ejecutar el código.

Cadena

Uno de los tipos primitivos más utilizados es la cadena, que rellena una variable con texto:

```
const producto: cadena = 'Teclado';
```

El tipo se define agregando dos puntos y el nombre del tipo junto a la variable.

Booleano

El tipo booleano define una variable que puede tener un valor verdadero o falso:

```
const isActive: booleano = verdadero;
```

El resultado de una variable booleana representa el cumplimiento de una declaración condicional.

Número

El tipo de número es probablemente el otro tipo de datos primitivo más utilizado, junto con la cadena y booleano:

```
precio constante : numero = 100;
```

Podemos utilizar el tipo de número para definir un número de punto flotante y hexadecimal, decimal, binario, y literales octales.

Formación

El tipo array define una lista de elementos que contienen únicamente un tipo determinado. Con este tipo, ahora es fácil evitar la gestión de excepciones derivadas de errores, como la asignación de tipos de miembro incorrectos en una lista. Podemos definir arrays utilizando la sintaxis de corchetes o la palabra clave Array :

```
const categorías: string[] = ['Computación', 'Multimedia'];
const categorías: Array<string> = ['Computación', 'Multimedia'];
```



Es aconsejable llegar a un acuerdo con su equipo sobre la sintaxis y mantenerla durante el desarrollo de la aplicación.

Si intentamos agregar un nuevo elemento a la matriz de categorías con un tipo distinto de cadena, TypeScript arrojará un error, lo que garantiza que nuestros miembros tipificados permanezcan consistentes y que nuestro código esté libre de errores.

cualquier

En todos los casos anteriores, la tipificación es opcional porque TypeScript es lo suficientemente inteligente como para inferir los tipos de datos de las variables a partir de sus valores con un cierto nivel de precisión.



Es muy importante dejar que el sistema de tipificación infiera los tipos, en lugar de escribirlos manualmente. El sistema de tipificación nunca se equivoca, pero el desarrollador sí.

Sin embargo, si no es posible, el sistema de tipado asignará automáticamente el tipo dinámico `any` a los datos de tipado flexible, reduciendo al mínimo la comprobación de tipos. Además, podemos añadir el tipo `any` a nuestro código manualmente cuando sea difícil inferir el tipo de dato a partir de la información disponible en un momento dado. El tipo `any` incluye todos los demás tipos existentes, por lo que podemos escribir cualquier valor de dato con él y asignarle cualquier valor posteriormente:

```
dejar orden: cualquiera;  
función setOrderNo() {  
    orden = '0001';  
}
```



TypeScript contiene otro tipo, similar al tipo "any", llamado "unknown". Una variable de tipo "unknown" puede tener un valor de cualquier tipo. La principal diferencia radica en que TypeScript no permite aplicar operaciones arbitrarias a valores desconocidos, como llamar a un método, a menos que se realice previamente una comprobación de tipos.

Sin embargo, un gran poder conlleva una gran responsabilidad. Si ignoramos la comodidad de la comprobación de tipos estática, exponemos a errores de tipo al canalizar datos a través de nuestra aplicación. Es nuestra responsabilidad garantizar la seguridad de tipos en toda nuestra aplicación.

Tipos personalizados

En TypeScript, puedes crear tu propio tipo si lo necesitas utilizando la palabra clave type de la siguiente manera:

```
tipo Categorías = 'informática' | 'multimedia';
```

Luego podemos crear una variable de un tipo específico de la siguiente manera:

```
const category: Categorías = 'informática';
```

El código anterior es perfectamente válido, ya que la computación es uno de los valores permitidos y funciona según lo previsto. Los tipos personalizados son una excelente manera de agregar tipos con un número finito de valores permitidos.

Cuando queremos crear un tipo personalizado a partir de un objeto, podemos usar el operador `keyof`. El operador `keyof` nos permite iterar sobre las propiedades de un objeto y extraerlas en un nuevo tipo:

```
tipo Categoría = {  
    computación: cadena;  
    multimedia: cadena;  
};  
tipo CategoryType = clave de Categoría;
```

En el fragmento anterior, `CategoryType` produjo el mismo resultado que el tipo `Categories`.

Aprenderemos cómo podemos usar el operador `keyof` para iterar sobre las propiedades de los objetos de forma dinámica en el Capítulo 4, Enriquecimiento de aplicaciones mediante tuberías y directivas.

El sistema de tipado de TypeScript se utiliza principalmente para anotar código JavaScript con tipos. Mejora la experiencia del desarrollador al proporcionar IntelliSense y prevenir errores en las primeras etapas del desarrollo. En la siguiente sección, aprenderemos más sobre cómo añadir anotaciones de tipo en funciones.

Funciones

Las funciones en TypeScript no difieren mucho del JavaScript normal, salvo que, como todo en TypeScript, se pueden anotar con tipos estáticos. Por lo tanto, mejoran el compilador al proporcionar la información que espera en su firma y el tipo de dato que pretende devolver, si lo hay.

El siguiente ejemplo muestra cómo se anota una función regular en TypeScript:

```
función getProduct(): cadena {  
    devuelve 'Teclado';  
}
```

En el fragmento anterior, anotamos el valor devuelto de la función agregando la cadena

Tipo a la declaración de la función. También podemos añadir tipos a los parámetros de la función, como:

```
función getfullname(nombre: cadena, apellido: cadena): cadena {  
    devuelve `${este.nombre} ${este.apellido}`;  
}
```

En el fragmento anterior, anotamos los parámetros declarados en la firma de la función, lo que tiene sentido ya que el compilador querrá verificar si los datos proporcionados tienen el tipo correcto.



Como se mencionó en la sección anterior, el compilador de TypeScript es lo suficientemente inteligente como para inferir tipos cuando no se proporciona ninguna anotación. En ambas funciones anteriores, podríamos omitir el tipo porque el compilador podría inferirlo a partir de los argumentos proporcionados y las sentencias devueltas.

Cuando una función no devuelve un tipo, podemos anotarla usando el tipo void :

```
función printFullscreen(nombre: cadena, apellido: cadena): void {  
    console.log(`{este.nombre} ${este.apellido}`);  
}
```

Ya hemos aprendido a usar los parámetros predeterminados y restantes en funciones de JavaScript. TypeScript amplía las funciones introduciendo parámetros opcionales. Los parámetros se definen como opcionales añadiendo el carácter "?" después del nombre del parámetro:

```
función addtoCart(productId: número, cantidad?: número) {  
    producto constante = {  
        id: id del producto,  
        cantidad: cantidad ?? 1  
    };  
}
```

En la función anterior, definimos la cantidad como parámetro opcional. También usamos la sintaxis de fusión nula para establecer la propiedad qty del objeto producto si no se pasa la cantidad .

Podemos invocar la función addToCart pasando solo el parámetro productId o ambos.



Los parámetros opcionales deben colocarse al final de la firma de una función.

Ya hemos aprendido cómo las clases de JavaScript pueden ayudarnos a estructurar el código de nuestra aplicación. En la siguiente sección, veremos cómo usarlas en TypeScript para mejorar aún más nuestra aplicación.

Clases

Considere la clase Usuario que definimos en el archivo user.js :

```
clase de exportación Usuario {  
    nombre = ";
```

```
apellido = "";
#isActive = false;

constructor(nombre, apellido, isActive = true) {
    este.nombre = nombre; este.apellido
    = apellido;
    esto.#isActive = isActive;
}

obtenerNombreCompleto() {
    devuelve `${este.nombre} ${este.apellido}`;
}

ponerse activo() {
    devuelve esto.#isActive;
}
}
```

Daremos pasos simples y pequeños para agregar tipos a lo largo de la clase:

1. Convierta el archivo a TypeScript cambiándole el nombre a user.ts.
2. Agregue los siguientes tipos a todas las propiedades de clase:

```
nombre: cadena = "";
apellido: cadena = ""; privado
isActive: booleano = falso;
```

En el fragmento anterior, también usamos el modificador privado para definir la propiedad isActive como privada.

3. Modifique el constructor agregando tipos a los parámetros:

```
constructor(nombre: cadena, apellido: cadena, isActive: booleano = true) { this.firstName =
nombre;

este.apellido = apellido;
esto.isActive = isActive;
}
```



Alternativamente, podríamos omitir las propiedades de clase y tener el constructor. créelos automáticamente declarando parámetros como privados:

```
constructor(private firstName: cadena, private lastName: cadena, private  
isActive: boolean = true) {}
```

4. Finalmente, agregue tipos en el descriptor de acceso de propiedad activo y en el método getFullname :

```
obtenerNombreCompleto(): cadena {  
    devuelve `${este.nombre} ${este.apellido}`;  
}  
  
activarse (): booleano {  
    devuelve esto.isActive;  
}
```

Convertir una clase de JavaScript en TypeScript y agregar tipos es un paso importante para aprovechar la función de escritura en TypeScript.

Otra gran característica de TypeScript relacionada con las clases es la palabra clave `instanceOf`. Esta permite comprobar el tipo de instancia de la clase y proporciona las propiedades correctas según la clase relacionada.

Explorémoslo con la clase Cliente definida en el archivo customer.js :

1. Convierta el archivo a TypeScript cambiándole el nombre a customer.ts.
2. Reescriba la clase Cliente de la siguiente manera para agregar tipos:

```
clase Cliente extiende Usuario {  
    taxNumber: número;  
  
    constructor(nombre: cadena, apellido: cadena) {  
        super(nombre, apellido);  
    }  
}
```

3. Cree un objeto fuera de la clase que pueda ser tanto del tipo Usuario como del tipo Cliente :

```
const cuenta: Usuario | Cliente = indefinido;
```

4. Ahora podemos usar la palabra clave `instanceOf` para acceder a diferentes propiedades de la cuenta. objeto según la clase subyacente:

```
si ( instancia de cuenta de Cliente) {  
    const taxNo = cuenta.taxNumber;  
} demás {  
    const nombre = cuenta.getFullscreen();  
}
```

TypeScript es lo suficientemente inteligente como para comprender que el objeto de cuenta en la instrucción `else` no tiene una propiedad `taxNumber` porque es del tipo `Usuario`. Incluso si intentamos acceder a él por error, VSCode generará un error:

```
Property 'taxNumber' does not exist on type 'User'. ts(2339)  
any  
View Problem (Alt+F8) Quick Fix... (Ctrl+.)
```

Figura 2.2: Error de acceso a la propiedad

Las clases de TypeScript nos ayudan a escribir código bien estructurado, se pueden instanciar, contienen lógica de negocio y proporcionan tipado estático en nuestra aplicación. A medida que las aplicaciones escalan y se crean más clases, necesitamos encontrar maneras de garantizar la consistencia y el cumplimiento de las reglas en nuestro código. Como aprenderemos en la siguiente sección, una de las mejores maneras de abordar la consistencia y la validación de tipos es... para crear interfaces.

Interfaces

Una interfaz es un contrato de código que define un esquema específico. Todos los artefactos, como clases y funciones, que implementan una interfaz deben cumplir con este esquema. Las interfaces son útiles cuando queremos aplicar un tipado estricto a las clases generadas por las fábricas o cuando definimos firmas de funciones para garantizar que una propiedad tipada específica se encuentre en la carga útil.



Las interfaces desaparecen durante la transpilación y no se incluyen en el JavaScript final. código.

En el siguiente fragmento, definimos una interfaz para administrar productos:

```
interfaz Producto {  
    nombre: cadena;  
    precio: numero;  
    obtenerCategorías: () => cadena[];  
}
```



Las interfaces son el enfoque recomendado cuando se trabaja con datos de una API de backend u otra fuente.

Una interfaz puede contener propiedades y métodos. En el fragmento anterior, la interfaz `Producto` contenía las propiedades `nombre` y `precio`. También definía el método `getCategories`. Una clase puede usar una interfaz añadiendo la palabra clave `implements` y el nombre de la interfaz en la declaración de clase:

```
La clase Teclado implementa Producto {  
    nombre: cadena = 'Teclado';  
    precio: numero = 20;  
  
    obtenerCategorías(): cadena[] {  
        devolver ['Computación', 'Periféricos'];  
    }  
}
```

En el fragmento anterior, la clase `Keyboard` debe implementar todos los miembros de la interfaz `Product`; de lo contrario, TypeScript generará un error. Si no queremos implementar un miembro de la interfaz, podemos definirlo como opcional mediante el carácter `?`.

```
interfaz Producto {  
    nombre: cadena;  
    precio: numero;  
    obtenerCategorías: () => cadena[];  
    descripción?: cadena;  
}
```

También podemos usar interfaces para cambiar el tipo de una variable, lo que se denomina conversión de tipos . Esta conversión es útil al trabajar con datos dinámicos o cuando TypeScript no puede inferir el tipo de una variable automáticamente. En el siguiente código, le indicamos a TypeScript que trate el objeto de producto como un tipo de producto :

```
producto constante = {  
    nombre: 'Teclado',  
    precio: 20  
} como Producto;
```

Sin embargo, la conversión de tipos debe usarse con precaución. En el fragmento anterior, omitimos intencionalmente el método `getCategories` , pero TypeScript no generó ningún error. Al usar la conversión de tipos, le indicamos a TypeScript que una variable simula ser de un tipo específico.



Se recomienda evitar la conversión de tipos siempre que sea posible y definir los tipos explícitamente.

Las interfaces se pueden combinar con genéricos para proporcionar un comportamiento de código general independientemente del tipo de datos, como aprenderemos en la siguiente sección.

Genéricos

Los genéricos se utilizan cuando queremos utilizar tipos dinámicos en otros artefactos de TypeScript, como métodos.

Supongamos que queremos crear una función para guardar un objeto `Producto` en el almacenamiento local del navegador:

```
función guardar(datos: Producto) {  
    localStorage.setItem('Producto', JSON.stringify(datos));  
}
```

En el código anterior, definimos explícitamente el parámetro de datos como `Producto`. Si también queremos guardar objetos de teclado , debemos modificar el método de guardado de la siguiente manera:

```
función guardar(datos: Producto | Teclado) {  
    localStorage.setItem('Producto', JSON.stringify(datos));  
}
```

Sin embargo, el enfoque anterior no es escalable si deseamos añadir otros tipos en el futuro. En su lugar, podemos usar genéricos para que el consumidor del método de guardado decida el tipo de dato que se pasa:

```
función guardar<T>(datos: T) {  
    localStorage.setItem('Producto', JSON.stringify(datos));  
}
```

En el ejemplo anterior, el tipo de T no se evalúa hasta que se utiliza el método. Usamos T como convención para definir genéricos, pero también se pueden usar otras letras. Podemos ejecutar el método de guardado para un objeto Producto de la siguiente manera:

```
guardar<Producto>({  
    nombre: 'Micrófono',  
    precio: 45,  
    getCategories: () => ['Periféricos', 'Multimedia']  
});
```

Como puede ver, su tipo varía según cómo lo llame. Esto también garantiza que esté pasando el tipo de datos correcto.

Supongamos que el método anterior se llama así:

```
guardar<Producto>('Micrófono');
```

Especificamos que T debe ser un Producto, pero insistimos en pasar su valor como una cadena. El compilador indica claramente que esto no es correcto. Si quisieramos usar más genéricos en nuestro método de guardado , podríamos usar letras diferentes, como:

```
función guardar<T, P>(datos: T, obj: P) {  
    localStorage.setItem('Producto', JSON.stringify(datos));  
}
```

Los genéricos se utilizan a menudo en colecciones porque tienen un comportamiento similar, independientemente del tipo. Sin embargo, pueden usarse en otras construcciones, como métodos. La idea es que los genéricos indiquen si se van a mezclar tipos de una forma no permitida.

Los genéricos son muy útiles si se tiene un comportamiento típico con muchos tipos de datos diferentes. Probablemente no se escriban genéricos personalizados, al menos al principio, pero conviene saber qué sucede.

En la siguiente sección, veremos algunos tipos de utilidades relacionadas con las interfaces que nos ayudarán durante el desarrollo de Angular.

Tipos de utilidad

Los tipos de utilidad son tipos que nos ayudan a derivar nuevos tipos a partir de los existentes.

El tipo `Partial` se utiliza cuando queremos crear un objeto desde una interfaz donde todas sus propiedades son opcionales.

En el siguiente fragmento, usamos la interfaz `Product` para declarar una versión reducida de un producto:

```
const mic: Parcial<Producto> = {  
    nombre: 'Micrófono',  
    precio: 67  
};
```

En el fragmento anterior, podemos ver que el objeto `mic` no contiene `getCategories`

Método. Como alternativa, podríamos usar el tipo `Pick`, que nos permite crear un objeto a partir de un subconjunto de propiedades de interfaz:

```
tipo Micrófono = Pick<Producto, 'nombre' | 'precio'>;  
const micrófono: Micrófono = {  
    nombre: 'Micrófono',  
    precio: 67  
};
```

Algunos lenguajes, como C#, tienen un tipo reservado al definir un objeto o diccionario de pares clave-valor. En TypeScript, si queremos definir dicho tipo, podemos usar un tipo de registro :

```
Interfaz Orden {  
    productos: Registro<cadena, número>;  
}
```

El fragmento anterior define el nombre del producto como una cadena y la cantidad como un número.

Puede encontrar más tipos de utilidades en <https://www.typescriptlang.org/docs/handbook/utility-types.html> .

Resumen

Fue una lectura larga, pero esta introducción a TypeScript fue necesaria para comprender la lógica detrás de muchas de las partes más brillantes de Angular. Nos permitió presentar la sintaxis del lenguaje y explicar las razones de su éxito como la sintaxis preferida para construir Angular. estructura.

Revisamos la arquitectura de tipos y cómo podemos crear lógica de negocio avanzada al diseñar funciones con diversas alternativas para firmas parametrizadas. Incluso descubrimos cómo evitar problemas relacionados con el alcance mediante las potentes funciones de flecha. Ampliamos nuestros conocimientos de TypeScript explorando algunas de las características más comunes de las aplicaciones Angular.

Probablemente la parte más relevante de este capítulo abarcó nuestra descripción general de clases, métodos, propiedades y accesores y cómo podemos manejar la herencia y un mejor diseño de aplicaciones a través de interfaces.

Con todo este conocimiento podemos empezar a aprender cómo aplicarlo construyendo aplicaciones Angular. En el próximo capítulo, aprenderemos cómo usar componentes Angular para crear interfaces de usuario componibles para mantener el código de nuestra aplicación y hacerlo más escalable.

Machine Translated by Google

3

Estructuración de interfaces de usuario con Componentes

Hasta ahora, hemos tenido la oportunidad de analizar el framework Angular a detalle. Aprendimos a crear una nueva aplicación Angular usando la CLI de Angular y a interactuar con un componente Angular mediante la sintaxis de plantilla. También exploramos TypeScript, lo que nos ayudará a comprender cómo escribir código Angular. Tenemos todo lo necesario para explorar las nuevas posibilidades que Angular ofrece al juego en cuanto a la creación de componentes interactivos y su comunicación entre ellos.

En este capítulo aprenderemos los siguientes conceptos:

- Creando nuestro primer componente
- Interactuar con la plantilla
- Intercomunicación de componentes
- Encapsular estilos CSS
- Decidir una estrategia de detección de cambios
- Introducción al ciclo de vida de los componentes

Requisitos técnicos

Este capítulo contiene varios ejemplos de código que te guiarán a través de los componentes de Angular. Puedes encontrar el código fuente relacionado en la carpeta ch03 del siguiente repositorio de GitHub:

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

Creando nuestro primer componente

Los componentes son los componentes básicos de una aplicación Angular. Controlan diferentes partes de una página web llamadas vistas, como una lista de productos o un formulario de pago. Son responsables de la lógica de presentación de una aplicación Angular y están organizados en un árbol jerárquico de componentes que pueden interactuar entre sí:

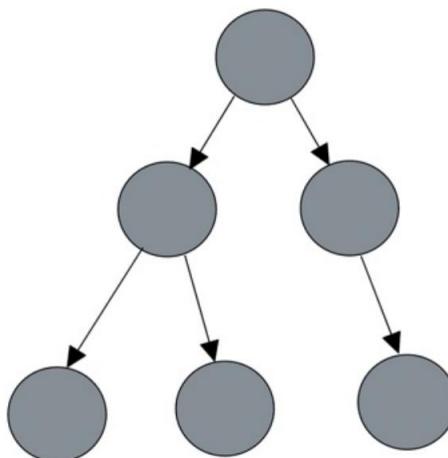


Figura 3.1: Arquitectura de componentes

La arquitectura de una aplicación Angular se basa en componentes Angular. Cada componente Angular puede comunicarse e interactuar con uno o más componentes en el árbol de componentes. Como se puede ver en la Figura 3.1, un componente puede ser simultáneamente padre de varios componentes secundarios y secundario de otro componente principal.

En esta sección, exploraremos los siguientes temas sobre los componentes angulares:

- La estructura de un componente Angular • Creación de componentes con la CLI de Angular

Comenzaremos nuestro viaje investigando el funcionamiento interno de los componentes Angular.

La estructura de un componente Angular

Como aprendimos en el Capítulo 1, "Creando tu primera aplicación Angular", una aplicación Angular típica contiene al menos un componente principal que consta de varios archivos. La clase TypeScript del componente se define en el archivo app.component.ts :

```
importar { Componente } desde '@angular/core';
importar { RouterOutlet } desde '@angular/router';
```

```
@Componente({  
  selector: 'app-root',  
  importaciones: [RouterOutlet],  
  URL de plantilla: './app.component.html',  
  URL de estilo: './app.component.css'  
})  
clase de exportación AppComponent {  
  título = 'Mundo';  
}
```

@Component es un decorador de Angular que define las propiedades del componente de Angular. Un decorador de Angular es un método que acepta un objeto con metadatos como parámetro. Los metadatos se utilizan para configurar una clase de TypeScript como un componente de Angular mediante las siguientes propiedades:

Selector : Un selector CSS que indica a Angular que cargue el componente en la ubicación que encuentra la etiqueta correspondiente en una plantilla HTML. La CLI de Angular añade la aplicación. prefijo por defecto, pero puedes personalizarlo usando la opción --prefix al crear el proyecto Angular.

- Importaciones: Define una lista de artefactos de Angular que el componente necesita para cargarse correctamente , como otros componentes de Angular. La CLI de Angular añade RouterOutlet al componente principal de la aplicación por defecto. RouterOutlet se utiliza cuando necesitamos funciones de enrutamiento en una aplicación Angular. Aprenderemos a configurar el enrutamiento en el capítulo 9, "Navegación por aplicaciones con enrutamiento".
- URL de plantilla: Define la ruta de un archivo HTML externo que contiene la plantilla HTML del componente. Alternativamente, puede proporcionar la plantilla en línea usando la plantilla. propiedad.
- styleUrls: Define la ruta de un archivo de hoja de estilos CSS externo que contiene los estilos CSS del componente. Alternativamente, puede proporcionar los estilos en línea mediante la propiedad "styles" .

En aplicaciones creadas con versiones anteriores de Angular, es posible que notes que las importaciones Falta la propiedad en el decorador @Component . Esto se debe a que dichos componentes dependen de módulos Angular para proporcionar la funcionalidad necesaria.

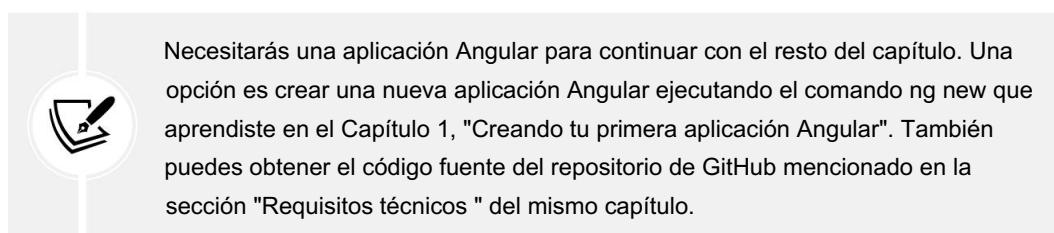


Sin embargo, a partir de Angular v16, se introdujo la propiedad independiente como alternativa a los módulos de Angular. Con Angular v19, los componentes independientes son ahora la opción predeterminada y se aplican en toda la estructura del proyecto. Este cambio significa que las aplicaciones creadas con Angular v19 utilizarán la matriz de importaciones en los componentes independientes de forma predeterminada, lo que supone una diferencia significativa con respecto a los componentes basados en módulos. arquitectura de versiones anteriores.

Ahora que hemos explorado la estructura de un componente Angular, aprenderemos a utilizar la CLI de Angular y a crear componentes por nosotros mismos.

Creación de componentes con Angular CLI

Además del componente principal de la aplicación, podemos crear otros componentes Angular que proporcionen una funcionalidad específica a la aplicación.



Necesitarás una aplicación Angular para continuar con el resto del capítulo. Una opción es crear una nueva aplicación Angular ejecutando el comando `ng new` que aprendiste en el Capítulo 1, "Creando tu primera aplicación Angular". También puedes obtener el código fuente del repositorio de GitHub mencionado en la sección "Requisitos técnicos" del mismo capítulo.

Para crear un nuevo componente en una aplicación Angular, usamos el comando `ng generate` de la CLI de Angular, pasando el nombre del componente como parámetro. Ejecutamos el siguiente comando dentro de la carpeta raíz del espacio de trabajo actual de la CLI de Angular:

```
ng genera la lista de productos de componentes
```

El comando anterior crea una carpeta dedicada para el componente llamado `product-list` que contiene todos los archivos necesarios:

- El archivo `product-list.component.css`, que todavía no contiene ningún estilo CSS.
- El archivo `product-list.component.html`, que contiene un elemento de párrafo que reproduce texto estático:

```
¡La lista de productos funciona!
```

- El archivo `product-list.component.spec.ts`, que contiene una prueba unitaria que verifica si el

El componente se puede crear correctamente:

```
importar { ComponentFixture, TestBed } desde '@angular/core/testing';
```

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

```
describe('ProductListComponent', () => {
  dejar componente: ProductListComponent;
  dejar que el acceso: ComponentFixture<ProductListComponent>;
```

```
beforeEach(async () =>
{ await TestBed.configureTestingModule({ imports:
[ProductListComponent]

}) .compileComponents();

accesorio = TestBed.createComponent(ProductListComponent);
componente = fixture.componentInstance;
fixture.detectChanges();
});

it('debería crear', () => {
esperar(componente).toBeTruthy();

});});
```

Aprenderemos más sobre las pruebas unitarias y su sintaxis en el Capítulo 13, Pruebas unitarias de aplicaciones angulares.

- El archivo product-list.component.ts , que contiene la lógica de presentación de nuestro componente:

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

@Component({ selector: 'lista-de-productos-de-la-aplicación',
importaciones:
[], templateUrl: './product-list.component.html',
styleUrl: './product-list.component.css' })

clase de exportación ProductListComponent {

}
```

En esta sección, nos centramos en la clase TypeScript de los componentes Angular, pero ¿cómo interactúan con su plantilla HTML?

En la siguiente sección, aprenderemos a mostrar la plantilla HTML de un componente de Angular en una página. También veremos cómo usar la sintaxis de plantilla de Angular para interactuar entre la clase TypeScript del componente y su plantilla HTML.

Interactuar con la plantilla

Como hemos aprendido, crear un componente Angular mediante la CLI de Angular implica generar un conjunto de archivos complementarios. Uno de estos archivos es la plantilla del componente, que contiene el contenido HTML que se muestra en la página. En esta sección, exploraremos cómo mostrar e interactuar con la plantilla a través de los siguientes temas:

- Carga de la plantilla del componente
- Visualización de datos de la clase de componente
- Dar estilo al componente
- Obtener datos de la plantilla

Comenzaremos nuestro viaje en la plantilla de componentes explorando cómo representamos un componente en la página web.

Cargando la plantilla del componente

Aprendimos que Angular usa la propiedad selector para cargar el componente en una plantilla HTML. Una aplicación Angular típica carga la plantilla del componente principal al iniciar la aplicación.

La etiqueta <app-root> que vimos en el Capítulo 1, Construyendo su primera aplicación Angular, es el selector del componente principal de la aplicación.

Para cargar un componente creado, como el componente de lista de productos, debemos agregar su selector dentro de una plantilla HTML. En este caso, lo cargaremos en la plantilla del componente principal de la aplicación:

1. Abra el archivo app.component.html y mueva el contenido de la etiqueta <style> en la aplicación.

archivo component.css .



Es más fácil de mantener y se considera una buena práctica tener todos los estilos CSS en un archivo separado.

2. Modifique el archivo app.component.html agregando la etiqueta <app-product-list> dentro del

Etiqueta <div> con la clase de contenido :

```
<div class="contenido">  
    <lista-de-productos-de-aplicaciones></lista-de-productos-de-aplicaciones>  
</div>
```



También podemos utilizar etiquetas autoencapsuladas, similares a los elementos HTML `<input>` y ``, para agregar el componente de lista de productos como `<app-product-list />`.

- Ejecute el comando `ng serve` en una ventana de terminal para iniciar la aplicación Angular.

El comando fallará y mostrará el siguiente error:

```
[ERROR] NG8001: 'app-product-list' no es un elemento conocido
```

Este error se produce porque el componente principal de la aplicación aún no reconoce el componente de lista de productos.

- Abra el archivo `app.component.ts` e importe la clase `ProductListComponent`:

```
importar { Component } desde '@angular/core';  
importar { RouterOutlet } desde '@angular/router';  
importar { ProductListComponent } desde './product-list/product-list.  
componente';  
  
@Componente({  
    selector: 'app-root',  
    importaciones: [RouterOutlet, ProductListComponent],  
    URL de plantilla: './app.component.html',  
    URL de estilo: './app.component.css'  
})  
clase de exportación AppComponent {  
    título = 'Mundo';  
}
```

Una vez compilada la aplicación, acceda a `http://localhost:4200` para obtener una vista previa . La página web muestra el texto estático de la plantilla del componente de lista de productos.

En las siguientes secciones, veremos cómo usar la sintaxis de plantilla de Angular e interactuar con ella mediante la clase TypeScript. Comenzaremos explorando cómo mostrar datos dinámicos definidos en la clase TypeScript del componente.

Visualización de datos de la clase de componente

Ya nos hemos topado con la interpolación para mostrar un valor de propiedad como texto desde la clase del componente a la plantilla:

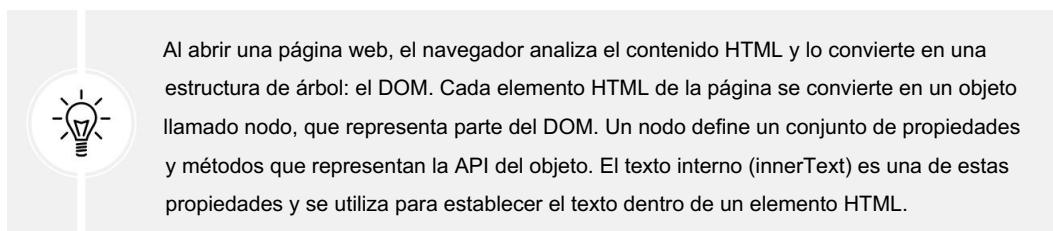
```
Hola, {{ título }}
```

Angular convierte la propiedad del componente de título en texto y lo muestra en la pantalla.

Una forma alternativa de realizar la interpolación es vincular la propiedad del título a la propiedad innerText del elemento HTML `<h1>`, un método llamado vinculación de propiedad:

```
<h1 [texto interno]="título"></h1>
```

En el fragmento anterior, enlazamos a la propiedad DOM de un elemento y no a su atributo HTML, como parece a simple vista. La propiedad entre corchetes se denomina propiedad objetivo y corresponde al elemento DOM al que queremos enlazar. La variable de la derecha se denomina expresión de plantilla y corresponde a la propiedad title del componente.



Para comprender mejor el funcionamiento del mecanismo de plantillas de Angular, primero debemos comprender cómo interactúa Angular con los atributos y las propiedades. Define atributos HTML para inicializar una propiedad del DOM y luego utiliza el enlace de datos para interactuar directamente con ella.

Para configurar el atributo de un elemento HTML, usamos la sintaxis attr. mediante la vinculación de propiedades seguida del nombre del atributo. Por ejemplo, para configurar el atributo de accesibilidad aria-label de un elemento HTML, escribiríamos lo siguiente:

```
<p [attr.aria-label]="miTexto"></p>
```

En el fragmento anterior, `myText` es una propiedad de un componente de Angular. Recuerde que la vinculación de propiedades interactúa con las propiedades de un componente de Angular. Por lo tanto, si quisieramos establecer el valor de la propiedad `innerText` directamente en el HTML, escribiríamos el valor del texto entre comillas simples:

```
<h1 [innerText]="'Mi título'"></h1>
```

En este caso, el valor pasado a la propiedad `innerText` es texto estático, no una propiedad del componente.

La vinculación de propiedades en Angular vincula los valores de propiedad de la clase TypeScript del componente a la plantilla. Como veremos a continuación, la sintaxis del flujo de control es adecuada para coordinar cómo se mostrarán dichos valores en la plantilla.

Controlar la representación de datos

La nueva sintaxis de flujo de control introducida en las últimas versiones del framework Angular permite manipular la representación de los datos en la plantilla del componente. Incluye un conjunto de bloques integrados que añaden las siguientes funciones a la sintaxis de la plantilla Angular:

- Visualización de datos de forma condicional
- Iterando a través de los datos
- Cambiar entre plantillas

En las siguientes secciones, exploraremos las capacidades anteriores, comenzando con la visualización de datos de componentes basados en una declaración condicional.

Visualización de datos condicionalmente

El bloque `@if` agrega o elimina un elemento HTML en el DOM basándose en la evaluación de una expresión.

Si la expresión se evalúa como verdadera, el elemento se inserta en el DOM. De lo contrario, se elimina.

Ilustraremos el uso del bloque `@if` con un ejemplo:

1. Ejecute el siguiente comando para crear una interfaz para productos:

```
ng genera un producto de interfaz
```

2. Abra el archivo `product.ts` y agregue las siguientes propiedades:

```
Interfaz de exportación Producto {  
    id: numero;  
    título: cadena;  
}
```

La interfaz de Producto define la estructura de un objeto Producto .

3. Abra el archivo app.component.css y mueva los estilos CSS que contienen los valores h1 y p .

lectores en el archivo product-list.component.css .

4. Abra el archivo product-list.component.ts y cree una matriz de productos vacía :

```
importar { Component } de '@angular/core'; importar
{ Producto } de '../product';

@Component({ selector: 'app-product-
list',
importaciones: [], templateUrl: './product-list.component.html',
styleUrl: './product-list.component.css' }) exportar

clase ProductListComponent {
    productos: Producto[] = [];
}
```

La matriz de productos se utilizará para almacenar una lista de objetos de producto .

5. Abra el archivo product-list.component.html y reemplace su contenido con lo siguiente
retazo:

```
@if (productos.longitud > 0) {
    Productos ({{products.length}})
}
```

El elemento <h1> de la plantilla HTML anterior se muestra en pantalla cuando la matriz de productos no está vacía. De lo contrario, se elimina por completo.

6. El bloque @if se comporta de forma similar a una sentencia if de JavaScript . Por lo tanto, podemos añadir una sección @else en la plantilla del componente para ejecutar lógica personalizada cuando aún no hay productos:

```
@if (productos.longitud > 0) {
    <h1>Productos ({{products.length}})</h1> } @else
    { <p>jNo
        se encontraron productos!</p>
}
```

Si tuviéramos una condición adicional que nos gustaría evaluar, podríamos usar una sección @else if :



```
@if (productos.longitud > 0) {  
    Productos ({{products.length}})  
} @else if (productos.longitud === 100) {  
    <span>  
        Haga clic en <a>Cargar más</a> para ver más productos.  
    </span>  
} @else  
{ <p>¡No  
    se encontraron productos!</p>  
}
```

7. Ejecute el comando ng serve para obtener una vista previa de la aplicación hasta el momento:



No products found!

Figura 3.2: Salida de la aplicación

En aplicaciones creadas con versiones anteriores de Angular donde la sintaxis de flujo de control no está disponible, es posible que notes que se utilizó la sintaxis *ngIf para mostrar datos condicionales:

```
<h1 *ngIf="products.length > 0"> Productos  
    ({{products.length}})  
</h1>
```

* ngIf es una directiva de Angular con el mismo comportamiento que el bloque @if .

Aprenderemos a crear directivas personalizadas de Angular en el siguiente capítulo.

Sin embargo, se recomienda encarecidamente utilizar el bloque @if por las siguientes razones:

- Hace que las plantillas sean mucho más legibles
- La sintaxis es más cercana a JavaScript y es más fácil de recordar
- Está integrado en el marco y está disponible de inmediato, lo que da como resultado tamaños de paquete más pequeños.

Puede encontrar más información sobre *ngIf en <https://angular.dev/guide/directivas#agregar-o-eliminar-un-elemento-con-ngif>.

La aplicación que hemos creado no muestra ningún dato porque la matriz de productos está vacía. En la siguiente sección, aprenderemos cómo agregar y mostrar datos de productos en la lista de productos. componente.

Iterando a través de los datos

El bloque @for nos permite recorrer una colección de elementos y generar una plantilla para cada uno, donde podemos definir marcadores de posición convenientes para interpolar los datos de los elementos. Cada plantilla generada tiene como ámbito el contexto externo, donde se coloca la directiva loop para que podamos acceder a otras vinculaciones.

Podemos pensar en el bloque @for como el bucle for de JavaScript pero para plantillas HTML.

Podemos usar el bloque @for para mostrar la lista de productos en nuestro componente de la siguiente manera:

1. Abra el archivo app.component.css y mueva los estilos CSS que contienen los selectores .pill-group, .pill y .pill:hover en el archivo product-list.component.css .
2. Modifique la matriz de productos en la clase ProductListComponent de la lista de productos. archivo component.ts para que contenga los siguientes datos:

```
clase de exportación ProductListComponent {  
    productos: Producto[] = [  
        { id: 1, título: 'Teclado' },  
        { id: 2, título: 'Micrófono' },  
        { id: 3, título: 'Cámara web' },  
        { id: 4, título: 'Tableta' }  
    ];  
}
```

3. Abra el archivo product-list.component.html y agregue el siguiente fragmento después del @ si bloque:

```
<ul class="grupo de píldoras">  
    @for (producto de productos; seguimiento producto.id) {  
        <li class="pill">{{producto.título}}</li>  
    }  
</ul>
```

En el código anterior, usamos el bloque @for y convertimos cada elemento obtenido de la matriz de productos en una variable de producto llamada variable de entrada de plantilla. Referenciamos la variable de plantilla en nuestro HTML enlazando su propiedad title mediante la sintaxis de interpolación de Angular.

Durante la ejecución del bloque `@for`, los datos pueden cambiar, se pueden añadir, mover o eliminar elementos HTML, e incluso se puede reemplazar la lista completa. Angular debe sincronizar los cambios de datos con el árbol DOM conectando el array iterado con su elemento DOM correspondiente. Este proceso puede resultar muy lento y costoso, y eventualmente resultar en un bajo rendimiento. Para ello, Angular utiliza la propiedad `track`, que registra los cambios de datos. En nuestro caso, la propiedad `track` define el nombre de la variable "product", que se utilizará para registrar cada elemento del array "products".

4. Ejecute el comando `ng serve` para obtener una vista previa de la aplicación:

Products (4)

Keyboard

Microphone

Web camera

Tablet

Figura 3.3: Lista de productos

5. El bloque `@for` permite añadir una sección `@empty`, que se ejecuta cuando el array de elementos está vacío.

Podemos refactorizar nuestro código eliminando la sección `@else` del bloque `@if` y añadiendo una sección `@empty` como se indica a continuación:

```
@if (productos.longitud > 0) {
    Productos ({{productos.length}})
}

<ul class="grupo de píldoras">
    @for (producto de productos; seguimiento producto.id) {
        <li class="pill">{{producto.título}}</li>
    } @vacío {
        <p>¡No se encontraron productos!</p>
    }
</ul>
```

El bloque @for permite observar cambios en la colección subyacente y añadir, eliminar u ordenar las plantillas renderizadas a medida que se añaden, eliminan o reordenan elementos en la colección. También permite realizar un seguimiento de otras propiedades útiles. Podemos usar la versión extendida del bloque @for con la siguiente sintaxis:

```
@for (producto de productos; seguimiento producto.id; dejar variable=propiedad) {}
```

La variable es una variable de entrada de plantilla a la que podemos hacer referencia más adelante en nuestra plantilla. La propiedad Puede tener los siguientes valores:

- \$count: Indica el número de elementos en la matriz
- \$index: Indica el índice del elemento en la matriz
- \$first/\$last: Indica si el elemento actual es el primero o el último en la matriz
- \$even/\$odd: Indica si el índice del elemento en la matriz es par o impar



Podemos utilizar las propiedades anteriores directamente o declarando un alias, como se ve en el siguiente ejemplo.

En el siguiente fragmento, Angular asigna el valor de la propiedad \$index a la variable de entrada i. La variable i se utiliza más adelante en la plantilla para mostrar cada producto como una lista numerada:

```
@for (producto de productos; rastrear producto.id; dejar i = $índice) {  
  <li class="pill">{{i+1}}. {{producto.título}}</li>  
}
```



Utilice la propiedad \$index en la variable de seguimiento si no está seguro de cuál seleccionar de los datos del objeto. Además, se recomienda usarla cuando el objeto no tiene ninguna propiedad única y no modifica el orden de la lista eliminando, añadiendo o moviendo elementos.

En aplicaciones creadas con versiones anteriores de Angular, es posible que observe la siguiente sintaxis para iterar sobre colecciones:



```
<ul class="grupo de píldoras">
  <li class="pill" *ngFor="let producto de productos">
    {{producto.título}}
  </li>
</ul>
```

*ngFor es una directiva de Angular que funciona de forma similar al bloque @for . Sin embargo, se recomienda encarecidamente usar @for por las mismas razones mencionadas para @. Si bloquea en la sección anterior.

Puede encontrar más información sobre *ngFor en <https://angular.dev/guide/directives#listado-de-elementos-con-ngfor>.

El último bloque de la sintaxis del flujo de control que cubriremos es el bloque @switch en la siguiente sección.

Cambiar entre plantillas

El bloque @switch cambia entre partes de la plantilla del componente y muestra cada una en función de un valor definido.

Puedes pensar en @switch como la sentencia switch de JavaScript . Consta de las siguientes secciones:

- @switch: Define la propiedad que queremos comprobar al aplicar el bloque
- @case: agrega o elimina una plantilla del árbol DOM dependiendo del valor de la propiedad definida en el bloque @switch
- @default: Agrega una plantilla al árbol DOM si el valor de la propiedad definida en @

El bloque switch no cumple ninguna declaración @case

Aprenderemos a usar el bloque @switch mostrando un emoji diferente según el título del producto. Abra el archivo product-list.component.html y modifique el bloque @for para que incluya el siguiente bloque @switch :

```
<ul class="grupo de píldoras">
  @for (producto de productos; seguimiento producto.id) {
    <li class="píldora">
      @switch (producto.título) {
        @case ('Teclado') {  }
```

```

@case ('Micrófono') 
{ @default {  }
}

{{producto.título}}
```



```

} @vacío {
    <p>¡No se encontraron productos!</p>
}

```

```
</ul>
```

El bloque `@switch` evalúa la propiedad `title` de cada producto. Cuando encuentra una coincidencia, activa la sección `@case` correspondiente . Si el valor de la propiedad `title` no coincide con ninguna `@ Sección de caso` , se activa la sección `@default` .

 En aplicaciones creadas con versiones anteriores de Angular, es posible que notes la siguiente sintaxis para cambiar partes de la plantilla:

```

<div [ngSwitch]="producto.título">
    <p *ngSwitchCase=""Teclado""> </p> 
    <p *ngSwitchCase=""Micrófono""> </p> 
    <p *ngSwitchDefault> </p> 
</div>
```

`[ngSwitch]` es una directiva Angular con el mismo comportamiento que el bloque `@switch` . Sin embargo, se recomienda encarecidamente utilizar `@switch` por las mismas razones mencionadas sobre el bloque `@if` en la sección anterior.

Puede encontrar más información sobre `[ngSwitch]` en <https://angular.dev/guide/directives#cambio-de-casos-con-ngswitch>.

La simplicidad y la ergonomía mejorada de la sintaxis del flujo de control han permitido la introducción del bloque `@defer` en el framework Angular. Este bloque ayuda a mejorar la experiencia de usuario (UX) y el rendimiento de la aplicación al cargar partes de la plantilla del componente de forma asíncrona. Aprenderemos más en el capítulo 15, Optimización del rendimiento de la aplicación.

En esta sección, aprendimos cómo aprovechar la sintaxis del flujo de control y coordinar cómo se mostrarán los datos en la plantilla del componente.



Si desea utilizar esta sintaxis en aplicaciones que ya utilizan el antiguo enfoque directivo , puede ejecutar la migración de CLI de Angular que se describe en <https://angular.dev/referencia/migraciones/flujo-de-control>.

Como aprenderemos en la siguiente sección, la vinculación de propiedades en el marco Angular aplica estilos y clases CSS en las plantillas Angular.

Dar estilo al componente

Los estilos en una aplicación web se pueden aplicar utilizando el atributo de clase o estilo , o ambos. un elemento HTML:

```
<p class="star"></p> <p  
style="color: verdeamarillo"></p>
```

El marco Angular proporciona dos tipos de vinculación de propiedades:

- Vinculación de clases
- Encuadernación de estilo

Comencemos nuestro viaje sobre el estilo de componentes con vinculación de clases en la siguiente sección.

Enlace de clases

Podemos aplicar una sola clase a un elemento HTML utilizando la siguiente sintaxis:

```
<p [class.star]="isLiked"></p>
```

En el fragmento anterior, la clase estrella se agregaría al elemento de párrafo cuando se indique isLiked. La expresión es verdadera. De lo contrario, se eliminará del elemento. Si queremos aplicar varias clases CSS simultáneamente, podemos usar la siguiente sintaxis:

```
<p [clase]="ClasesActuales"></p>
```

La variable currentClasses es una propiedad del componente. El valor de una expresión utilizada en un enlace de clase puede ser uno de los siguientes:

- Una cadena de nombres de clase delimitada por espacios, como 'star active'.
- Un objeto con claves como nombres de clase y valores como condiciones booleanas para cada clave. Se añade una clase al elemento cuando el valor de la clave, con su nombre, se evalúa como verdadero. De lo contrario, la clase se elimina del elemento:

```
clasesactuales = {
```

```
estrella: cierto,  
activo: falso  
};
```

En lugar de diseñar nuestros elementos usando clases CSS, podemos establecer estilos directamente con el enlace de estilo.

Encuadernación de estilo

Al igual que la vinculación de clases, podemos aplicar estilos individuales o múltiples simultáneamente usando una vinculación de estilos.

Se puede establecer un solo estilo en un elemento HTML utilizando la siguiente sintaxis:

```
<p [style.color]=""verdeamarillo""></p>
```

En el fragmento anterior, el elemento de párrafo tendrá un color verde-amarillo . Algunos estilos se pueden ampliar en la encuadernación, como el ancho del elemento de párrafo, que podemos definir junto con la unidad de medida:

```
<p [style.width.px]="100"></p>
```

El elemento de párrafo tendrá una longitud de 100 píxeles. Si necesitamos alternar varios estilos a la vez, podemos usar la sintaxis de objeto:

```
<p [style]="currentStyles"></p>
```

La variable currentStyles es una propiedad del componente. El valor de una expresión utilizada en un enlace de estilo puede ser uno de los siguientes:

- Una cadena con estilos separados por punto y coma como 'color: verdeamarillo; ancho:

100 píxeles

- Un objeto donde sus claves son los nombres de los estilos y los valores son los valores de estilo reales:

```
currentStyles = { color:  
  'verdeamarillo',  
  ancho: '100px'  
};
```

Los enlaces de clases y estilos son funciones potentes que Angular ofrece de forma predeterminada. Junto con la configuración de estilos CSS que podemos definir en el decorador @Component , ofrece infinitas posibilidades para aplicar estilos a los componentes de Angular. Una función igualmente atractiva es la posibilidad de leer datos de una plantilla en la clase del componente, que veremos a continuación.

Obtener datos de la plantilla

En la sección anterior, aprendimos a usar la vinculación de propiedades para mostrar datos de la clase del componente. En situaciones reales, los datos suelen fluir bidireccionalmente a través de los componentes. Para devolver los datos de la plantilla a la clase del componente, utilizamos una técnica llamada vinculación de eventos. Aprenderemos cómo usar la vinculación de eventos notificando a la clase de componente cuando un producto ha sido seleccionado de la lista:

1. Abra el archivo product-list.component.ts y agregue una propiedad selectedProduct :

```
selectedProduct: Producto | indefinido;
```

2. Abra el archivo product-list.component.html y use la sintaxis de interpolación para mostrar el producto seleccionado si existe:

```
@if (productoseleccionado) {  
    <p>Usted seleccionó:  
        <strong>{{producto seleccionado.título}}</strong>  
    </p>  
}
```

3. Agregue un enlace de evento de clic en la etiqueta para establecer el producto seleccionado en el valor actual. variable de producto del bloque @for :

```
@for (producto de productos; seguimiento producto.id) {  
    <li class="pill" (clic)="productoSeleccionado = producto">  
        @switch (producto.título) {  
            @case ('Teclado')  }  
            { @case ('Micrófono')  }  
            @por defecto {  }  
        }  
        {{producto.título}}  
    </li>  
}
```

4. Ejecute ng serve para iniciar la aplicación y haga clic en un producto de la lista:

Products (4)

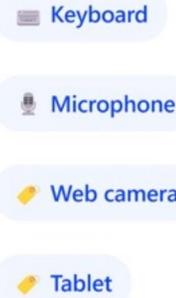


Figura 3.4: Selección de productos

Un enlace de eventos detecta eventos DOM en el elemento HTML de destino y responde a ellos interactuando con los miembros de la clase del componente. El evento entre paréntesis se denomina evento de destino y es el evento que estamos detectando. La expresión de la derecha se denomina declaración de plantilla e interactúa con la clase del componente. El enlace de eventos en Angular admite todos los eventos DOM nativos disponibles en <https://developer.mozilla.org/docs/Web/Events>.

La interacción de una plantilla de componente con su clase TypeScript correspondiente se resume en el siguiente diagrama:

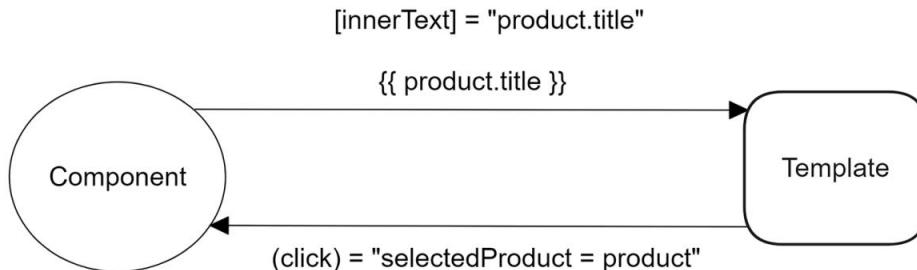


Figura 3.5: Interacción componente-plantilla

El mismo principio que seguimos para interactuar con la plantilla y la clase del componente se puede utilizar al comunicarnos entre componentes.

Intercomunicación de componentes

Los componentes de Angular exponen una API pública que les permite comunicarse con otros componentes. Esta API abarca las propiedades de entrada, que utilizamos para alimentar el componente con datos. También expone las propiedades de salida a las que podemos vincular detectores de eventos, obteniendo así información oportuna sobre los cambios en el estado del componente.

En esta sección, aprenderemos cómo Angular resuelve el problema de inyectar y extraer datos de los componentes a través de ejemplos rápidos y sencillos.

Pasar datos mediante un enlace de entrada

La aplicación muestra actualmente la lista de productos y los detalles del producto seleccionado en el mismo componente. Para aprender a transferir datos entre diferentes componentes, crearemos un nuevo componente Angular que mostrará los detalles del producto seleccionado. Los datos que representan los detalles específicos del producto se transferirán dinámicamente desde el componente de lista de productos.

Comenzaremos creando y configurando el componente para mostrar los detalles del producto:

1. Ejecute el siguiente comando CLI de Angular para crear el nuevo componente Angular:

```
ng genera el componente detalle del producto
```

2. Abra el archivo product-detail.component.ts y modifique las declaraciones de importación según corresponda.
ingly:

```
importar { Componente, entrada } desde '@angular/core';
importar { Producto } de '../producto';
```

La función de entrada es parte de la API de señales y se utiliza cuando queremos pasar datos de un componente a otro componente.



Aprenderemos más sobre la API de señales en el Capítulo 7, Seguimiento del estado de la aplicación con señales.

3. Defina una propiedad de producto en la clase ProductDetailComponent que utilice la entrada función:

```
clase de exportación ProductDetailComponent {
  producto = entrada<Producto>();
}
```



En versiones anteriores de Angular, usamos el decorador `@Input` para pasar datos entre componentes.

Puedes obtener más información en <https://angular.dev/guide/componentes/entradas>.

4. Abra el archivo `product-detail.component.html` y agregue el siguiente contenido:

```
@if (producto()) {
  <p>Usted seleccionó:</p>
  <strong>{{producto()!.título}}</strong>
</p>
}
```

En el fragmento anterior, utilizamos un bloque `@if` para verificar si la propiedad de entrada del producto se ha configurado antes de mostrar su título.

5. Abra el archivo `product-list.component.ts` e importe la clase `ProductDetailComponent`:

```
importar { Componente } desde '@angular/core';
importar { Producto } de './producto';
importar { ProductDetailComponent } de '../product-detail/product-detail.component';

@Component({
  selector: 'lista-de-productos-de-aplicaciones',
  importaciones: [ProductDetailComponent],
  templateUrl: './lista-de-productos.componente.html',
  styleUrls: ['./lista-de-productos.componente.css']
})
```

6. Finalmente, reemplace el último bloque `@if` en el archivo `product-list.component.html` con el siguiente fragmento:

```
<app-product-detail [producto]="producto seleccionado"></app-product-
detalle>
```

En el fragmento anterior, utilizamos la vinculación de propiedades para vincular el valor del producto seleccionado. Propiedad en la propiedad de entrada del producto del componente de detalle del producto. Este enfoque se denomina enlace de entrada.

Si ejecutamos la aplicación y hacemos clic en un producto de la lista, veremos que la selección de productos continúa funcionando como se esperaba.

El bloque @if en la plantilla del componente de detalle del producto implica que la propiedad de entrada del producto es obligatoria; de lo contrario, no muestra su título. Angular desconoce si el componente de lista de productos pasa un valor para la vinculación de entrada del producto durante la compilación. Si queremos aplicar esta regla durante la compilación, podemos definir una propiedad de entrada según sea necesario:

```
producto = entrada.required<Producto>();
```

Según el fragmento anterior, si el componente de lista de productos no pasa un valor para la propiedad de entrada del producto , el compilador Angular arrojará el siguiente error:

```
[ERROR] NG8008: Se debe especificar la entrada requerida 'producto'  
del componente ProductDetailComponent.
```

¡Listo! Hemos transferido datos correctamente de un componente a otro. En la siguiente sección, aprenderemos a escuchar eventos en un componente y a responder a ellos.

Escuchar eventos mediante un enlace de salida

Aprendimos que la vinculación de entrada se utiliza para transferir datos entre componentes. Este método es aplicable en escenarios con dos componentes: uno que actúa como componente principal y el otro como componente secundario. ¿Qué ocurre si queremos comunicarnos en sentido inverso, del componente secundario al principal? ¿Cómo notificamos al componente principal sobre acciones específicas en el componente secundario?

Imagine un escenario en el que el componente de detalle del producto debería tener un botón para añadir el producto actual al carrito de compras. El carrito de compras sería una propiedad del componente de lista de productos. ¿Cómo notificaría el componente de detalle del producto al componente de lista de productos que se hizo clic en el botón? Veamos cómo implementaríamos esta funcionalidad en nuestra aplicación:

1. Abra el archivo product-detail.component.ts e importe la función de salida desde el

Paquete npm de @angular/core :

```
importar { Component, entrada, salida } de '@angular/core';
```

La función de salida se utiliza cuando queremos crear eventos que se activarán desde un componente a otro.

2. Defina una nueva propiedad de componente dentro de la clase ProductDetailComponent que utilice la función de salida :

```
añadido = salida();
```



En versiones anteriores de Angular, usamos el decorador `@Output` para activar eventos entre componentes. Puedes obtener más información en <https://angular.dev/guia/componentes/salidas>.

3. En la misma clase TypeScript, cree el siguiente método:

```
addToCart()
  { this.added.emit();
  }
```

El método `addToCart` llama al método `emit` en el evento de salida añadido que creamos en el paso anterior. El método `emit` activa un evento y notifica a cualquier componente que lo esté escuchando.

4. Ahora, agregue un elemento `<button>` en la plantilla del componente y vincule su evento de clic al Método `addToCart` :

```
@if (producto()) {
  <p>Usted seleccionó:
    <strong>{{producto()!.título}}</strong> </p>
  Añadir al carrito
}
```

5. Abra el archivo `product-detail.component.css` y agregue los siguientes estilos CSS que se aplicarán al elemento `<button>` :

```
botón {
  pantalla: flex;
  align-items: center; --
  button-accent: var(--azul brillante); fondo:
    mezcla de colores(en srgb, var(--button-accent) 65%,
  transparente);
  color: blanco;
  relleno en línea: 0,75rem;
  bloque de relleno: 0,375rem;
  radio del borde: 0,5rem;
  borde: 0;
  transición: fondo 0.3s facilidad; familia-de-
  fuentes: var(--inter-font);
```

```

    tamaño de fuente: 0.875rem;
    estilo de fuente: normal;
    peso de fuente: 500;
    altura de línea: 1.4rem;
    espacio entre letras: -0.00875rem;
    cursor: puntero;
}

botón:hover {
    fondo: mezcla de colores (en srgb, var (--button-accent) 50%,
transparente);
}

```

6. ¡Ya casi terminamos! Ahora, necesitamos conectar el enlace en el componente de lista de productos para que ambos componentes puedan comunicarse. Abra el archivo product-list.component.ts y cree el siguiente método:

```

al agregar() {
    alert(`${this.selectedProduct?.title} ¡ agregado al carrito!`);
}

```

En el fragmento anterior, utilizamos el método de alerta nativo del navegador para mostrar un cuadro de diálogo al usuario.

7. Finalmente, modifique la etiqueta <app-product-detail> en product-list.component.html

archivo de la siguiente manera:

```

<detalle-del Producto-de-la-aplicación
    [producto]="ProductoSeleccionado"
    (añadido)="onAdded()"
></app-product-detail>

```

En el fragmento anterior, usamos la vinculación de eventos para enlazar el método onAdded con la propiedad de salida agregada del componente de detalle del producto. Este enfoque se denomina vinculación de salida.

Si seleccionamos un producto de la lista y hacemos clic en el botón Añadir al carrito , aparecerá un cuadro de diálogo con un mensaje como el siguiente:

¡Cámara web añadida al carrito!

Puede ver una descripción general del mecanismo de comunicación de componentes que hemos discutido en el siguiente diagrama:

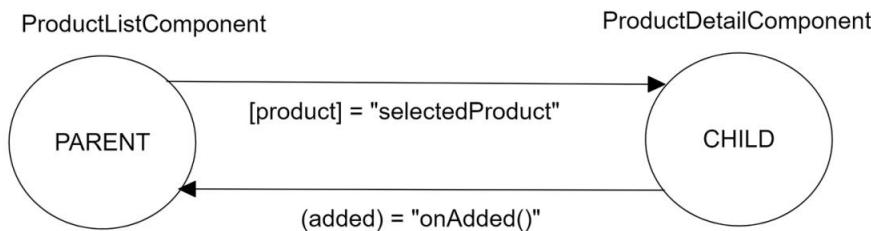


Figura 3.6: Intercomunicación de componentes

El evento de salida del componente de detalle del producto no hace nada más que emitir un evento al componente principal. Sin embargo, podemos usarlo para pasar datos arbitrarios mediante el método de emisión , como veremos en la siguiente sección.

Emisión de datos a través de eventos personalizados

El método de emisión de un evento de salida puede aceptar cualquier dato para pasarlo al componente principal. Es recomendable definir el tipo de dato que se puede pasar para aplicar la comprobación de tipos estática.

Actualmente, el componente de lista de productos ya reconoce el producto seleccionado. Supongamos que solo lo detecta después de que el usuario haga clic en el botón "Añadir al carrito" .

1. Abra el archivo product-detail.component.ts y use genéricos para declarar el tipo de datos

que se pasará al componente de lista de productos:

```
añadido = salida<Producto>();
```

2. Modifique el método addToCart para que el método de emisión pase el valor seleccionado actualmente.

producto:

```
añadir a la cesta() {
  este.añadido.emit(este.producto());
}
```

3. Abra el archivo product-list.component.html y pase la variable \$event en onAdded método:

```
<detalle-del-producto-de-la-aplicación
  [producto] = "ProductoSeleccionado"
  (añadido) = "onAdded($evento)"
></detalle-del-producto-de-la-aplicación>
```

El objeto \$event es una palabra clave reservada en Angular que contiene los datos de carga útil de un emisor de eventos desde un enlace de salida, en nuestro caso, un objeto Producto .

4. Abra el archivo product-list.component.ts y cambie la firma de onAdded

método en consecuencia:

```
onAdded(producto: Producto) {  
  alert(`#${product.title} agregado al carrito!`);  
}
```

Como vimos, los enlaces de eventos de salida son una excelente manera de notificar a un componente principal sobre un cambio en el estado del componente o enviar cualquier dato.

Además de utilizar los enlaces de entrada y salida para comunicarnos con los componentes, podemos acceder a sus propiedades y métodos directamente utilizando variables de referencia de plantilla locales.

Variables de referencia locales en plantillas

Hemos visto cómo vincular datos a nuestras plantillas mediante interpolación con la sintaxis de llaves dobles. Además, a menudo encontramos identificadores con nombre precedidos por un símbolo de almohadilla (#) en los elementos de nuestros componentes o incluso en elementos HTML estándar. Estos identificadores de referencia, es decir, las variables de referencia de plantilla, hacen referencia a los componentes marcados con ellos en nuestras vistas de plantilla y luego acceden a ellos mediante programación. Los componentes también pueden usarlos para hacer referencia a otros elementos en el DOM y acceder a sus propiedades.

Hemos aprendido cómo se comunican los componentes escuchando los eventos emitidos mediante la vinculación de salida o pasando datos mediante la vinculación de entrada. Pero ¿qué pasaría si pudiéramos inspeccionar el componente a fondo, o al menos sus propiedades y métodos expuestos, y acceder a ellos sin pasar por las vinculaciones de entrada y salida? Establecer una referencia local en el componente abre la puerta a su API pública.



La API pública de un componente consta de todos los miembros públicos de la clase TypeScript.

Podemos declarar una variable de referencia de plantilla para el componente de detalle del producto en el archivo product-list.component.html de la siguiente manera:

```
<detalle-del-producto-de-la-aplicación  
#DetalleDeProducto  
[producto]="ProductoSeleccionado"
```

```
(añadido)="onAdded()"
```

```
></detalle-del-producto-de-la-aplicación>
```

A partir de ese momento, podremos acceder directamente a los miembros del componente e incluso enlazarlos en otras ubicaciones de la plantilla, como por ejemplo mostrando el título del producto:

```
<span>{{productDetail.product()!.title}}
```

De esta manera, no necesitamos depender de las propiedades de entrada y salida y podemos manipular el valor de dichas propiedades.



El enfoque de variable de referencia local es particularmente útil cuando se utilizan bibliotecas en las que no podemos controlar los componentes secundarios para agregar propiedades de enlace de entrada o salida .

Hemos explicado principalmente cómo la clase del componente interactúa con su plantilla u otros componentes, pero apenas nos hemos preocupado por su estilo. A continuación, lo exploraremos con más detalle.

Encapsulando estilos CSS

Podemos definir estilos CSS dentro de nuestros componentes para encapsular mejor nuestro código y hacerlo más reutilizable. En la sección "Creando nuestro primer componente" , aprendimos a definir estilos CSS para un componente usando un archivo CSS externo mediante la propiedad styleUrl o definiéndolos dentro del archivo del componente TypeScript con la propiedad designs .



Las reglas habituales de especificidad de CSS rigen en ambos sentidos: <https://developer.mozilla.org/docs/Web/CSS/Especificidad>.

Gracias al estilo con alcance, la gestión y especificidad de CSS se vuelven muy fáciles en los navegadores que Admite shadow DOM. Los estilos CSS se aplican a los elementos contenidos en el componente, pero no se extienden más allá de sus límites.



Puede encontrar más detalles sobre shadow DOM en https://developer.mozilla.org/docs/Web/API/Componentes_web/Usando_shadow_DOM.

Además, Angular integra hojas de estilo en el elemento `<head>` de una página web para que puedan afectar a otros elementos de nuestra aplicación. Podemos configurar diferentes niveles de encapsulación de vistas para evitar que esto ocurra.

La encapsulación de vistas es la forma en que Angular necesita gestionar el alcance de CSS dentro del componente. Podemos cambiarla configurando la propiedad de encapsulación del decorador `@Component` en uno de los siguientes valores de la enumeración `ViewEncapsulation` :

- **Emulado:** Implica una emulación del alcance nativo en shadow DOM mediante el aislamiento de las reglas CSS bajo un selector específico que apunta a un componente. Esta opción es la preferida para garantizar que los estilos del componente no se filren fuera del componente ni se vean afectados por otros estilos externos. Es el comportamiento predeterminado en los proyectos de Angular CLI.
- **Native:** utiliza el mecanismo de encapsulación DOM de sombra nativo del renderizador que Sólo funciona en navegadores que admiten shadow DOM.
- **Ninguno:** No se proporciona encapsulación de plantillas ni estilos. Los estilos se inyectan tal como se agregaron al elemento `<head>` del documento. Es la única opción si se usa shadow. Los navegadores compatibles con DOM no están involucrados.

Exploraremos las opciones Emulado y Ninguno debido a su soporte extendido usando un ejemplo:

1. Abra el archivo `product-detail.component.html` y adjunte el contenido del bloque `@if` en un elemento `<div>` :

```
@if (producto()) {  
  <div>  
    <p>Usted seleccionó:  
      <strong>{{producto()!.título}}</strong>  
    </p>  
    Añadir al carrito  
  </div>  
}
```

2. Abra el archivo `product-detail.component.css` y agregue un estilo CSS para cambiar el borde de un elemento `<div>` :

```
div {  
  relleno en línea: 0,75rem;  
  bloque de relleno: 0,375rem;  
  borde: 2px discontinuo;  
}
```

3. Ejecute la aplicación usando el comando ng serve y observe que el detalle del producto

El componente tiene un borde discontinuo a su alrededor cuando selecciona un producto:

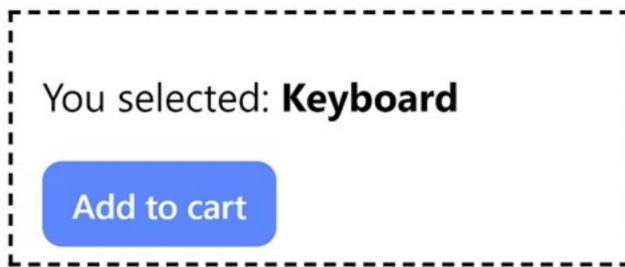


Figura 3.7: Detalles del producto

El estilo no afectó al elemento <div> en el archivo app.component.html porque la encapsulación predeterminada abarca todos los estilos CSS definidos para el componente específico.



4. Abra el archivo product-detail.component.ts y configure la encapsulación del componente en

VerEncapsulación.Ninguno:

```
importar { Component, entrada, salida, ViewEncapsulation } de '@ angular/core';
importar { Producto }
de './product';

@Component({
selector: 'app-product-detail', importa: [],

templateUrl: './product-detail.component.html', styleUrl: './product-
detail.component.css', encapsulación:
ViewEncapsulation.None })
```

La salida de la aplicación debería verse así:

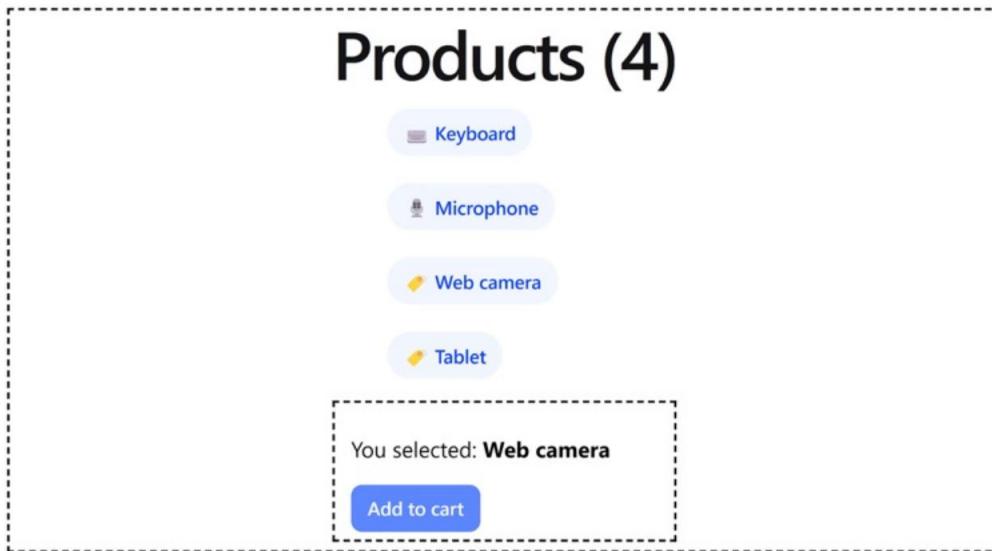


Figura 3.8: Sin encapsulación de vista

En la imagen anterior, el estilo CSS se filtró al árbol de componentes y afectó al elemento <div> del componente principal de la aplicación.

La encapsulación de vistas puede resolver muchos problemas al diseñar nuestros componentes. Sin embargo, debe usarse con precaución, ya que, como ya vimos, los estilos CSS pueden filtrarse en partes de la aplicación y producir efectos no deseados.

La estrategia de detección de cambios es otra propiedad muy potente del decorador @Component . Analicémosla a continuación.

Decidir sobre una estrategia de detección de cambios

La detección de cambios es el mecanismo que Angular utiliza internamente para detectar cambios en las propiedades de los componentes y reflejarlos en la vista. Se activa ante eventos específicos, como cuando el usuario hace clic en un botón, se completa una solicitud asíncrona o se establece un tiempo de espera.

Se ejecuta el método setInterval. Angular utiliza un proceso llamado "monkey patching" para modificar dichos eventos sobrescribiendo su comportamiento predeterminado mediante una biblioteca llamada Zone.js.

Cada componente cuenta con un detector de cambios que detecta si se ha producido un cambio en sus propiedades comparando el valor actual de una propiedad con el anterior. Si existen diferencias, aplica el cambio a la plantilla del componente. En el componente de detalle del producto, cuando la propiedad de entrada del producto cambia como resultado de un evento mencionado anteriormente, el mecanismo de detección de cambios se ejecuta para este componente y actualiza la plantilla en consecuencia.

Sin embargo, existen casos en los que este comportamiento no es deseable, como en el caso de componentes que renderizan una gran cantidad de datos. En ese caso, el mecanismo predeterminado de detección de cambios resulta insuficiente, ya que puede generar cuellos de botella en el rendimiento de la aplicación. Como alternativa, podríamos usar la propiedad changeDetection del decorador @Component , que determina la estrategia que seguirá el componente para la detección de cambios.

Aprenderemos cómo utilizar un mecanismo de detección de cambios al perilar nuestra aplicación Angular con Angular DevTools:

1. Abra el archivo product-detail.component.ts y cree una propiedad getter que devuelva el título del producto actual:

```
obtener productTitle() {  
    devuelve este.producto()!.título;  
}
```

2. Abra el archivo product-detail.component.html y reemplace la expresión product.title dentro de la etiqueta con el productTitle:

```
@if (producto()) {  
    <p>Usted seleccionó:  
        <strong>{{título del producto}}</strong>  
    </p>  
    Añadir al carrito  
}
```

3. Ejecute la aplicación usando el comando ng serve y obtenga una vista previa en <http://localhost:4200>.
4. Inicie Angular DevTools, seleccione la pestaña Profiler y haga clic en el botón Iniciar grabación para Comience a perilar la aplicación Angular.

5. Haga clic en el producto Teclado de la lista de productos y seleccione la primera barra en la barra

Gráfico para revisar la detección de cambios:

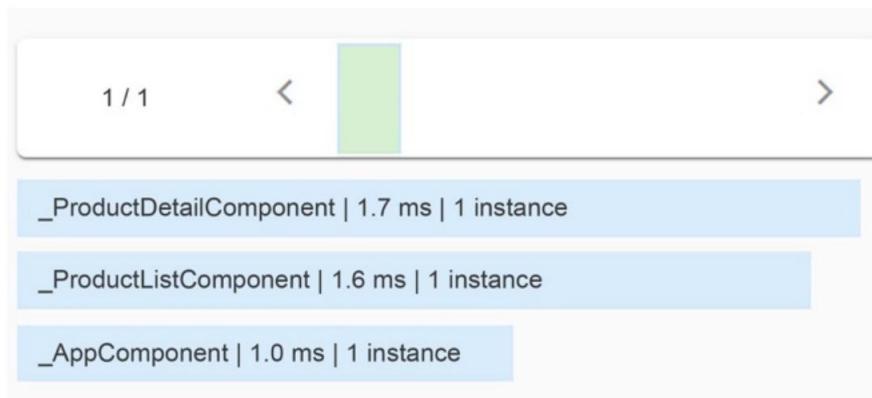


Figura 3.9: Gráfico de barras de detección de cambios

En la imagen anterior, podemos ver que la detección de cambios se activa para cada componente en el árbol de componentes de la aplicación.

6. Haga clic en el botón Agregar al carrito y seleccione la segunda barra en el gráfico de barras:

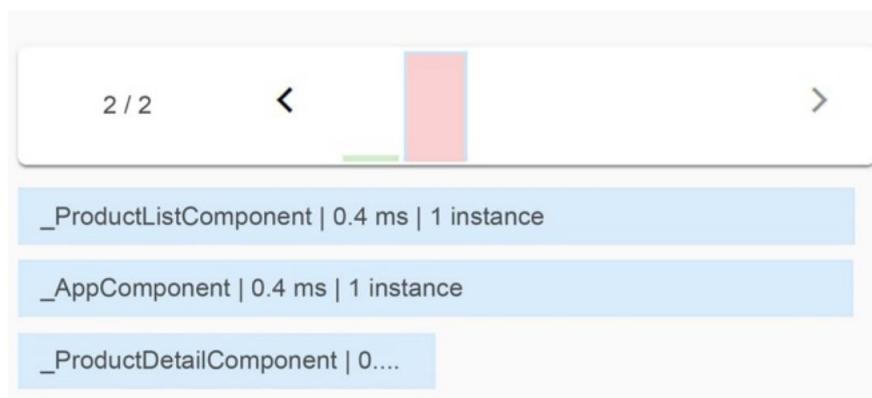


Figura 3.10: Gráfico de barras de detección de cambios

Angular ejecutó la detección de cambios en el componente de detalle del producto aunque no cambiamos sus propiedades.

7. Modifique el decorador @Component del archivo product-detail.component.ts configurando la propiedad changeDetection a ChangeDetectionStrategy.OnPush:

```
importar { ChangeDetectionStrategy, Component, entrada, salida } de '@ angular/core';

importar { Producto } de './producto';

@Component({
  selector: 'aplicación-producto-detalle',
  importaciones:
  [],
  templateUrl: './product-detail.component.html',
  styleUrls: ['./product-detail.component.css', detección de
  cambios: ChangeDetectionStrategy.OnPush })
```

8. Repita los pasos 4 a 6 y observe la salida de la segunda barra en la barra de detección de cambios. cuadro:

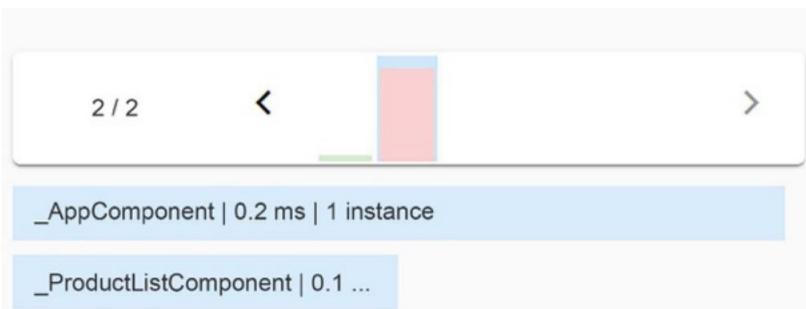


Figura 3.11: Gráfico de barras de detección de cambios

Esta vez no se ejecutó la detección de cambios para el componente de detalles del producto.

9. Haga clic en el producto Micrófono de la lista y observe la nueva barra en el gráfico de barras:

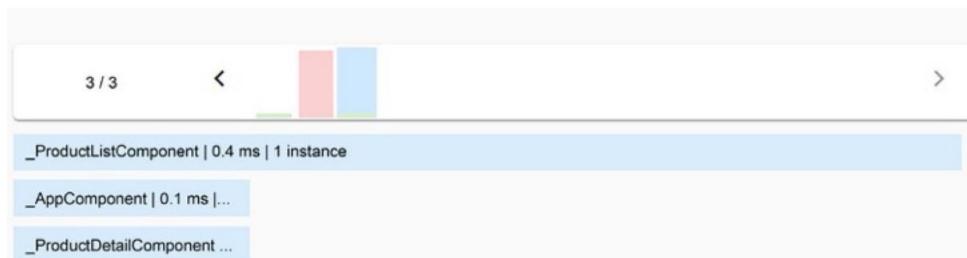


Figura 3.12: Gráfico de barras de detección de cambios

La detección de cambios se ejecutó esta vez porque modificamos la referencia de la propiedad de entrada del producto . Si simplemente hubiéramos modificado una propiedad con la estrategia de detección de cambios OnPush , el mecanismo de detección de cambios no se habría activado. Puede obtener más información sobre otros escenarios de detección de cambios en <https://angular.dev/best-practices/skipping-subtrees>.

La estrategia de detección de cambios es un mecanismo que nos permite modificar la forma en que nuestros componentes detectan cambios en sus datos, mejorando significativamente el rendimiento en aplicaciones a gran escala. Con esto concluye nuestro proceso de configuración de componentes, pero el framework Angular no se detiene ahí. Como aprenderemos en la siguiente sección, podemos conectarnos a momentos específicos en el ciclo de vida del componente.

Presentación del ciclo de vida de los componentes

Los eventos de ciclo de vida son ganchos que nos permiten acceder a etapas específicas del ciclo de vida de un componente y aplicar lógica personalizada. Su uso es opcional, pero pueden ser útiles si comprendes... Cómo utilizarlos.

Algunos ganchos se consideran buenas prácticas, mientras que otros ayudan a depurar y comprender qué sucede en una aplicación Angular. Un gancho tiene una interfaz que define un método que debemos implementar. El framework Angular garantiza la llamada al gancho, siempre que este método esté implementado en el componente.



Definir la interfaz en el componente no es obligatorio, pero se considera una buena práctica. Angular solo se preocupa por si hemos implementado el método.
O no.

Los ganchos del ciclo de vida más básicos de un componente Angular son:

- ngOnInit: Esto se llama cuando se inicializa un componente
- ngOnDestroy: Esto se llama cuando se destruye un componente
- ngOnChanges: Esto se llama cuando los valores de las propiedades de enlace de entrada en el componente cambiar
- ngAfterViewInit: Esto se llama cuando Angular inicializa la vista del componente actual. nente y sus componentes secundarios

Todos estos ganchos de ciclo de vida están disponibles en el paquete npm `@angular/core` de Angular estructura.



Una lista completa de todos los ganchos de ciclo de vida compatibles está disponible en la documentación oficial de Angular en <https://angular.dev/guide/components/lifecycle>.

Exploraremos cada uno mediante un ejemplo en las siguientes secciones. Empecemos con el gancho `ngOnInit`, que es el evento más básico del ciclo de vida de un componente.

Realizar la inicialización del componente

El gancho de ciclo de vida `ngOnInit` es un método que se llama durante la inicialización del componente. En esta etapa, todos los enlaces de entrada y las propiedades enlazadas a datos se han configurado correctamente, por lo que podemos usarlos con seguridad. Usar el constructor del componente para acceder a ellos puede ser tentador, pero sus valores no se habrían configurado en ese momento. Aprenderemos a usar el gancho de ciclo de vida `ngOnInit` con el siguiente ejemplo:

1. Abra el archivo `product-detail.component.ts` y agregue un constructor que registre el valor de la propiedad del producto en la consola del navegador:

```
constructor() {  
    console.log('Producto:', this.product());  
}
```

2. Importe la interfaz `OnInit` desde el paquete npm `@angular/core`:

```
importar { Componente, entrada, OnInit, salida } de '@angular/core';
```

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

La clase de exportación `ProductDetailComponent` implementa `OnInit`

4. Agregue el siguiente método en la clase `ProductDetailComponent` para registrar la misma información que en el paso 1:

```
ngOnInit(): vacío {  
    console.log('Producto:', this.product());  
}
```

5. Abra el archivo `product-list.component.ts` y establezca un valor inicial para el producto seleccionado. propiedad:

```
selectedProduct: Producto | undefined = this.products[0];
```

6. Ejecute la aplicación usando el comando ng serve e inspeccione la salida del navegador.
consola:

Product: undefined

Product: ► {id: 1, title: 'Keyboard'}

Figura 3.13: Salida de la consola

El primer mensaje del constructor contiene un valor indefinido , pero en el segundo mensaje, el valor de la propiedad del producto se muestra correctamente.

Los constructores deben estar relativamente vacíos y desprovistos de otra lógica que la de establecer variables iniciales. Agregar lógica empresarial dentro de un constructor hace que sea difícil simularlo en escenarios de prueba.

Otro buen uso del gancho ngOnInit es cuando necesitamos inicializar un componente con datos de una fuente externa, como un servicio Angular, como aprenderemos en el Capítulo 5, Administración de componentes complejos. Tareas con Servicios.

El marco Angular proporciona ganchos para todas las etapas del ciclo de vida del componente, desde la inicialización-
ción a la destrucción.

Limpieza de recursos de componentes

La interfaz que usamos para conectar el evento de destrucción de un componente es el gancho de ciclo de vida ngOnDestroy . Necesitamos importar la interfaz OnDestroy e implementar el método ngOnDestroy para empezar a usarla:

```
importar { Componente, entrada, OnDestroy, salida } de '@angular/core';
importar { Producto } de '../producto';

@Component({
  selector: 'aplicación-producto-detalle',
  importaciones: [],
  URL de plantilla: './producto-detalle.componente.html',
  styleUrl: './producto-detalle.componente.css'
})
clase de exportación ProductDetailComponent implementa OnDestroy {
```

```

producto = entrada<Producto>();
añadido = salida();

añadir a la cesta() {
    esto.agregado.emit();
}

ngOnDestroy(): vacío {

}
}

```

En el fragmento anterior, agregamos la interfaz OnDestroy e implementamos su ngOnDestroy método. Luego podemos agregar cualquier lógica personalizada en el método ngOnDestroy para ejecutar código cuando se destruye el componente.

Un componente se destruye cuando se elimina del árbol DOM de una página web debido a las siguientes razones:

- Uso del bloque @if de la sintaxis del flujo de control
- Navegar fuera de un componente usando el enrutador Angular, que aprenderemos sobre en el Capítulo 9, Navegación por aplicaciones con enrutamiento

Generalmente realizamos una limpieza de los recursos del componente dentro del método ngOnDestroy , como la siguiente:

- Restablecimiento de temporizadores e intervalos
- Cancelar la suscripción a flujos observables, que aprenderemos en el Capítulo 6, Reactivo Patrones en Angular

Un método alternativo al gancho de ciclo de vida ngOnDestroy es utilizar un servicio Angular integrado como como DestroyRef:

```

importar { Componente, DestroyRef, entrada, salida } de '@angular/core';
importar { Producto } de '../producto';

@Component({
    selector: 'aplicación-producto-detalle',
    importaciones: [],
    URL de plantilla: './producto-detalle.componente.html',
}

```

```
        styleUrl: './producto-detalle.componente.css'  
    })  
    clase de exportación ProductDetailComponent {  
        producto = entrada<Producto>();  
        añadido = salida();  
  
        constructor(destroyRef: DestroyRef) {  
            destruirRef.onDestroy(() => {  
                //  
            });  
        }  
  
        añadir a la cesta() {  
            esto.agregado.emit();  
        }  
    }  
}
```

Como aprenderemos en el Capítulo 5, "Gestión de Tareas Complejas con Servicios", usar un constructor es una forma de injectar servicios de Angular en otros artefactos de Angular. En este caso, el servicio `destroyRef` expone el método `onDestroy`, que acepta una función de devolución de llamada como parámetro. Esta función se llamará cuando se destruya el componente.

Ya aprendimos a pasar datos a un componente mediante un enlace de entrada. El framework Angular proporciona el gancho de ciclo de vida `ngOnChanges`, que podemos usar para inspeccionar cuándo cambia el valor de dicho enlace.

Detección de cambios en la vinculación de entrada

El gancho de ciclo de vida `ngOnChanges` se llama cuando Angular detecta que el valor de un enlace de datos de entrada ha cambiado. Lo usaremos en el componente de detalle del producto para comprender su comportamiento al seleccionar un producto diferente de la lista:

1. Importe las interfaces OnChanges y SimpleChanges en el componente product-detail.

archivo ts :

```
importar {  
    Componente,  
    aporte,  
    En caso de cambios,  
    producción,
```

```
Cambios simples
} de '@angular/core';
```

2. Modifique la definición de la clase ProductDetailComponent para que implemente la Interfaz OnChanges :

La clase de exportación ProductDetailComponent implementa OnChanges

3. Implemente el método ngOnChanges definido en la interfaz OnChanges . Este acepta un objeto del tipo SimpleChanges como parámetro, que contiene una clave para cada propiedad de entrada que cambia. Cada clave apunta a otro objeto con las propiedades currentValue .
- y previousValue , que denotan el valor nuevo y antiguo de la propiedad de entrada, respectivamente:

```
ngOnChanges(cambios: SimpleChanges): void {
  const producto = cambios['producto'];
  const oldValue = producto.previousValue; const
  newValue = producto.currentValue; console.log(
    'Valor antiguo', oldValue);
  console.log('Nuevo valor', newValue);
}
```

El fragmento anterior rastrea la propiedad de entrada del producto en busca de cambios y registra los cambios antiguos y nuevos valores en la ventana de la consola del navegador.

4. Para inspeccionar la aplicación, ejecute el comando ng serve , seleccione un producto de la lista y Observa la salida en la consola. Deberías obtener algo como esto:

```
Old value undefined
New value undefined
Angular is running in development mode.
Old value undefined
New value ▶ {id: 3, title: 'Web camera'}
```

Figura 3.14: Salida de la consola

En la imagen anterior, las dos primeras líneas indican que el valor del producto cambió de indefinido a indefinido. Este es el momento en que se inicializa el componente de detalle del producto, y la propiedad del producto aún no tiene valor. El evento del ciclo de vida OnChanges se activa al establecer el valor por primera vez y en todos los cambios posteriores que se realicen mediante el mecanismo de enlace.

5. Para eliminar los mensajes de registro innecesarios, podemos verificar si es la primera vez que se cambia la propiedad del producto utilizando el método isFirstChange :

```
ngOnChanges(cambios: SimpleChanges): void {
  const producto = cambios['producto']; si (!
    producto.isFirstChange()) {
    const oldValue = producto.previousValue; const
    newValue = producto.currentValue; console.log('
      Valor antiguo', oldValue);
    console.log('Nuevo valor', newValue);
  }
}
```

Si actualizamos el navegador, podremos ver el mensaje correcto en la ventana de la consola.

El gancho de ciclo de vida ngOnChanges es una excelente manera de detectar cuándo cambia el valor de una propiedad de entrada . Con la llegada de la API de Señales, disponemos de métodos mucho mejores para detectar y reaccionar ante estos cambios, como veremos en el Capítulo 7, Seguimiento del Estado de la Aplicación con Señales. Sin embargo, para versiones anteriores de Angular, el gancho sigue siendo la solución preferida.

El último evento del ciclo de vida de un componente Angular que exploraremos es el gancho ngAfterViewInit .

Acceso a componentes secundarios

El gancho del ciclo de vida ngAfterViewInit de un componente Angular se llama cuando:

- Se ha inicializado la plantilla HTML del componente.
- Se han inicializado las plantillas HTML de todos los componentes secundarios.

Podemos explorar cómo funciona el evento ngAfterViewInit usando la lista de productos y los detalles del producto. componentes:

1. Abra el archivo product-list.component.ts e importe AfterViewInit y ViewChild
artefactos del paquete npm @angular/core :

```
importar { AfterViewInit, Component, ViewChild } de '@angular/core';
```

2. Cree la siguiente propiedad en la clase ProductListComponent :

```
detalleDeProducto = ViewChild(ProductDetailComponent);
```

Ya aprendimos a consultar una clase de componente desde una plantilla HTML mediante variables de referencia locales. Como alternativa, podemos usar la función ViewChild para consultar un componente secundario desde la clase del componente principal.



En versiones anteriores de Angular, usamos el decorador `@ViewChild` para consultar componentes secundarios. Puedes obtener más información en <https://angular.dev/guide/components/queries>.

La función `viewChild` acepta el tipo de componente que queremos consultar como parámetro.

3. Modifique la definición de la clase `ProductListComponent` para que implemente la Interfaz `AfterViewInit` :

`La clase de exportación ProductListComponent implementa AfterViewInit`

4. La interfaz `AfterViewInit` implementa el método `ngAfterViewInit`, que podemos usar para acceder a la propiedad `productDetail` :

```
ngAfterViewInit(): vacío {  
  console.log(this.productDetail()!.product());  
}
```

Al consultar la propiedad `productDetail`, obtenemos una instancia de la clase `ProductDetailComponent`. Podemos acceder a cualquier miembro de su API pública, como el producto. propiedad.



Al ejecutar el código anterior se mostrará un valor indefinido para la propiedad del producto porque no establecemos un valor inicial cuando se usa el componente de detalle del producto. inicializado.

El evento de ciclo de vida `ngAfterViewInit` concluye nuestro recorrido por el ciclo de vida de los componentes de Angular. Los ganchos de ciclo de vida de componentes son una característica útil del framework y se usarán con frecuencia en el desarrollo de aplicaciones de Angular.

Resumen

En este capítulo, exploramos los componentes de Angular. Vimos su estructura y cómo crearlos, y explicamos cómo aislar la plantilla HTML de un componente en un archivo externo para facilitar su mantenimiento. Además, vimos cómo hacer lo mismo con cualquier hoja de estilos que quisieramos vincular al componente en caso de que no quisieramos agrupar los estilos del componente en línea. También aprendimos a usar la sintaxis de plantilla de Angular e interactuar con la plantilla del componente. De igual forma, vimos cómo los componentes se comunican bidireccionalmente mediante enlaces de propiedades y eventos.

Revisamos las opciones disponibles en Angular para crear API potentes para nuestros componentes, lo que nos permitió proporcionar altos niveles de interoperabilidad entre ellos, configurando sus propiedades mediante la asignación de valores estáticos o enlaces administrados. También vimos cómo un componente podía actuar como componente anfitrión de otro componente secundario, instanciando el elemento personalizado del primero en su plantilla y sentando las bases para árboles de componentes más grandes en nuestras aplicaciones . Los parámetros de salida nos brindan la capa de interactividad necesaria al convertir nuestros componentes en emisores de eventos para que puedan comunicarse adecuadamente con cualquier componente principal que eventualmente los aloje.

Las referencias de plantilla nos permitieron crear referencias en nuestros elementos personalizados, que podemos usar como accesos a sus propiedades y métodos desde la plantilla de forma declarativa. Un resumen de las funciones integradas para gestionar la encapsulación de vistas CSS en Angular nos brindó información adicional sobre cómo podemos beneficiarnos del alcance CSS de shadow DOM por componente. Finalmente, aprendimos lo importante que es la detección de cambios en una aplicación Angular y cómo podemos personalizarla para mejorar aún más su rendimiento.

También estudiamos el ciclo de vida de los componentes y aprendimos a ejecutar lógica personalizada mediante los ganchos de ciclo de vida integrados en Angular. Aún nos queda mucho por aprender sobre la gestión de plantillas en Angular, principalmente en lo que respecta a dos conceptos que usarás en tu experiencia con Angular: directivas y tuberías, que abordaremos en el siguiente capítulo.

Machine Translated by Google

4

Enriquecimiento de aplicaciones utilizando Tuberías y directivas

En el capítulo anterior, creamos varios componentes que renderizaban datos en pantalla mediante propiedades de entrada y salida.

En este capítulo, aprovecharemos este conocimiento para llevar nuestros componentes al siguiente nivel mediante tuberías y directivas de Angular. Las tuberías nos permiten procesar y transformar la información que enlazamos en nuestras plantillas. Las directivas habilitan funciones más ambiciosas , como manipular el DOM o modificar la apariencia y el comportamiento de los elementos HTML.

En este capítulo aprenderemos los siguientes conceptos:

- Manipulación de datos con tuberías
- Tuberías de construcción
- Directivas de construcción

Requisitos técnicos

Este capítulo contiene ejemplos de código que te guiarán a través de las tuberías y directivas de Angular. Puedes encontrar el código fuente relacionado en la carpeta ch04 del siguiente repositorio de GitHub:

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

Manipulación de datos con tuberías

Las tuberías nos permiten transformar el resultado de nuestras expresiones a nivel de vista. Reciben datos como entrada, los transforman al formato deseado y muestran el resultado en la plantilla.

La sintaxis de una tubería consiste en el nombre de la tubería después de la expresión que queremos transformar, separados por un símbolo de tubería (|):

```
expresión | tubería
```

Todos los parámetros se agregan después del nombre de la tubería y se separan por dos puntos:

```
expresión | tubería:parámetro
```

Las tuberías se pueden usar con interpolación y vinculación de propiedades en plantillas angulares y se pueden encadenar entre sí.

Angular tiene una amplia gama de tipos de tuberías integrados:

- mayúsculas/minúsculas: Esto transforma una cadena en letras mayúsculas o minúsculas.
- porcentaje: Esto formatea un número como porcentaje.
- Fecha: Formatea una fecha o una cadena con un formato de fecha específico. El uso predeterminado de la barra vertical muestra la fecha según la configuración local del equipo del usuario. Sin embargo, podemos pasar formatos adicionales que Angular ya incluye como parámetros.
- Moneda: Formatea un número como moneda local. Podemos anular la configuración local y cambiar el símbolo de la moneda, pasando el código de la moneda como parámetro a la canalización.
- json: Toma un objeto como entrada y lo genera en formato JSON, reemplazando las comillas simples por comillas dobles. El uso principal de la tubería json es la depuración. Es una excelente manera de ver el contenido de un objeto complejo e imprimirla correctamente en pantalla.
- keyvalue:

Convierte un objeto en una colección de pares clave-valor, donde la clave de cada elemento representa la propiedad del objeto y el valor es su valor real.

- Segmentación: Resta un subconjunto (segmento) de una colección o cadena. Acepta como parámetros un índice inicial, donde comenzará a segmentar los datos de entrada, y, opcionalmente, un índice final. Cuando se especifica el índice final, el elemento en ese índice no se incluye en la matriz resultante. Si se omite, se recurre al último índice de los datos.



La tubería de segmentación transforma datos inmutables. La lista transformada siempre es una copia de los datos originales, incluso si devuelve todos los elementos.

- asíncrono: Se utiliza cuando gestionamos datos que nuestra clase de componente maneja de forma asíncrona y necesitamos asegurarnos de que nuestras vistas reflejen los cambios rápidamente. Aprenderemos más sobre esta tubería más adelante en el Capítulo 8, Comunicación con Servicios de Datos a través de HTTP, donde la usaremos para obtener y mostrar datos de forma asíncrona.



Necesitará el código fuente de la aplicación Angular que creamos en el Capítulo 3, Estructuración de interfaces de usuario con componentes, para seguir con el resto del capítulo.

Cubriremos las tuberías de minúsculas, moneda y valor clave con más detalle, pero lo alejaremos a explorar el resto en la referencia de API en <https://angular.dev/api>:

1. Abra el archivo product-detail.component.ts e importe la clase CommonModule :

```
importar { CommonModule } desde '@angular/common';
importar { Componente, entrada, salida } de '@angular/core';
importar { Producto } de './producto';

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

La clase CommonModule exporta las tuberías integradas de Angular. Un componente de Angular debe importar CommonModule antes de usar las tuberías integradas en la plantilla del componente.

2. Abra el archivo product.ts y agregue los siguientes campos a la interfaz Producto que describen propiedades adicionales para un producto:

```
Interfaz de exportación Producto {
  id: numero;
  título: cadena;
  precio: numero;
  categorías: Registro<número, cadena>;
}
```

La propiedad categorías es un objeto donde la clave representa el ID de la categoría y el valor representa la descripción de la categoría.

3. Abra el archivo product-list.component.ts y modifique la matriz de productos para establecer valores Para las nuevas propiedades:

```
productos: Producto[] = [
  {
```

```

    id: 1,
    título: 'Teclado',
    precio: 100,
    categorías: {
        1: 'Computación',
        2: 'Periféricos'
    },
    {

        identificación: 2,
        Título: 'Micrófono', precio:
        35,
        categorías: { 3: 'Multimedia' } },

    {

        identificación: 3,
        Título: 'Cámara web',
        precio: 79,
        categorías: {
            1: 'Computación',
            3: 'Multimedia'
        }
    },
    {

        identificación: 4,
        Título: 'Tableta',
        Precio: 500,
        Categorías: { 4: 'Entretenimiento' }

    }];

```

4. Abra el archivo product-detail.component.html y agregue un elemento de párrafo para mostrar el precio del producto seleccionado en euros:

```

@if (producto()) {
    <p>Usted seleccionó:
    <strong>{{producto()!.título}}</strong> </p>

```

```
<p>{{product()!.price | currency:'EUR'}}</p> <button  
(click)="addToCart()">Añadir al carrito</button>  
>
```

5. Ejecute ng serve para iniciar la aplicación y seleccione el Micrófono de la lista de productos:

You selected: **Microphone**

€35.00

Add to cart

Figura 4.1: Detalles del producto

En la imagen anterior, el precio del producto se muestra en formato de moneda.

6. Agregue el siguiente fragmento debajo del precio del producto para mostrar las categorías del producto:

```
<div class="pill-group"> @for  
(cat.de producto ()!.categorías | valorclave; pista cat.clave) { <p class="pill">{{cat.valor  
| minúsculas}}</p>  
}  
</div>
```

En el fragmento anterior, usamos el bloque @for para iterar sobre la propiedad categorías de la variable producto . Esta propiedad no es iterable porque es un objeto simple , por lo que usamos la canalización clave-valor para convertirla en una matriz que contiene las propiedades clave y valor . La propiedad clave representa el ID de la categoría, un identificador único que podemos usar con la variable pista . La propiedad valor almacena la descripción de la categoría.

Además, usamos la barra vertical en minúsculas para convertir la descripción de la categoría a minúsculas. texto del caso.

7. Agregue los siguientes estilos CSS al archivo product-detail.component.css :

```
.grupo de píldoras {  
    pantalla: flexible;  
    flexión-dirección: fila;  
    alinear-elementos: inicio;  
    flex-wrap: envolver;  
    brecha: 1,25rem;  
}
```

```
.píldora {
    pantalla: flex;
    alinear-elementos: centro;
    --pill-accent: var(--gray-900); fondo:
    mezcla de colores(en srgb, var(--pill-accent) 5%,
    transparente);
    color: var(--pill-accent); relleno-en-
    línea: 0.75rem;
    bloque de relleno: 0,375rem;
    radio del borde: 2,75rem;
    borde: 0;
    transición: fondo 0,3 s facilidad;
    familia de fuentes: var(--inter-font); tamaño
    de fuente: 0.875rem;
    estilo de fuente: normal;
    peso de fuente: 500;
    altura de línea: 1,4rem;
    espaciado entre letras: -0,00875rem;
    decoración de texto: ninguna;
}
```

8. Mientras ejecuta la aplicación, seleccione el producto Cámara web de la lista:

You selected: **Web camera**

€79.00

computing

multimedia

Add to cart

Figura 4.2: Detalles del producto con categorías

Como alternativa a utilizar CommonModule, podríamos haber importado cada clase de tubería por separado desde el paquete npm @angular/common :

```
importar { CurrencyPipe, KeyValuePipe, LowerCasePipe } de '@angular/ common';
```

```
importar { Componente, entrada, salida } de '@angular/core'; importar
{ Producto } de './product';

@Component({
  selector: 'app-product-detail', importa:
  [KeyValuePipe, CurrencyPipe, LowerCasePipe],
  URL de plantilla: './product-detail.component.html', URL de
  estilo: './product-detail.component.css'
})
```

En el archivo final product-detail.component.html , usamos el fragmento `product()!` varias veces para leer el valor de la propiedad del producto . Como alternativa, podríamos crear un alias con la sintaxis @let , como se indica a continuación:

```
@let selectedProduct = producto();
```

La palabra clave @let es similar a la palabra clave let en JavaScript y se utiliza para declarar variables disponibles únicamente en la plantilla del componente. En el fragmento anterior, declaramos la variable selectedProduct , que puede usarse en el resto del código HTML de la siguiente manera:

```
@if (selectedProduct)
  { <p>Seleccionaste:
    <strong>{{productoSeleccionado.título}}</strong> </p>
    <p>{{productoSeleccionado.precio | moneda:'EUR'}}</p> <div
    class="pill-group">
      @for (cat de selectedProduct.categories | keyvalue; track cat.key) {
        <p class="pill">{{cat.value | minúsculas}}</p>
      }
    </div>
    Añadir al carrito
  }
```

La palabra clave @let nos ayuda en los casos en los que queremos utilizar expresiones complejas en plantillas. como:

- Operadores ternarios
- Propiedades de objetos anidados
- La tubería asíncrona

Las tuberías integradas son suficientes para la mayoría de los casos de uso, pero en otros casos debemos aplicar transformaciones complejas a nuestros datos. El framework Angular proporciona un mecanismo para crear tuberías personalizadas , como veremos en la siguiente sección.

Tuberías de construcción

Ya hemos visto qué son las tuberías y cuál es su propósito en el ecosistema Angular. A continuación, profundizaremos en cómo crear una tubería para proporcionar transformaciones personalizadas a los enlaces de datos. En la siguiente sección, crearemos una tubería que ordene nuestra lista de productos por título.

Ordenar datos mediante tuberías

Para crear una nueva tubería, utilizamos el comando ng generate de la CLI de Angular, pasando su nombre como un parámetro:

```
ng genera una clasificación de tuberías
```

El comando anterior generará todos los archivos necesarios de la tubería de ordenación dentro de la carpeta donde ejecutamos el comando ng generate . La clase TypeScript de la tubería se define en sort.pipe.

archivo ts :

```
importar { Pipe, PipeTransform } desde '@angular/core';

@Tubo({
  nombre: 'sort'
})
La clase de exportación SortPipe implementa PipeTransform {

  transformar(valor: desconocido, ...args: desconocido[]): desconocido {
    devuelve nulo;
  }
}
```

@Pipe es un decorador angular que define el nombre de la tubería angular.

La clase TypeScript de una tubería implementa el método de transformación de la interfaz PipeTransform y acepta dos parámetros:

- valor: Los datos de entrada que queremos transformar

- args: Una lista opcional de argumentos que podemos proporcionar al método de transformación, cada uno separados por dos puntos

La CLI de Angular nos ayudó a crear un método de transformación vacío . Ahora necesitamos modificarlo para satisfacer nuestras necesidades de negocio. La tubería operará sobre una lista de objetos de producto , por lo que debemos realizar los ajustes necesarios en los tipos proporcionados:

1. Agregue la siguiente declaración para importar la interfaz del producto :

```
importar { Producto } desde './producto';
```

2. Cambie el tipo del parámetro de valor a Producto[] ya que queremos ordenar una lista de Objetos de producto .

3. Cambie el tipo de método a Producto[] ya que la lista ordenada solo contendrá Producto objetos y modificarlo para que devuelva una matriz vacía de forma predeterminada.

El archivo sort.pipe.ts resultante ahora debería verse como el siguiente:

```
importar { Pipe, PipeTransform } desde '@angular/core';
importar { Producto } desde './producto';

@Tubo({
  nombre: 'sort'
})
La clase de exportación SortPipe implementa PipeTransform {

  transformar(valor: Producto[], ...args: desconocido[]): Producto[] {
    devolver [];
  }
}
```

Ahora estamos listos para implementar el algoritmo de ordenamiento de nuestro método. Utilizaremos el método de ordenamiento nativo , que ordena los elementos alfabéticamente de forma predeterminada. Proporcionaremos una función de comparación personalizada para el método de ordenamiento que anula la funcionalidad predeterminada y ejecuta la lógica de ordenamiento deseada:

```
transformar(valor: Producto[], ...args: desconocido[]): Producto[] {
  si (valor) {
    valor de retorno.sort((a: Producto, b: Producto) => {
      si (a.título < b.título) {
```

```

    devuelve -1;
} de lo contrario si (b.título < a.título) {
    devuelve 1;
}
devuelve
0; });

} devolver [];
}

```

Cabe destacar que el método de transformación verifica si existen datos de entrada antes de proceder al proceso de ordenación. De lo contrario, devuelve una matriz vacía. Esto mitiga los casos en los que la colección se configura de forma asíncrona o el componente que consume la tubería no lo hace. establecer la colección en absoluto.



Para obtener más información sobre el método de clasificación , consulte https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Array/sort.

¡Listo! Hemos creado nuestra primera tubería correctamente. Necesitamos llamarla desde nuestra plantilla de componente para verla en acción:

1. Abra el archivo product-list.component.ts e importe la clase SortPipe :

```

importar { Componente } de '@angular/core'; importar
{ Producto } de './product'; importar
{ ProductDetailComponent } de '../product-detail/product-detail.component'; importar
{ SortPipe } de './
sort.pipe';

@Component({
  selector: 'app-product-list', importa:
  [ProductDetailComponent, SortPipe],
  URL de plantilla: './lista-de-productos.componente.html', URL
  de estilo: './lista-de-productos.componente.css' })

```

2. Abra el archivo product-list.component.html y agregue la tubería en el bloque @for :

```
<ul class="grupo de píldoras">
  @for (producto de productos | ordenar; rastrear producto.id) {
    <li class="pill" (clic)="productoSeleccionado = producto">
      @switch (producto.título) {
        @case ('Teclado') { @case ☖️ }
        ('Micrófono') { @default { 🎤 } }
      }
    {{producto.título}}
    </li>
  } @vacío {
    <p>¡No se encontraron productos!</p>
  }
</ul>
```

3. Si ejecutamos la aplicación usando el comando ng serve , notaremos que la lista de productos ahora está ordenada alfabéticamente por título:

Products (4)

 Keyboard

 Microphone

 Tablet

 Web camera

Figura 4.3: Lista de productos ordenada alfabéticamente por título

El canal de ordenación permite ordenar los datos de productos únicamente por título. En la siguiente sección, aprenderemos a configurarlo para que también pueda ordenar por otras propiedades del producto.

Pasando parámetros a las tuberías

Como aprendimos en la sección "Manipulación de datos con tuberías", podemos pasar parámetros adicionales a una tubería mediante dos puntos. Usamos el parámetro args en el método de transformación de una tubería para obtener el valor de cada parámetro separado por dos puntos. Aprendimos que la CLI de Angular crea los args. parámetro por defecto y utiliza el operador de propagación para expandir sus valores en el método:

```
transformar(valor: Producto[], ...args: desconocido[]): Producto[] {  
    si (valor) {  
        valor de retorno.sort ((a: Producto, b: Producto) => {  
            si (a.título < b.título) {  
                devuelve -1;  
            } de lo contrario si (b.título < a.título) {  
                devuelve 1;  
            }  
            devuelve 0;  
        });  
  
        } devolver [];  
    }  
}
```

Actualmente, el método de transformación solo funciona con la propiedad de título de un producto. Podríamos aprovechar el parámetro args para hacerlo dinámico y permitir que el consumidor de la tubería defina la propiedad con la que desea ordenar los datos, como el precio del producto:

1. Elimine el operador de propagación del parámetro args porque pasaremos una sola propiedad.

propiedad de un producto cada vez y cambiar su tipo, de la siguiente manera:

```
transformar(valor: Producto[], argumentos: clave de Producto): Producto[] {  
    si (valor) {  
        valor de retorno.sort ((a: Producto, b: Producto) => {  
            si (a.título < b.título) {  
                devuelve -1;  
            } de lo contrario si (b.título < a.título) {  
                devuelve 1;  
            }  
            devuelve 0;  
        });  
    }  
}
```

```
    devolver [];
}
```

En el método anterior, utilizamos el operador de tipo keyof de TypeScript para definir que el parámetro args puede ser cualquier propiedad de un objeto Producto .

2. Reemplace la propiedad del título con el parámetro args dentro de la declaración if :

```
si (valor) {
  valor de retorno.sort ((a: Producto, b: Producto) => {
    si (a[argumentos] < b[argumentos]) {
      devuelve -1;
    } de lo contrario si (b[args] < a[args]) {
      devuelve 1;
    }
    devuelve 0;
  });
}
```

Tenga en cuenta que en el fragmento anterior, accedemos a los objetos a y b utilizando la sintaxis de corchetes en lugar de la sintaxis de punto como antes.

3. Modifique el parámetro args en la firma del método para que utilice la propiedad de título de manera predeterminada, si el consumidor no pasa ningún parámetro en la tubería:

```
transformar(valor: Producto[], argumentos: clave de Producto = 'título')
```

El comportamiento anterior garantiza que el componente de lista de productos funcionará sin ningún cambio en el uso de la tubería.

4. Ejecute el comando ng serve y verifique que la lista de productos esté ordenada inicialmente por título.

5. Abra el archivo product-list.component.html y pase la propiedad de precio como una tubería parámetro:

```
@for (producto de productos | sort:'precio'; seguimiento producto.id) {
  <li class="pill" (clic)="productoSeleccionado = producto">
    @switch (producto.título) {
      @case ('Teclado') { @case ⌨ }
      ('Micrófono') { @default { ⌛ } }
    }
}
```

```

    {{producto.título}}
</li>
}

```

6. Guarde el archivo y espere a que la aplicación se recargue. Debería ver la lista de productos.

Ahora está ordenado por precio:

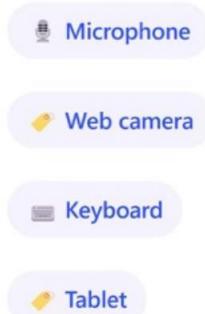


Figura 4.4: Lista de productos ordenada por precio

El decorador @Pipe contiene otra propiedad importante que podemos configurar y que está directamente relacionada con la forma en que las tuberías reaccionan en el mecanismo de detección de cambios del marco Angular.

Detección de cambios con tuberías

Existen dos categorías de tuberías: puras e impuras. Todas las tuberías se consideran puras por defecto, a menos que establezcamos la propiedad "pure" explícitamente como "false" en el decorador @Pipe :

```

@Tubo({
  nombre: 'ordenar',
  puro: falso
})

```

Angular ejecuta canalizaciones puras cuando se produce un cambio en la referencia de la variable de entrada. Por ejemplo, si se asigna un nuevo valor al array de productos de la clase ProductListComponent , la canalización reflejará correctamente dicho cambio. Sin embargo, si añadimos un nuevo producto al array mediante el método nativo Array.push , la canalización no se activará porque la referencia al objeto del array no cambia.

Otro ejemplo es cuando creamos una tubería pura que opera sobre un solo objeto. De igual forma, si la referencia del valor cambia, la tubería se ejecuta correctamente. Si cambia una propiedad del objeto, la tubería no puede detectar el cambio.

Sin embargo, tenga cuidado: las tuberías impuras llaman al método de transformación cada vez que se activa el ciclo de detección de cambios. Por lo tanto, esto podría perjudicar el rendimiento. Como alternativa, puede dejar la propiedad pura sin definir e intentar almacenar en caché el valor o trabajar con reductores y datos inmutables para solucionar esto de una mejor manera, como se muestra a continuación:

```
este.productos= [
  ...estos.productos,
  {
    identificación: 5,
    Título: 'Auriculares',
    precio: 55,
    categorías: { 3: 'Multimedia' }
  }
];
```

En el fragmento anterior, utilizamos la sintaxis del parámetro spread para crear una nueva referencia de la matriz de productos agregando un nuevo elemento a la referencia de la matriz existente.

Como alternativa a una tubería pura, podemos utilizar una señal calculada, que es más efectiva y ergonómica por las siguientes razones:

- Podemos acceder al valor de la señal en la clase del componente, a diferencia de las tuberías, donde Sus valores sólo se pueden leer en la plantilla
- Una señal calculada es una función simple y sencilla, por lo que no necesitamos usar una clase TypeScript como en tuberías

Aprenderemos más sobre las señales en el Capítulo 7, Seguimiento del estado de la aplicación con señales.

La creación de tuberías personalizadas nos permite transformar nuestros datos de una manera particular según nuestras necesidades. Debemos crear directivas personalizadas si también queremos transformar elementos de la plantilla.

Directivas de construcción

Las directivas de Angular son atributos HTML que amplían el comportamiento o la apariencia de un elemento HTML estándar. Al aplicar una directiva a un elemento HTML o incluso a un componente de Angular, podemos añadir un comportamiento personalizado o modificar su apariencia. Existen tres tipos de directivas:

- Componentes: Los componentes son directivas que contienen una plantilla HTML asociada.
- Directivas estructurales: agregan o eliminan elementos del DOM.

Directivas de atributo: Modifican la apariencia de un elemento DOM o definen un comportamiento personalizado. En el capítulo anterior, vimos las directivas de atributo en las vinculaciones de clases y estilos.

Si una directiva tiene una plantilla adjunta, se convierte en un componente. En otras palabras, los componentes son directivas de Angular con una vista. Esta regla es útil para decidir si crear un componente o una directiva según tus necesidades. Si necesitas una plantilla, crea un componente; de lo contrario, conviértalo en una directiva.

Las directivas personalizadas nos permiten asignar comportamientos avanzados a los elementos del DOM o modificar su apariencia. En las siguientes secciones, exploraremos cómo crear directivas de atributos.

Visualización de datos dinámicos

Las directivas de atributos se usan comúnmente para modificar la apariencia de un elemento HTML. Probablemente todos nos hayamos encontrado en la situación de querer añadir información con derechos de autor a nuestras aplicaciones. Idealmente, queremos usar esta información en varias partes de nuestra aplicación, como un panel de control o una página de contacto. El contenido de la información también debería ser dinámico. El año o el rango de años (depende de cómo se quiera usar) debería actualizarse dinámicamente según la fecha actual. Probablemente nuestra primera intención sea crear un componente, pero ¿qué tal si lo convertimos en una directiva? De esta forma, podríamos asociar la directiva a cualquier elemento que queramos sin tener que usar una plantilla específica. ¡Comencemos!

Utilizaremos el comando ng generate del CLI de Angular, pasando el nombre de la directiva como un parámetro:

```
ng generar directiva de derechos de autor
```

El comando anterior generará todos los archivos necesarios de la directiva de derechos de autor dentro de la carpeta donde ejecutamos el comando ng generate . La clase TypeScript de la directiva se define en el archivo copyright.directive.ts :

```
importar { Directiva } desde '@angular/core';

@Directiva({
  selector: '[appCopyright]'
})
clase de exportación CopyrightDirective {

  constructor() { }

}
```

@Directive es un decorador de Angular que define las propiedades de la directiva de Angular. Configura una clase TypeScript como directiva de Angular mediante la propiedad selector . Es un selector CSS que indica a Angular que cargue la directiva en la ubicación que encuentra el atributo correspondiente en una plantilla HTML. La CLI de Angular añade el prefijo "app" por defecto, pero se puede personalizar con la opción --prefix al crear el proyecto de Angular.



Cuando utilizamos el selector en una plantilla HTML, no agregamos los corchetes.

Utilicemos la directiva recién creada para agregar información de derechos de autor a nuestra aplicación:

1. Abra el archivo designs.css y agregue los siguientes estilos CSS:

```
.derechos de autor {  
    familia de fuentes: "Inter", -apple-system, BlinkMacSystemFont, "Segoe  
    interfaz de usuario", roboto,  
    Helvética, Arial, sans-serif, "Apple Color Emoji", "Segoe UI"  
    Emojis;  
    "Símbolo de interfaz de usuario de Segoe";  
    ancho: 100%;  
    min-altura: 100%;  
    pantalla: flexible;  
    justificar-contenido: centro;  
    alinear-elementos: centro;  
    relleno: 1rem;  
    tamaño de caja: heredar;  
    posición: relativa;  
}
```

En el fragmento anterior, agregamos los estilos CSS para nuestra directiva de derechos de autor en la hoja de estilos CSS global. Las directivas no tienen un archivo CSS adjunto que podamos usar, como como componentes.

2. Abra el archivo copyright.directive.ts e importe la clase ElementRef desde el

Paquete npm de @angular/core :

```
importar { Directiva, ElementRef } de '@angular/core';
```

3. Modifique el constructor de la directiva de la siguiente manera:

```
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
Reservado`;
}
```

En el fragmento anterior, usamos la clase ElementRef para acceder y manipular el elemento HTML subyacente asociado a la directiva. La propiedad nativeElement contiene el elemento HTML nativo. También añadimos la clase copyright mediante el método add de la propiedad classList . Finalmente, cambiamos el texto del elemento modificando el

Propiedad.textContent .



ElementRef es un servicio integrado de Angular. Para usar un servicio en un componente o directiva, debemos inyectarlo en el constructor, como veremos en el Capítulo 5, "Gestión de tareas complejas con servicios".

4. Abra el archivo app.component.ts e importe la clase CopyrightDirective :

```
importar { Componente } desde '@angular/core';
importar { RouterOutlet } desde '@angular/router';
importar { ProductListComponent } desde './product-list/product-list.
componente';
importar { CopyrightDirective } desde './copyright.directive';
```

```
@Componente({
  selector: 'app-root',
  importaciones: [
    Salida de enrutador,
    Componente de lista de productos,
    Directiva sobre derechos de autor
```

```
    ],
    URL de plantilla: './app.component.html',
    estiloUrl: './app.component.css' })
```

5. Abra el archivo app.component.html y agregue un elemento <footer> para mostrar los derechos de autor información:

```
<main class="principal">
  <div class="contenido">
    <lista-de-productos-de-aplicaciones></lista-de-productos-de-aplicaciones>
  </div>
</principal>
<footer appCopyright></footer>
<enrutador-de-salida />
```

6. Ejecute la aplicación utilizando el comando ng serve y observe la salida de la aplicación:

Products (4)

Keyboard

Microphone

Tablet

Web camera

Copyright ©2024 All Rights Reserved

Figura 4.5: Salida de la aplicación

Al crear directivas, es importante considerar la funcionalidad reutilizable que no necesariamente se relaciona con una característica específica. El tema que analizamos fue información con derechos de autor, pero podríamos crear otras funcionalidades, como descripciones emergentes y funciones de desplazamiento plegable o infinito, con relativa facilidad. En la siguiente sección, crearemos otra directiva de atributo que explora las opciones disponibles con más detalle.

Vinculación de propiedades y respuesta a eventos

Las directivas de atributos también se relacionan con el comportamiento de un elemento HTML. Pueden ampliar la funcionalidad del elemento y añadir nuevas características. El framework Angular proporciona dos decoradores útiles que podemos usar en nuestras directivas para mejorar la funcionalidad de un elemento HTML:

- `@HostBinding`: Esto vincula un valor a la propiedad del elemento host nativo.
- `@HostListener`: Esto se vincula a un evento del elemento host nativo.



El elemento host nativo es el elemento donde nuestra directiva toma acción.

El elemento HTML nativo `<input>` admite diferentes tipos de entrada, incluyendo texto simple, botones de opción y valores numéricos. Al usar este último, la entrada añade dos flechas en línea, arriba y abajo, para controlar su valor. Esta característica del elemento de entrada hace que parezca incompleto. Si escribimos un carácter no numérico, la entrada lo representa igualmente.

Crearemos una directiva de atributo que rechace valores no numéricos ingresados por el teclado:

1. Ejecute el siguiente comando CLI de Angular para crear una nueva directiva denominada numérica:

```
ng genera directiva numérica
```

2. Abra el archivo `numeric.directive.ts` e importe los dos decoradores que vamos a

Para utilizar:

```
importar { Directiva, HostBinding, HostListener } desde '@angular/
centro';
```

3. Defina una propiedad `currentClass` usando el decorador `@HostBinding` que estará vinculada a la propiedad de clase del elemento `<input>`:

```
@HostBinding('class') claseActual = '';
```

4. Defina un método onKeyPress usando el decorador @HostListener que estará vinculado a el evento nativo de pulsación de tecla del elemento <input> :

```
@HostListener('keypress', ['$event']) onKeyPress(evento: KeyboardEvent)
{
  const charCode = evento.key.charCodeAt(0);
  si (código_carácter > 31 && (código_carácter < 48 || código_carácter > 57)) {
    este.currentClass = 'inválido';
    evento.preventDefault();
  } demás {
    este.currentClass = 'válido';
  }
}
```

5. Abra el archivo style.css y agregue los siguientes estilos CSS que se aplicarán cuando un

El componente utiliza la directiva:

```
entrada.válida {
  borde: verde sólido;
}
entrada.inválida {
  borde: rojo sólido;
}
```

El método onKeyPress contiene la lógica de cómo funciona nuestra directiva bajo el capó.

Cuando el usuario presiona una tecla dentro de un elemento <input> , Angular sabe que debe llamar a onKeyPress porque lo hemos registrado con el decorador @HostListener . El decorador @HostListener El decorador acepta el nombre del evento y una lista de argumentos como parámetros. En nuestro caso, pasamos el nombre del evento de pulsación de tecla y el argumento \$event , respectivamente. \$event es el objeto actual que activó el evento, que es del tipo KeyboardEvent y contiene las pulsaciones de teclas realizadas por el usuario.

Cada vez que el usuario pulsa una tecla, la extraemos del objeto \$event , la convertimos a un carácter Unicode mediante el método charCodeAt y la comparamos con un código no numérico. Si el carácter no es numérico, llamamos al método preventDefault del objeto \$event para cancelar la acción del usuario y revertir el elemento <input> a su estado anterior. Al mismo tiempo, configuramos la clase correspondiente como válida si la tecla es numérica e inválida si no lo es.

Podemos aplicar la directiva en una etiqueta <input> de la siguiente manera:

```
<entrada appNumeric />
```



Veremos un uso real de la directiva en el Capítulo 10, Recopilación de datos de usuario con formularios. Mientras tanto, si desea probarlo usted mismo, recuerde importar la clase NumericDirective en su componente antes de usarla.

Resumen

Llegados a este punto, es justo decir que ya conoces casi todos los artefactos de Angular para crear componentes, que son, de hecho, el motor de todas las aplicaciones Angular. En los próximos capítulos, veremos cómo podemos diseñar mejor la arquitectura de nuestra aplicación, gestionar la inyección de dependencias en nuestro árbol de componentes, consumir servicios de datos y aprovechar el nuevo enrutador Angular para mostrar y ocultar componentes cuando sea necesario.

Ahora, prepárese para asumir nuevos desafíos: en el próximo capítulo, descubriremos cómo utilizar los servicios de datos para administrar tareas complejas en nuestros componentes.

5

Gestión de tareas complejas con Servicios

Hemos llegado a un punto en nuestro camino donde podemos desarrollar con éxito aplicaciones más complejas anidando componentes dentro de otros en una especie de árbol de componentes. Sin embargo, agrupar toda nuestra lógica de negocio en un solo componente no es la solución. Nuestra aplicación podría volverse inmantenible muy pronto, a medida que se desarrolle.

En este capítulo, investigaremos las ventajas que el mecanismo de gestión de dependencias de Angular puede aportar para superar estos problemas. Aprenderemos a usar el mecanismo de inyección de dependencias (ID) de Angular para declarar y consumir nuestras dependencias en toda la aplicación con el mínimo esfuerzo y resultados óptimos. Al finalizar este capítulo, podrá crear una aplicación Angular correctamente estructurada para implementar el patrón de Separación de Intereses (SoC) mediante servicios.

Cubriremos los siguientes conceptos relacionados con los servicios Angular:

- Introducción a Angular DI
- Creando nuestro primer servicio Angular
- Proporcionar dependencias en toda la aplicación
- Inyección de servicios en el árbol de componentes
- Anulación de proveedores en la jerarquía de inyectores

Requisitos técnicos

El capítulo contiene varios ejemplos de código para guiarlo a través del concepto de servicios Angular.

Puede encontrar el código fuente relacionado en la carpeta ch05 del siguiente repositorio de GitHub:

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

Presentación de Angular DI

DI es un patrón de diseño de aplicaciones que también encontramos en otros lenguajes, como C# y Java. A medida que nuestras aplicaciones crecen y evolucionan, cada entidad de código requerirá internamente instancias de otros objetos, mejor conocidos como dependencias. Pasar estas dependencias a la entidad de código del consumidor se conoce como inyección y también implica la participación de otra entidad de código llamada inyector. Un inyector es responsable de instanciar y arrancar las dependencias necesarias para que estén listas para su uso al inyectarse en un consumidor. El consumidor desconoce cómo instanciar sus dependencias y solo conoce la interfaz que implementa para usarlas.

Angular incluye un mecanismo de DI de primera categoría para exponer las dependencias requeridas a cualquier artefacto de una aplicación Angular. Antes de profundizar en este tema, veamos el problema que la DI en Angular intenta resolver.

En el Capítulo 3, "Estructura de Interfaces de Usuario con Componentes", aprendimos a mostrar una lista de objetos mediante el bloque `@for`. Usamos una lista estática de objetos de Producto declarados en el archivo `product-list.component.ts`, como se muestra a continuación:

```
productos: Producto[] = [
  {
    identificación: 1,
    Título: 'Teclado',
    precio: 100,
    categorías: {
      1: 'Computación',
      2: 'Periféricos'
    }
  },
  {
    identificación: 2,
    Título: 'Micrófono',
    precio: 35,
    categorías: { 3: 'Multimedia' }
  }
]
```

```
    },
    {

        identificación: 3,
        Título: 'Cámara web',
        precio: 79,
        categorías: {
            1: 'Computación',
            3: 'Multimedia'
        }
    },
    {

        identificación: 4,
        Título: 'Tableta',
        precio: 500,
        categorías: { 4: 'Entretenimiento' }
    }
];
};
```

Este enfoque anterior tiene dos inconvenientes principales:

- En aplicaciones del mundo real, rara vez trabajamos con datos estáticos. Generalmente provienen de un back-API final o alguna otra fuente externa.

La lista de productos está estrechamente vinculada al componente. Los componentes Angular son responsables de la lógica de presentación y no deberían preocuparse por cómo obtener los datos. Solo necesitan mostrarlos en la plantilla HTML. Por lo tanto, deberían delegar la lógica de negocio a servicios para manejar tales tareas.

En la siguiente sección, aprenderemos cómo evitar estos obstáculos utilizando los servicios de Angular.



Necesitará el código fuente de la aplicación Angular que creamos en el Capítulo 4, Enriquecimiento de aplicaciones mediante tuberías y directivas, para continuar con el resto del capítulo.

Crearemos un servicio Angular que devolverá la lista de productos. De esta forma, delegaremos eficazmente las tareas de lógica de negocio fuera del componente. Recuerda: el componente solo debe ocuparse de la lógica de presentación.

Creando nuestro primer servicio Angular

Para crear un nuevo servicio Angular, utilizamos el comando ng generate de la CLI de Angular mientras pasamos el nombre del servicio como parámetro:

```
ng generar productos de servicio
```

Al ejecutar el comando anterior se creará el servicio de productos , que consta de los productos. archivo service.ts y el archivo de prueba unitaria que lo acompaña, products.service.spec.ts.

Solemos nombrar un servicio según la funcionalidad que representa. Cada servicio tiene un contexto o dominio empresarial dentro del cual opera. Cuando empieza a traspasar los límites entre diferentes contextos, indica que debe dividirse en diferentes servicios. Un servicio de productos debe ocuparse de los productos. De igual forma, los pedidos deben ser gestionados por un servicio de pedidos independiente. servicio.

Un servicio Angular es una clase de TypeScript marcada con el decorador @Injectable . Este decorador identifica la clase como un servicio Angular que puede inyectarse en otros artefactos Angular, como componentes, directivas o incluso otros servicios. Acepta un objeto como parámetro con una sola propiedad llamada providedIn, que define qué inyector proporciona el servicio.

Un servicio Angular, de manera predeterminada, se registra con un inyector: el inyector raíz de la aplicación Angular, como se define en el archivo products.service.ts :

```
importar { Inyectable } desde '@angular/core';

@Inyectable({
    proporcionado en: 'root'
})
clase de exportación ProductosServicio {

    constructor() { }
}
```

Nuestro servicio no contiene ninguna implementación. Añadamos lógica para que nuestro componente...

Puedes usarlo:

1. Agregue la siguiente declaración para importar la interfaz del producto :

```
importar { Producto } desde './producto';
```

2. Cree el siguiente método en la clase ProductsService :

```
obtenerProductos(): Producto[]
{
    devolver
    [
        {
            identificación: 1,
            Título: 'Teclado', precio:
            100,
            categorías: {
                1: 'Computación',
                2: 'Periféricos'
            },
            {
                id: 2,
                título: 'Micrófono',
                precio: 35,
                categorías: { 3: 'Multimedia' }
            },
            {
                id: 3,
                título: 'Cámara web',
                precio: 79,
                categorías: {
                    1: 'Computación',
                    3: 'Multimedia'
                },
                {
                    id: 4,
                    título: 'Tableta',
                    precio: 500,
                    categorías: { 4: 'Entretenimiento' }
                }
            ];
        }
    ]
}
```

En las siguientes secciones aprenderemos cómo utilizar el servicio en nuestra aplicación.

Injectando servicios en el constructor

La forma más común de utilizar un servicio en un componente Angular es a través de su constructor:

1. Abra el archivo product-list.component.ts y modifique la propiedad de productos para que se inicializa en una matriz vacía:

```
productos: Producto[] = [];
```

2. Agregue la siguiente declaración para importar la clase ProductsService :

```
importar { ProductsService } desde '../products.service';
```

3. Cree una propiedad de componente llamada productService y asígnele un tipo ProductsService:

```
productoServicio privado : ProductosServicio;
```

4. Instancie la propiedad usando la palabra clave new en el constructor del componente :

```
constructor() {
  este.productService = nuevo ProductsService();
}
```

5. Importe la interfaz OnInit desde el paquete npm @angular/core :

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

6. Agregue la interfaz OnInit a la lista de interfaces implementadas de ProductListComponent clase:

La clase de exportación ProductListComponent implementa OnInit

7. Agregue el siguiente método ngOnInit que llama al método getProducts de la propiedad productService y asigna el valor devuelto a la propiedad products :

```
ngOnInit(): vacío {
  este.productos = este.servicioproducto.getProducts();
}
```

Ejecute la aplicación usando el comando ng serve para verificar que la lista de productos todavía se muestra correctamente en la página:

Products (4)



Copyright ©2024 All Rights Reserved

Figura 5.1: Lista de productos

¡Genial! Hemos conectado correctamente nuestro componente con el servicio y nuestra aplicación se ve genial. Bueno, parece ser así, pero en realidad no lo es. Hay algunos problemas con la implementación. Si la clase ProductsService debe cambiar, quizás para acomodar otra dependencia, ProductListComponent también debería cambiar la implementación de su constructor. Por lo tanto, es evidente que el componente de lista de productos está estrechamente vinculado a la implementación de ProductsService. Esto nos impide modificar, sobrescribir o probar el servicio si es necesario. También implica que se crea un nuevo objeto ProductsService cada vez que renderizamos un componente de lista de productos, lo cual podría no ser deseable en escenarios específicos, como cuando esperamos usar un servicio singleton real.

Los sistemas DI intentan resolver estos problemas proponiendo varios patrones y la inyección del constructor. El patrón es el que aplica Angular. Podríamos eliminar la propiedad del componente productService e injectar el servicio directamente en el constructor. El ProductListComponent resultante La clase sería la siguiente:

```
clase de exportación ProductListComponent implementa OnInit {  
  productos: Producto[] = [];  
  productoSeleccionado: Producto | indefinido;
```

```

constructor( productoServicioprivado: ProductosServicio) {}

al agregarlo() {
  alert(`${this.selectedProduct?.title} | agregado al carrito!`);

}

ngOnInit(): vacío {
  este.productos = este.servicioproducto.getProducts();
}
}

```



Considere declarar los servicios inyectados como de solo lectura para proporcionar un código más estable y evitar la reasignación del servicio. En el fragmento anterior, el constructor podría reescribirse como constructor(private readonly productService: ProductsService) {}.

El componente no necesita saber cómo instanciar el servicio. Sin embargo, espera que dicha dependencia esté disponible antes de instanciarlo para poder inyectarla a través de su constructor. Este enfoque es más fácil de probar, ya que permite sobrescribirlo o simularlo.

Sin embargo, usar un constructor no es la única forma de inyectar servicios en una aplicación Angular, como aprenderemos en la siguiente sección.

La palabra clave inyectar

El framework Angular incluye un método de inyección integrado que permite inyectar servicios sin usar el constructor. En algunos casos, nos gustaría usar este método :

- El constructor contiene muchos servicios inyectados, lo que hace que nuestro código sea ilegible.
- Los constructores no se pueden utilizar cuando se trabaja con funciones puras en el enrutador Angular o el cliente HTTP, como aprenderemos en los siguientes capítulos.

Veamos cómo podríamos refactorizar el componente de lista de productos para utilizar el método de inyección :

1. Abra el archivo product-list.component.ts e importe el método de inyección desde el Paquete npm de @angular/core :

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

2. Declare la siguiente propiedad en la clase ProductListComponent :

```
productoServicio privado = injectar(ProductosServicio);
```

3. Elimine el constructor de la clase ProductListComponent .

La aplicación debería seguir funcionando correctamente si ejecutamos el comando `ng serve` . La lista de productos debería mostrarse como en la sección anterior.

Exploraremos casos de uso adicionales para el método de inyección en el Capítulo 8, Comunicación con servicios de datos a través de HTTP, y en el Capítulo 9, Navegación por aplicaciones con enruteamiento.

En comparación con el enfoque del constructor , el método de inyección proporciona tipos más precisos, lo que refuerza las aplicaciones Angular fuertemente tipadas.

La CLI de Angular proporciona un esquema que podemos ejecutar para migrar al nuevo método de inyección . Puede encontrar más detalles sobre cómo ejecutar el esquema en <https://angular.dev/reference/migrations/injection-function>.



En este libro, utilizamos tanto el método de inyección como el enfoque del constructor, según el contexto de ejecución del código de la aplicación.

Como aprendimos, al crear un nuevo servicio Angular, la CLI de Angular lo registra con el inyector raíz de la aplicación por defecto. En la siguiente sección, aprenderemos sobre el funcionamiento interno del mecanismo de DI y el inyector raíz.

Proporcionar dependencias en toda la aplicación

El framework Angular ofrece un mecanismo de DI para proporcionar dependencias en artefactos Angular, como componentes, directivas, tuberías y servicios. El DI Angular se basa en una jerarquía de inyectores, donde en la parte superior se encuentra el inyector raíz de una aplicación Angular.

Los inyectores en Angular pueden examinar las dependencias en el constructor de un artefacto Angular y devolver una instancia del tipo representado por cada dependencia, para que podamos usarla directamente en la implementación de nuestra clase Angular. El inyector mantiene una lista de todas las dependencias que necesita una aplicación Angular. Cuando un componente u otro artefacto desea usar una dependencia, el inyector primero verifica si ya ha creado una instancia de esta. De no ser así, crea una nueva, la devuelve al componente y guarda una copia para su uso posterior. La próxima vez que se solicita la misma dependencia, devuelve la copia creada previamente. Pero ¿cómo sabe el inyector qué dependencias necesita una aplicación Angular?

Al crear un servicio Angular, usamos la propiedad `providedIn` del decorador `@Injectable` para definir cómo se proporciona a la aplicación. Es decir, creamos un proveedor para este servicio. Un proveedor es una receta que contiene las instrucciones para crear un servicio específico. Durante el inicio de la aplicación, el framework se encarga de configurar el inyector con los proveedores de servicios para que sepa cómo crear uno cuando se le solicite. Un servicio Angular se configura con el inyector raíz por defecto al crearse con la CLI. El inyector raíz crea servicios singleton disponibles globalmente a través de la aplicación.

En el Capítulo 1, "Creando tu primera aplicación Angular", aprendimos que el objeto de configuración de la aplicación definido en el archivo `app.config.ts` tiene una propiedad "proveedores" donde podemos registrar los servicios de la aplicación. Podríamos eliminar la propiedad "providedIn" del decorador `@Injectable` del archivo `products.service.ts` y añadirla directamente a ese array. Registrar un servicio de esta manera es lo mismo que configurarlo con "providedIn: 'root'". La principal diferencia entre ambos es que la sintaxis "providedIn" permite la manipulación de árboles.



Tree Shaking es el proceso de encontrar dependencias que no se utilizan en una aplicación y eliminarlas del paquete final. En el contexto de Angular, el compilador de Angular puede detectar y eliminar servicios de Angular que no se utilizan, lo que resulta en... paquete más pequeño.

Al proporcionar un servicio a través del objeto de configuración de la aplicación, el compilador de Angular no puede determinar si el servicio se utiliza en alguna parte de la aplicación. Por lo tanto, lo incluye a priori en el paquete final . Por lo tanto, es preferible usar el decorador `@Injectable` sobre el array de proveedores de la configuración de la aplicación.



Siempre debe registrar los servicios singleton con el inyector raíz.

El inyector raíz no es el único en una aplicación Angular. Los componentes también tienen sus propios inyectores. Los inyectores Angular también son jerárquicos. Cuando un componente Angular define un token en su constructor, el inyector busca un tipo que coincida con dicho token en el conjunto de proveedores registrados. Si no encuentra ninguna coincidencia, delega la búsqueda al proveedor del componente principal y continúa expandiendo el árbol de inyectores del componente hasta llegar al inyector raíz. Si no encuentra ninguna coincidencia, Angular lanza una excepción.

Exploraremos la jerarquía de inyectores del componente de lista de productos usando Angular DevTools:

1. Ejecute la aplicación usando el comando `ng serve` y obtenga una vista previa en `http://localhost:4200`.
2. Inicie Angular DevTools y seleccione la pestaña Componentes .
3. Seleccione el componente `app-product-list` del árbol de componentes:

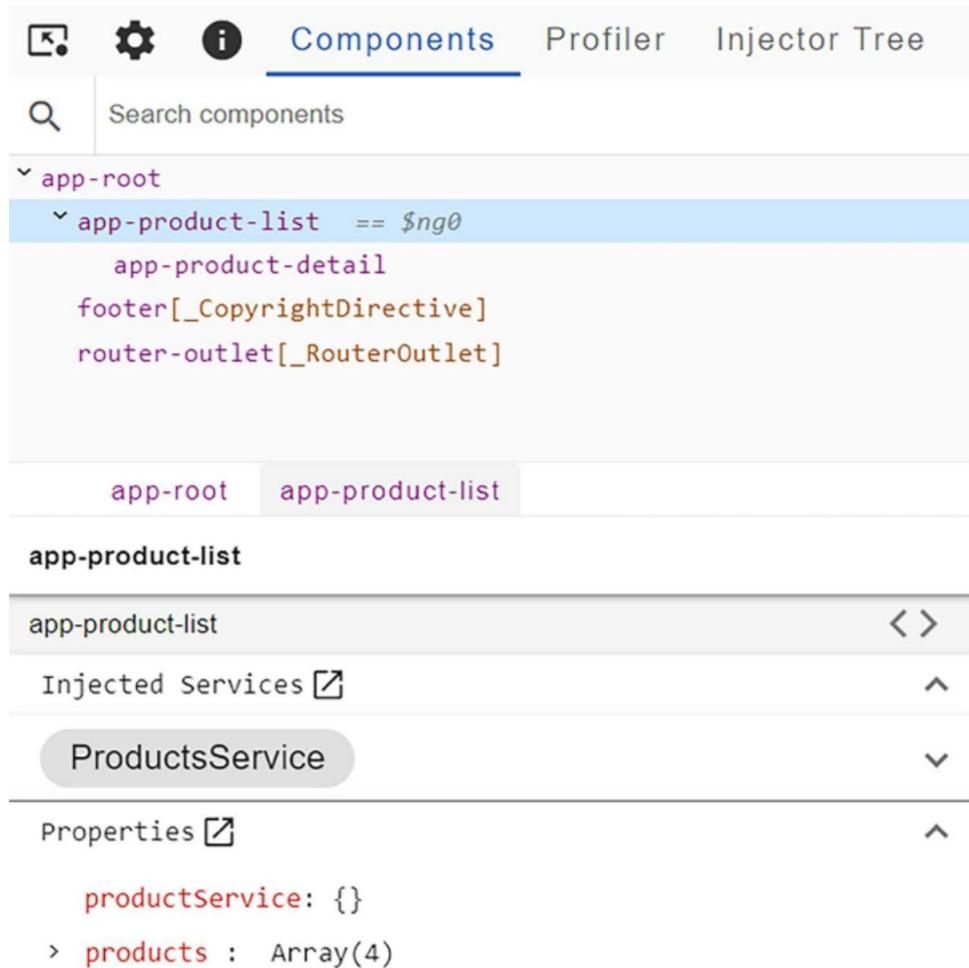


Figura 5.2: Pestaña Componentes

En la imagen anterior, la sección Servicios Inyectados contiene los servicios inyectados en el componente.

4. Haga clic en la flecha hacia abajo junto a la etiqueta `ProductosServicio` y verá lo siguiente diagrama:

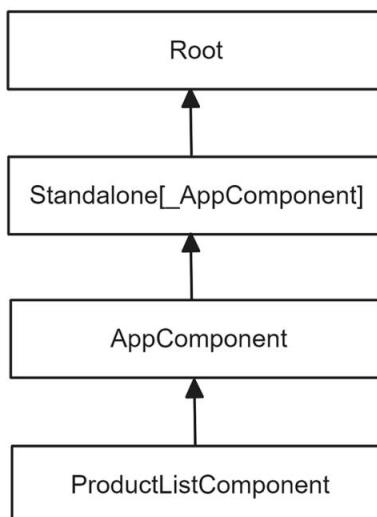


Figura 5.3: Jerarquía de inyectores de la lista de productos



El diagrama de jerarquía de inyectores en Angular DevTools tiene orientación horizontal. Aquí lo mostramos verticalmente para facilitar la lectura.

El diagrama anterior muestra la jerarquía de inyectores del componente de lista de productos. Contiene dos tipos principales de jerarquía de inyectores comunes en una aplicación Angular: entorno y elemento inyectores.

Los inyectores de entorno se configuran mediante la propiedad `providedIn` y la matriz de proveedores en el objeto de configuración de la aplicación. En nuestro caso, vemos los inyectores `Root` y `Standalone[_AppComponent]` porque el servicio de productos se proporciona desde el inyector raíz mediante la propiedad `providedIn`.

Angular crea un inyector de elementos para cada componente, que se puede configurar desde la matriz de proveedores del decorador `@Component`, como veremos en la siguiente sección. En nuestro caso, vemos los inyectores `AppComponent` y `ProductListComponent`, ya que estos componentes están directamente relacionados con la lista de productos.



Puedes seleccionar la pestaña "Árbol de inyectores" de Angular DevTools para un análisis más detallado de la jerarquía de inyectores de la aplicación por tipo. También puedes obtener más información sobre los diferentes tipos de inyectores en <https://angular.dev/guide/di/hierarchical-dependency-injection#types-of-injector-hierarchies>.

Los componentes crean inyectores, por lo que están disponibles inmediatamente para sus componentes secundarios. Veremos esto en detalle en la siguiente sección.

Inyección de servicios en el árbol de componentes

Como aprendimos en la sección anterior, Angular utiliza un inyector de elementos para proporcionar servicios en los componentes mediante la propiedad "providers" del decorador `@Component`. Un servicio que se registra con el inyector de elementos puede cumplir dos funciones:

- Se puede compartir con sus componentes secundarios.
- Puede crear múltiples copias del servicio cada vez que el componente que lo proporciona el servicio se presta

En las siguientes secciones, aprenderemos cómo aplicar cada enfoque.

Compartir dependencias a través de componentes

Un servicio proporcionado a través de un componente puede compartirse entre los componentes secundarios del componente principal y está disponible inmediatamente para su inyección en sus constructores. Los componentes secundarios reutilizan la misma instancia del servicio que el componente principal. Veamos un ejemplo para comprenderlo mejor:

1. Crea un nuevo componente Angular llamado favoritos:

```
ng generar componentes favoritos
```

2. Abra el archivo `favorites.component.ts` y modifique las declaraciones de importación según corresponda:

```
importar { Componente, OnInit } desde '@angular/core';
importar { Producto } de '../producto';
importar { ProductsService } desde '../products.service';
```

3. Modifique la clase FavoritesComponent para usar la clase ProductService y obtener el producto.

Lista de productos en una propiedad del componente de productos :

```
clase de exportación FavoritesComponent implementa OnInit {
    productos: Producto[] = [];

    constructor( productoServicioprivado: ProductosServicio ) {}

    ngOnInit(): vacío {
        este.productos = este.servicioproducto.getProducts();
    }
}
```

4. Abra el archivo favorites.component.html y reemplace su contenido con lo siguiente

Código HTML:

```
<ul class="grupo de píldoras">
    @for (producto de productos | slice:1:3; seguimiento producto.id) { <li
        class="pill">
            ★ {{producto.título}}
        </li>
    }
</ul>
```

En el fragmento anterior, iteramos sobre la matriz de productos y usamos la tubería de corte para mostrar solo dos productos.

5. Modifique el archivo favorites.component.ts para que importe la clase CommonModule que

Se necesita para la tubería de corte :

```
importar { CommonModule } de '@angular/common'; importar
{ Component, OnInit } de '@angular/core'; importar { Product } de
'./product';
importar { ProductsService } desde './products.service';

@Component({ selector: 'aplicación-favoritos',
importaciones: [CommonModule],
templateUrl: './favorites.component.html',
```

```
styleUrl: './favorites.component.css' })
```

6. Abra el archivo favorites.component.css para agregar algunos estilos CSS a nuestros productos favoritos:

```
.grupo de píldoras {  
    pantalla: flex; flex-  
    direction: columna; align-items:  
    inicio; flex-wrap: ajustar;  
  
    brecha: 1,25rem;  
}  
  
.pill  
{ pantalla: flex;  
    alinear-elementos: centro;  
    --pill-accent: var(--hot-red);  
    fondo: color-mix(en srgb, var(--hot-red) 5%, transparente); color: var(--pill-accent);  
  
    relleno en línea: 0,75rem; bloque  
    de relleno: 0,375rem;  
    radio del borde: 2,75rem;  
    borde: 0;  
    transición: fondo 0.3s facilidad; familia-de-  
    fuentes: var(--inter-font);  
    tamaño de fuente: 0.875rem;  
    estilo de fuente: normal;  
    peso de fuente: 500;  
    altura de línea: 1.4rem;  
    espaciado entre letras: -0,00875rem;  
    decoración de texto: ninguna;  
}
```

7. Abra el archivo product-list.component.ts , importe la clase FavoritesComponent y agregue la clase ProductsService a la matriz de proveedores del decorador @Component :

```
importar { Componente, OnInit } desde '@angular/core';
importar { Producto } de '../producto';
importar { ProductDetailComponent } de '../product-detail/product-detail.component';

importar { SortPipe } desde '../sort.pipe';
importar { ProductsService } desde './products.service';
importar { FavoritesComponent } desde './favorites/favorites.
componente';

@Component({
  selector: 'lista-de-productos-de-aplicaciones',
  importaciones: [ProductDetailComponent, SortPipe, FavoritesComponent],
  templateUrl: './lista-de-productos.componente.html',
  styleUrls: ['./lista-de-productos.componente.css'],
  proveedores: [ProductosServicio]
})
```

8. Abra el archivo products.service.ts y elimine la propiedad providedIn del decorador @Injectable ya que el inyector de elementos del componente de lista de productos la proporcionará.
9. Finalmente, abra el archivo product-list.component.html y agregue el siguiente fragmento HTML Para mostrar el contenido del componente favoritos:

```
Favoritos
<favoritos de la aplicación></favoritos de la aplicación>
```

Al ejecutar la aplicación usando ng serve, debería ver el siguiente resultado:

Products (4)

Keyboard

Microphone

Tablet

Web camera

Favorites

★ Microphone

★ Web camera

Copyright ©2024 All Rights Reserved

Figura 5.4: Lista de productos con favoritos

Expliquemos con más detalle lo que hicimos en el ejemplo anterior. Inyectamos ProductsService. en FavoritesComponent , pero no lo proporcionamos a través de su inyector. Entonces, ¿cómo sabía el componente cómo crear una instancia de la clase ProductsService y usarla? No lo sabía. Al añadir el componente "favoritos" a la plantilla "ProductListComponent" , lo convertimos en un componente secundario directo de este, lo que le otorga acceso a todos sus servicios. En resumen, "FavoritesComponent" puede usar "ProductsService" de forma predeterminada, ya que se proporciona a través del inyector de elementos de su componente principal, "ProductListComponent".

Por lo tanto, incluso si `ProductsService` se registró inicialmente con el inyector raíz del entorno, también podríamos registrarlo con el inyector de elementos de `ProductListComponent`. En la siguiente sección, investigaremos cómo lograr este comportamiento.

Inyectores de raíz y componentes

Ya hemos aprendido que, al crear un servicio Angular con la CLI de Angular, este se proporciona por defecto en el inyector raíz de la aplicación. ¿Cuál es la diferencia al proporcionar un servicio a través del inyector de elementos de un componente?

Los servicios proporcionados con el inyector raíz de la aplicación están disponibles en toda la aplicación . Cuando un componente desea utilizar dicho servicio, solo necesita inyectarlo, nada más.

Ahora bien, si el componente proporciona el mismo servicio a través de su inyector, obtendrá una instancia del servicio completamente diferente a la del inyector raíz. Esta técnica se denomina limitación del alcance del servicio , ya que limitamos el alcance del servicio a una parte específica del árbol de componentes:

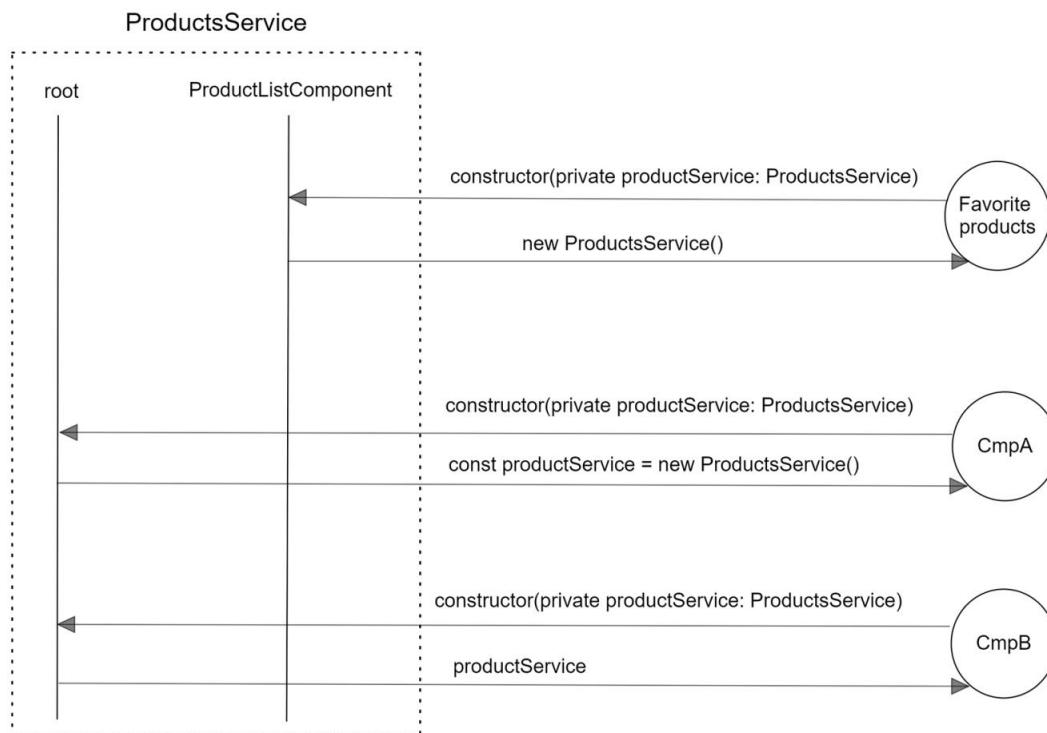


Figura 5.5: Limitación del alcance del servicio

El diagrama anterior muestra que ProductsService se puede proporcionar a través de dos inyectores: el inyector de raíz de la aplicación y el inyector de elementos del componente de lista de productos.

La clase FavoritesComponent inyecta ProductsService para usarlo. Como ya hemos visto, FavoritesComponent es un componente secundario de ProductListComponent.

Según la jerarquía del inyector, primero preguntará al inyector de su componente principal, ProductListComponent, sobre la prestación del servicio. La clase ProductListComponent proporciona ProductsService, por lo que crea una nueva instancia del servicio y la devuelve a Componente Favoritos.

Ahora, considere que otro componente de nuestra aplicación, CmpA, desea utilizar ProductsService.

Dado que no es un componente secundario de ProductListComponent y no contiene ningún componente principal que proporcione el servicio requerido, finalmente llegará al inyector raíz de la aplicación. El inyector raíz que proporciona ProductsService comprueba si ya ha creado una instancia para ese servicio. De no ser así, crea una nueva, llamada productService, y la devuelve a CmpA. También conserva productService en el conjunto local de servicios para su uso posterior.

Supongamos que otro componente, CmpB, desea usar ProductsService y solicita la autorización del inyector raíz de la aplicación. El inyector raíz sabe que ya ha creado la instancia de productService cuando CmpA... lo solicitó y lo devuelve inmediatamente al componente CmpB .

Componentes de sandbox con múltiples instancias

Cuando proporcionamos un servicio mediante el inyector de elementos y lo inyectamos en el constructor del componente, se crea una nueva instancia cada vez que el componente se renderiza en la página.

Esto puede ser útil en casos como cuando queremos tener un servicio de caché local para cada componente. Exploraremos este escenario transformando nuestra aplicación Angular para que la lista de productos muestre una vista rápida de cada producto usando un servicio Angular:

1. Ejecute el siguiente comando para crear un nuevo componente Angular para la vista del producto:

```
ng genera el componente vista-del-producto
```

2. Abra el archivo product-view.component.ts y declare una propiedad de entrada llamada id .

Podemos pasar un identificador único del producto que queremos mostrar:

```
importar { Componente, entrada } desde '@angular/core';

@Component({
  selector: 'aplicación-producto-vista',
  importaciones: [],
```

```

    URL de plantilla: './product-view.component.html',
    styleUrls: ['./product-view.component.css'
  })
clase de exportación ProductViewComponent {
  id = entrada<número>();
}

```

3. Ejecute el siguiente comando CLI de Angular dentro de la carpeta product-view para crear un Servicio angular que se dedicará al componente de vista del producto:

```
ng generar servicio vista-de-producto
```

4. Abra el archivo product-view.service.ts y elimine la propiedad providedIn del decorador @Injectable porque la proporcionaremos más adelante en el componente de vista del producto.
5. Inyecte ProductsService en el constructor de la clase ProductViewService :

```

importar { Inyectable } desde '@angular/core';
importar { ProductsService } desde '../products.service';

@Inyectable()
clase de exportación ProductViewService {

  constructor( productoServicioprivado: ProductosServicio ) {}
}

```

La técnica anterior se llama servicio en servicio porque inyectamos un servicio Angular en otro.

6. Cree un método llamado getProduct que tome la propiedad id como parámetro. El método `meth-od` llamará al método `getProducts` de la clase `ProductsService` y buscará en la lista de productos según el id. Si encuentra el producto, lo guardará en una variable local llamada `product`:

```

importar { Inyectable } desde '@angular/core';
importar { ProductsService } desde '../products.service';
importar { Producto } de '../producto';

@Inyectable()
clase de exportación ProductViewService {
  producto privado: Producto | indefinido;
}

```

```
constructor( productoServicioprivado: ProductosServicio ) { }

obtenerProducto(id: número): Producto | indefinido {
    const productos = este.productService.getProducts(); si (!
    este.producto) {
        este.producto = productos.find(producto => producto.id === id)
    }
    devuelve este.producto;
}
}
```

Ya hemos creado los artefactos Angular esenciales para trabajar con el componente de vista de producto.

Ahora solo nos queda conectarlos a la lista de productos:

1. Inyecte ProductViewService en el constructor de ProductViewComponent e im-

Complemente el método ngOnInit :

```
importar { Componente, entrada, OnInit } desde '@angular/core';
importar { ProductViewService } desde './product-view.service';
```

```
@Component({ selector: 'app-product-view',
    importaciones:
    [], templateUrl: './product-view.component.html',
    styleUrls: ['./product-view.component.css'], proveedores:
    [ProductViewService]
})
exporta la clase ProductViewComponent implementa OnInit {
```

```
    id = entrada<número>();
```

```
    constructor(private productViewService: ProductViewService) {}
```

```
    ngOnInit(): vacío {
```

```
}
```

```
}
```

2. Crea una propiedad de componente para guardar el producto que obtendremos de la Clase ProductViewService :

```
importar { Componente, entrada, OnInit } desde '@angular/core';
importar { ProductViewService } desde './product-view.service'; importar
{ Producto } desde '../product';

@Componente({
  selector: 'app-product-view',
  importaciones: [],
  URL de plantilla: './product-view.component.html', URL de
  estilo: './product-view.component.css',
  proveedores: [ProductViewService] })

clase de exportación ProductViewComponent implementa OnInit {
  id = input<número>();
  producto: Producto | indefinido;

  constructor(private productViewService: ProductViewService) {}

  ngOnInit(): vacío {

  }
}
```

3. Modifique el método ngOnInit para que llame al método getProduct del

Clase ProductViewService como sigue:

```
ngOnInit(): void
  { este.producto = este.productViewService.getProduct(este.id()); }
```

En el fragmento anterior, pasamos la propiedad del componente id al método getProduct como parámetro y asignamos el valor devuelto a la propiedad del producto .

4. Abra el archivo product-view.component.html y reemplace su contenido con lo siguiente

Plantilla HTML:

```
@switch (producto?.título) {
  @case ('Teclado') {  }
```

```

@case ('Micrófono') 
{ @default {  }

} {{producto?.título}}

```

5. Abra el archivo product-list.component.ts e importe la clase ProductViewComponent :

```

importar { Componente, OnInit } de '@angular/core'; importar
{ Producto } de './product';
importar { ProductDetailComponent } desde './product-detail/product-detail.component';
importar { SortPipe }
desde './sort.pipe'; importar { ProductsService }
desde './products.service'; importar { ProductViewComponent } desde
'./product-view/product-view.
componente';

```

```

@Component({ selector: 'app-product-
list', importaciones: [ProductDetailComponent, SortPipe, ProductViewComponent], URL
de plantilla: './product-list.component.html', URL de estilo: './
product-list.component.css' })

```

6. Finalmente, abra el archivo product-list.component.html y modifique el bloque @for para utilizar

El componente de vista del producto:

```

<ul class="grupo de píldoras">
@for (producto de productos | ordenar; rastrear producto.id) {
    <li class="pill" (clic)="productoSeleccionado = producto">
        <app-product-view [id]="producto.id"></app-product-view>
    </li>
} @vacío {
    <p>¡No se encontraron productos!</p>
}
</ul>

```

Si ejecutamos nuestra aplicación con el comando ng serve , veremos que la lista de productos todavía se muestra correctamente.

Cada componente de vista de producto renderizado crea un ProductViewService en espacio aislado dedicado Instancia para su propósito. Ningún otro componente puede compartir la instancia ni ser modificado, excepto por el componente que la proporciona.

Intente proporcionar ProductViewService en ProductListComponent en lugar de ProductViewComponent; verá que solo se representa un producto varias veces:

Products (4)

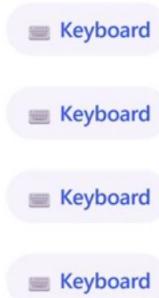


Figura 5.6: Lista de productos

En este caso, solo se comparte una instancia de servicio entre los componentes secundarios. ¿Por qué? Recuerde la lógica de negocio del método getProduct de la clase ProductViewService :

```
obtenerProducto(id: número): Producto | indefinido {
  const productos = este.productService.getProducts();
  si (!este.producto) {
    este.producto = productos.find(producto => producto.id === id)
  }
  devuelve este.producto;
}
```

En el método anterior, la propiedad del producto se establece inicialmente al proporcionar el servicio dentro de ProductListComponent. Dado que solo tenemos una instancia del servicio, el valor de la propiedad se mantendrá igual al renderizar el componente de vista del producto varias veces.

Hemos aprendido cómo se inyectan las dependencias en la jerarquía de componentes y cómo se realiza la búsqueda de proveedores mediante la propagación de la solicitud hacia arriba en el árbol de componentes. Sin embargo, ¿qué ocurre si queremos restringir dichas acciones de inyección o búsqueda? Veremos cómo hacerlo en la siguiente sección.

Restricción de la búsqueda de proveedores: Solo

podemos restringir la búsqueda de dependencias al nivel superior. Para ello, debemos aplicar el decorador @Host a los parámetros de dependencia cuya búsqueda de proveedores queremos restringir:

```
importar { CommonModule } desde '@angular/common';
importar { Component, Host, OnInit } desde '@angular/core'; importar
{ Product } desde '../product'; importar
{ ProductsService } desde '../products.service';

@Component({ selector: 'app-
favorites', importaciones:
[CommonModule], templateUrl: './
favorites.component.html', styleUrls: './

favorites.component.css' }) exporta clase FavoritesComponent implementa OnInit {
  productos: Producto[] = [];

  constructor(@Host() private productService: ProductsService) {}

  ngOnInit(): vacío {
    este.productos = este.servicioproducto.getProducts();
  }
}
```

En el ejemplo anterior, el inyector de elementos de FavoritesComponent buscará la clase ProductsService en sus proveedores. Si no proporciona el servicio, no se propagará a través de la jerarquía de inyectores; en su lugar, se detendrá y lanzará una excepción en la ventana de consola del navegador:

Error: NG0201: No se encontró ningún proveedor para _ProductsService en NodeInjector

Podemos configurar el inyector para que no lance un error si decoramos el servicio con el decorador @Optional :

```
importar { CommonModule } desde '@angular/common';
importar { Componente, Host, OnInit, Opcional } de '@angular/core'; importar
{ Producto } de '../product';
importar { ProductsService } desde '../products.service';
```

```
@Componente({  
    selector: 'aplicación-favoritas',  
    importaciones: [CommonModule],  
    URL de plantilla: './favorites.component.html',  
    styleUrl: './favorites.component.css'  
})  
clase de exportación FavoritesComponent implementa OnInit {  
    productos: Producto[] = [];  
  
    constructor(@Optional() @Host() productoServicio privado : ProductosServicio) {}  
  
    ngOnInit(): vacío {  
        este.productos = este.servicioproducto.getProducts();  
    }  
}
```

Sin embargo, usar el decorador `@Optional` no resuelve el problema. El fragmento anterior seguirá generando un error, diferente al anterior, porque seguimos usando el decorador `@Host`, que limita la búsqueda de la clase `ProductsService` en la jerarquía de inyectores. Necesitamos refactorizar el evento de gancho del ciclo de vida `ngOnInit` para que se encargue de no encontrar la instancia del servicio.

Los decoradores `@Host` y `@Optional` definen el nivel en el que el inyector busca dependencias. Hay otros dos decoradores, llamados `@Self` y `@SkipSelf`. Al usar `@Self`

Decorador: el inyector busca dependencias en el inyector del componente actual. Por el contrario, el decorador `@SkipSelf` indica al inyector que omita el inyector local y busque más arriba en la jerarquía de inyectores.



Los decoradores `@Host` y `@Self` funcionan de forma similar. Para más información sobre cuándo usar cada uno, consulta <https://angular.dev/guide/di/hierarchical-dependency-injection#self> y <https://angular.dev/guide/di/injection-dependency#host>.

Hasta ahora, hemos aprendido cómo el framework Angular DI usa clases como tokens de dependencia para determinar el tipo requerido y devolverlo desde cualquier proveedor disponible en la jerarquía de inyectores. Sin embargo, en algunos casos, podríamos necesitar sobrescribir la instancia de una clase o proporcionar tipos que no son clases reales, como los tipos primitivos.

Anulación de proveedores en la jerarquía de inyectores

Ya aprendimos a utilizar la matriz de proveedores del decorador @Component en la sección Compartir dependencias a través de componentes :

```
proveedores: [ProductosServicio]
```

La sintaxis anterior se denomina sintaxis del proveedor de clase y es una abreviatura de la sintaxis del literal de objeto provide que se muestra a continuación:

```
proveedores: [
  { proporcionar: ProductosServicio, usarClase: ProductosServicio }
]
```

La sintaxis anterior utiliza un objeto con las siguientes propiedades:

- Proporcionar: Este es el token utilizado para configurar el inyector. Es la clase que los consumidores de la dependencia inyectan en sus constructores.
- useClass: esta es la implementación real que el inyector proporcionará a los consumidores.
El nombre de la propiedad variará según el tipo de implementación proporcionado. El tipo puede ser una clase, un valor o una función de fábrica. En este caso, usamos useClass porque proporcionamos una clase.

Veamos algunos ejemplos para obtener una descripción general de cómo utilizar la sintaxis de proporcionar objeto literal.

Implementación del servicio de anulación

Ya hemos aprendido que un componente podría compartir sus dependencias con sus componentes secundarios. Considere el componente Favoritos, donde usamos la tubería de segmentación para mostrar una lista de productos favoritos en su plantilla. ¿Qué sucede si necesita obtener datos a través de una versión reducida de ProductsService? ¿Y no directamente desde la instancia de servicio de ProductListComponent? Podríamos crear un nuevo servicio que extienda la clase ProductsService y filtre los datos usando el Array™slice nativo.

Método. Creemos el nuevo servicio y aprendamos a usarlo:

1. Ejecute el siguiente comando para generar el servicio:

```
ng genera favoritos de servicio
```

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

```
importar { Producto } desde './producto';
importar { ProductsService } desde './products.service';
```

3. Utilice la palabra clave extends en la definición de clase para indicar que ProductsService es el clase base de FavoritesService:

```
clase de exportación FavoritesService extiende ProductsService {  
  
    constructor() {}  
}
```

4. Modifique el constructor para llamar al método super y ejecutar cualquier lógica de negocios dentro el constructor de la clase base:

```
constructor()  
{ super();  
}
```

5. Cree el siguiente método de servicio que utiliza el método slice para devolver solo el primer dos productos de la lista:

```
anular getProducts(): Producto[] {  
    devolver super.getProducts().slice(1, 3);  
}
```

El método anterior está marcado con la palabra clave override para indicar que la implementación del método reemplaza el método correspondiente de la clase base.

6. Abra el archivo favorites.component.ts y agregue la siguiente declaración de importación :

```
importar { FavoritesService } desde './favorites.service';
```

7. Agregue la clase FavoritesService en la matriz de proveedores del decorador @Component como Sigue:

```
@Componente({  
    selector: 'app-favorites',  
    importaciones: [],  
    URL de plantilla: './favorites.component.html', URL de  
    estilo: './favorites.component.css',  
    proveedores: [  
        { proporcionar: ProductosServicio, usarClase: FavoritosServicio }  
    ]  
})
```

En el fragmento anterior, eliminamos CommonModule de la matriz de importaciones porque ya no necesitamos la tubería de corte .

8. Por último, abra el archivo favorites.component.html y elimine la tubería de corte del @para bloque.

Si ejecutamos la aplicación mediante el comando ng serve , veremos que la sección Favoritos aún se muestra correctamente:

Favorites

★ Microphone

★ Web camera

Figura 5.7: Lista de productos favoritos



La salida anterior supone que ya ha importado y agregado el componente de favoritos en el componente de lista de productos.

La propiedad useClass esencialmente sobrescribió la implementación inicial de ProductsService Clase para el componente de favoritos. Como alternativa, podemos ir más allá y usar una función para devolver una instancia de objeto específica que necesitemos, como veremos en la siguiente sección.

Prestación de servicios de forma condicional

En el ejemplo de la sección anterior, usamos la sintaxis useClass para reemplazar la implementación de la clase ProductsService inyectada . Como alternativa, podríamos crear una función de fábrica que decida si devolverá una instancia de la clase FavoritesService o ProductsService .

Según una condición. La función residiría en un archivo TypeScript simple llamado favorites.ts:

```
importar { FavoritesService } desde './favorites.service';
importar { ProductsService } desde './products.service';

función de exportación favoritesFactory(isFavorite: boolean) {
  devolver () => {
    si (esFavorito) {
```

```
        devolver nuevo FavoritesService();
    }
    devolver nuevo ProductsService();
};
}
```

Luego podríamos modificar la matriz de proveedores en el archivo favorites.component.ts de la siguiente manera:

```
importar { CommonModule } de '@angular/common'; importar
{ Component, OnInit } de '@angular/core'; importar { Product } de
'./product';
importar { ProductsService } de './products.service'; importar
{ favoritesFactory } de './favorites';

@Component({
  selector: 'app-favorites', importa:
[CommonModule],
  URL de plantilla: './favorites.component.html', URL de
  estilo: './favorites.component.css',
  proveedores: [
    { proporcionar: ProductosServicio, useClass: favoritesFactory(true) }
  ]
})
```

Cabe destacar que si uno de los servicios también injectara otras dependencias, la sintaxis anterior no sería suficiente. Por ejemplo, si la clase FavoritesService dependiera de la clase ProductViewService , la añadiríamos a la propiedad deps del literal de objeto "provider". sintaxis:

```
proveedores: [ {

  proporcionar: ProductsService,
  useClass: favoritesFactory(true), deps:
  [ProductViewService]
}
]
```

Luego podríamos usarlo en la función de fábrica del archivo favorites.ts de la siguiente manera:

```
función de exportación favoritesFactory(isFavorite: boolean) {
    devolver (productoViewService: ProductoViewService) => {
        si (esFavorito) {
            devolver nuevo FavoritesService();
        }
        devolver nuevo ProductsService();
    };
}
```

Ya hemos aprendido cómo proporcionar una implementación de clase alternativa para un servicio Angular.

¿Qué sucede si la dependencia que queremos proporcionar no es una clase, sino una cadena o un objeto?

Podemos usar la sintaxis `useValue` para lograrlo.

Transformación de objetos en servicios angulares

Es común mantener la configuración de la aplicación en un objeto constante en aplicaciones reales. ¿Cómo podríamos usar la sintaxis `useValue` para proporcionar esta configuración en nuestros componentes?

Aprenderemos más creando configuraciones para nuestra aplicación, como el número de versión y el título:

1. Cree un archivo `app.settings.ts` en la carpeta `src\app` del espacio de trabajo Angular CLI y

Añade el siguiente contenido:

```
interfaz de exportación AppSettings {
    título: cadena;
    versión: cadena;
}

exportar const appSettings: AppSettings = {
    Título: 'Mi tienda online',
    versión: '1.0'
};
```

Podrías pensar que podríamos proporcionar estas configuraciones como `{ provide: AppSettings, useValue: appSettings }`, pero esto generaría un error porque `AppSettings` es una interfaz, no una clase. Las interfaces son simples sintácticas en TypeScript que se descartan durante la compilación. En su lugar, deberíamos proporcionar un objeto `InjectionToken`.

2. Agregue la siguiente declaración para importar la clase `InjectionToken` desde el paquete `npm @angular/core` :

```
importar { InjectionToken } desde '@angular/core';
```

3. Declare la siguiente variable constante que utiliza el tipo `InjectionToken` :

```
exportar const APP_SETTINGS = nuevo InjectionToken<AppSettings>('app.settings');
```

4. Abra el archivo `app.component.ts` y modifique las declaraciones de importación de la siguiente manera:

```
importar { Componente, inyectar } de '@angular/core'; importar
{ RouterOutlet } de '@angular/router'; importar
{ ProductListComponent } de './product-list/product-list.component'; importar

{ CopyrightDirective } de './copyright.directive'; importar { APP_SETTINGS,
appSettings } de './app.settings';
```

5. Agregue el token de configuración de la aplicación en la matriz de proveedores del decorador `@Component` :

```
@Componente({
  selector: 'app-root',
  importaciones: [
    Salida de enrutador,
    [Componente de lista de
      productos, Directiva de

      derechos de autor ], URL de plantilla: './app.component.html',
      styleUrls: ['./app.component.css'], proveedores:
    [
      { proporcionar: APP_SETTINGS, valor de uso: appSettings }
    ]
  })
})
```



La sintaxis `useValue` es especialmente útil al probar aplicaciones Angular . La usaremos ampliamente cuando aprendamos sobre pruebas unitarias en el Capítulo 13, Pruebas unitarias de aplicaciones Angular.

6. Agregue la siguiente propiedad en la clase AppComponent :

```
configuraciones = injectar(APP_SETTINGS);
```

7. Abra el archivo app.component.html y modifique la etiqueta <footer> para incluir la aplicación.

Versión de ción:

```
<footer appCopyright> - v{{ settings.version }}</footer>
```

8. Ejecute la aplicación usando el comando ng serve y observe el pie de página en la aplicación.

Salida de catión:

```
Copyright ©2024 Todos los derechos reservados - v1 0
```

Tenga en cuenta que si bien la interfaz AppSettings no jugó un papel importante en el proceso de inyección, la necesitamos para proporcionar escritura en el objeto de configuración.

Angular DI es un mecanismo potente y robusto que nos permite gestionar las dependencias de nuestras aplicaciones de forma eficiente. El equipo de Angular se ha esforzado mucho para simplificarlo y eliminar la carga de trabajo del desarrollador. Como hemos visto, existen numerosas combinaciones, y cómo las usaremos dependerá del caso de uso.

Resumen

La implementación de Angular DI es la columna vertebral del framework Angular. Los componentes Angular delegan tareas complejas a servicios Angular, basándose en Angular DI.

En este capítulo, aprendimos qué es Angular DI y cómo aprovecharlo creando servicios Angular. Exploramos diferentes maneras de injectar servicios Angular en componentes. Vimos cómo compartir servicios entre componentes, aislarlos en componentes y definir el acceso a dependencias mediante el árbol de componentes.

Finalmente, investigamos cómo anular los servicios Angular reemplazando la implementación del servicio o transformando objetos existentes en servicios.

En el próximo capítulo, aprenderemos qué es la programación reactiva y cómo podemos usar observables en el contexto de aplicaciones Angular.

Machine Translated by Google

6

Patrones reactivos en Angular

Gestionar información asíncrona es una tarea común en nuestra vida diaria como desarrolladores. La programación reactiva es un paradigma que nos ayuda a consumir, procesar y transformar información asíncrona mediante flujos de datos. RxJS es una biblioteca de JavaScript que proporciona métodos para manipular flujos de datos mediante observables.

Angular ofrece un conjunto de herramientas inigualable para facilitar el trabajo con datos asíncronos. Los flujos observables son la base de este conjunto de herramientas, ofreciendo a los desarrolladores un amplio abanico de capacidades para crear aplicaciones Angular. El núcleo del framework Angular depende ligeramente de RxJS. Otros paquetes Angular, como el enrutador y el cliente HTTP, están más estrechamente vinculados con los observables. Sin embargo, al momento de escribir este artículo, el equipo Angular está investigando cómo reducir la dependencia de los paquetes anteriores con respecto a los observables.

En este capítulo aprenderemos los siguientes conceptos:

- Estrategias para el manejo de información asíncrona
- Programación reactiva en Angular
- La biblioteca RxJS
- Suscribirse a observables
- Cancelar la suscripción a observables

Requisitos técnicos

El capítulo contiene varios ejemplos de código que te guiarán a través de los observables y RxJS. Puedes encontrar el código fuente relacionado en la carpeta ch06 del siguiente repositorio de GitHub:

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

Estrategias para el manejo de información asincrónica

Gestionamos datos de forma asíncrona de diferentes formas, como al consumir datos de una API de backend (una operación típica en nuestro flujo de trabajo de desarrollo diario) o al leer contenido del sistema de archivos local. Siempre consumimos datos mediante HTTP, por ejemplo, al autenticar usuarios mediante el envío de credenciales a un servicio de autenticación. También usamos HTTP para obtener las últimas publicaciones en nuestra aplicación de red social favorita.

Los dispositivos móviles modernos han introducido una forma única de consumir servicios remotos. Aplazan el consumo de solicitudes y respuestas hasta que la conectividad móvil esté disponible. La capacidad de respuesta y la disponibilidad se han vuelto cruciales.

Aunque las conexiones a internet son de alta velocidad hoy en día, el tiempo de respuesta siempre es importante al proporcionar dicha información. Por ello, como veremos en esta sección, implementamos mecanismos para gestionar el estado de nuestras aplicaciones de forma transparente para el usuario final.

Pasando del infierno de las devoluciones de llamadas a las promesas

A veces, necesitamos crear funcionalidades en nuestra aplicación que cambien su estado de forma asíncrona una vez transcurrido el tiempo. En estos casos, debemos introducir patrones de código, como el patrón de devolución de llamada, para gestionar este cambio diferido en el estado de la aplicación.

En una devolución de llamada, la función que activa la acción asíncrona acepta otra función como parámetro. La función se ejecuta una vez completada la operación asíncrona.

Necesitará el código fuente de la aplicación Angular que creamos en el Capítulo 5, Administración de tareas complejas con servicios, para seguir con el resto del capítulo. Después de obtener el código, le sugerimos que realice las siguientes acciones para simplificar:



- Eliminar la carpeta de favoritos
- Elimine favorite.service.ts y su archivo de prueba unitaria
- Eliminar el archivo favorite.ts
- Elimine el archivo numeric.directive.ts y su archivo de prueba unitaria
- Eliminar la carpeta de vista del producto

Veamos cómo utilizar un callback a través de un ejemplo:

1. Abra el archivo app.component.html y agregue un elemento HTML <header> para mostrar el Propiedad del componente de título mediante interpolación:

```
<header>{{ título }}</header>
<main class="principal">
  <div class="contenido">
    <lista-de-productos-de-aplicaciones></lista-de-productos-de-aplicaciones>
  </div>
</principal>
<footer appCopyright> - v{{ settings.version }}</footer>
<enrutador-de-salida />
```

2. Abra el archivo app.component.ts y cree la siguiente propiedad:

```
conjunto privadoTítulo = () => {
  este.título = este.configuración.título;
}
```

La propiedad setTitle se utiliza para cambiar la propiedad del componente de título según la propiedad de título de la configuración de la aplicación. Devuelve una función de flecha porque... Úselo como una devolución de llamada a otro método.

3. A continuación, cree un método changeTitle que llame a otro método, llamado, por convención,

devolución de llamada, después de dos segundos:

```
cambio de título privado (devolución de llamada: Función) {
  establecerTiempo de espera() => {
    llamar de vuelta();
  }, 2000);
}
```

4. Agregue un constructor para llamar al método changeTitle , pasando la propiedad setTitle como un parámetro:

```
constructor() {
  este.changeTitle(este.setTitle);
```

En el fragmento anterior, utilizamos la propiedad setTitle sin paréntesis porque pasamos firmas de funciones y no llamadas de funciones reales cuando usamos devoluciones de llamadas.

Si ejecutamos la aplicación Angular con el comando ng serve , observamos que la propiedad title cambia después de dos segundos. El problema con el patrón que acabamos de describir es que el código puede volverse confuso y engoroso al introducir más devoluciones de llamada anidadas.

Consideremos el siguiente escenario en el que necesitamos explorar en profundidad una jerarquía de carpetas para acceder a las fotos en un dispositivo:

```
obtenerCarpetaRaíz(carpeta => {
  getAssetsFolder(carpeta, activos => {
    getPhotos(activos, fotos => {}); });
});
```

Dependemos de la llamada asíncrona anterior y de los datos que devuelve antes de poder realizar la siguiente llamada. Debemos ejecutar un método dentro de una devolución de llamada que ejecuta otro método con una devolución de llamada.

El código rápidamente parece complejo y difícil de leer, lo que conduce a una situación conocida como "infierno de devoluciones de llamadas".

Podemos evitar el caos de las devoluciones de llamadas mediante promesas. Las promesas introducen una nueva forma de concebir la gestión asíncrona de datos al adaptarse a una interfaz más limpia y sólida. Diferentes operaciones asíncronas pueden encadenarse al mismo nivel e incluso dividirse y devolverse desde otras funciones.

Para comprender mejor cómo funcionan las promesas, refactoricemos nuestro ejemplo de devolución de llamada anterior:

1. Cree un nuevo método en la clase AppComponent llamado onComplete que devuelva una Promesa Objeto. Una promesa puede resolverse o rechazarse. El parámetro resolve indica que la promesa se completó correctamente y, opcionalmente, devuelve un resultado:

```
privado onComplete() {
  devolver nueva Promesa<void>(resolver => {
});}
```

2. Introduzca un tiempo de espera de dos segundos en la promesa para que se resuelva después de este tiempo.

transcurrido:

```
privado onComplete() {
  devolver nueva Promesa<void>(resolver => {
    establecerTiempo de espera(() => {
      resolver();
    }, 2000);
});}
```

3. Ahora, reemplace la llamada `changeTitle` en el constructor con el método basado en promesa.

Para ejecutar un método que devuelve una promesa, invocamos el método y lo encadenamos con el entonces método:

```
constructor() {  
  este.onComplete().then(este.setTitle);  
}
```

No deberíamos notar ninguna diferencia significativa si volvemos a ejecutar la aplicación Angular. El verdadero valor de las promesas reside en la simplicidad y legibilidad que ofrecen a nuestro código. Ahora podríamos refactorizar el ejemplo anterior de jerarquía de carpetas en consecuencia:

```
obtenerCarpetaRaíz()  
.then(obtenerCarpetaDeActivos)  
.then(obtenerFotos);
```

El encadenamiento del método `then` en el código anterior muestra cómo podemos alinear una llamada asincrónica tras otra. Cada llamada asincrónica previa pasa su resultado al siguiente método asincrónico.

Las promesas son convincentes, pero a veces podríamos necesitar producir una salida de respuesta que siga un proceso de resumen más complejo o incluso cancelar todo el proceso. No podemos lograr tal comportamiento con promesas porque se activan tan pronto como se instancian. En otras palabras, las promesas no son perezosas. Por otro lado, la posibilidad de desmantelar una operación asincrónica después de que se haya disparado pero aún no se haya completado puede resultar bastante útil en escenarios específicos. Las promesas nos permiten resolver o rechazar una operación asincrónica, pero a veces podríamos querer abortar todo antes de llegar a ese punto.

Además, las promesas se comportan como operaciones puntuales. Una vez resueltas, no podemos esperar recibir más información ni notificaciones de cambio de estado a menos que ejecutemos todo desde cero. En resumen, las limitaciones de las promesas:

- No se pueden cancelar
- Se ejecutan inmediatamente
- Son operaciones únicas; no hay una manera fácil de volver a intentarlas.
- Responden con un solo valor

Ilustremos algunas de las limitaciones con un ejemplo:

1. Reemplace setTimeout con setInterval en el método onComplete :

```
privado onComplete() {  
    devolver nueva Promesa<void>(resolver => {  
        establecerIntervalo(() => {  
            resolver();  
        }, 2000);  
    });  
}
```

La promesa ahora se resolverá repetidamente cada dos segundos.

2. Modifique la propiedad setTitle para agregar la marca de tiempo actual en la propiedad de título de

El componente:

```
conjunto privadoTítulo = () => {  
    const timestamp = nueva Fecha();  
    este.título = `${este.configuración.título} (${marca de tiempo})`;  
}
```

3. Ejecute la aplicación Angular y observará que la marca de tiempo se establece solo una vez después de dos segundos y no vuelve a cambiar. La promesa se resuelve sola y el evento asincrónico finaliza en ese preciso instante.

Es posible que necesitemos una implementación más proactiva del manejo asincrónico de datos para corregir el comportamiento anterior, y aquí es donde los observables entran en escena.

Observables en pocas palabras

Un observable es un objeto que mantiene una lista de dependientes, llamados observadores, y les informa sobre cambios de estado y datos mediante la emisión de eventos asincrónicos. Para ello, el observable implementa todos los mecanismos necesarios para producir y emitir dichos eventos. Puede activarse y cancelarse en cualquier momento, independientemente de si ya ha emitido los datos esperados.

Los observadores deben suscribirse a un observable para recibir notificaciones y reaccionar para reflejar el cambio de estado. Este patrón, conocido como patrón de observador, permite operaciones concurrentes y una lógica más avanzada. Estos observadores, también conocidos como suscriptores, siguen escuchando lo que sucede en el observable hasta que se destruye. Podemos ver todo esto con mayor claridad en un ejemplo real:

1. Importe el artefacto Observable desde el paquete npm rxjs :

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

2. Cree una propiedad de componente llamada title\$ que cree un objeto Observable . El constructor de un observable acepta un objeto observador como parámetro. El observador es una función de flecha que contiene la lógica de negocio que se ejecutará cuando alguien use el observable. Llame al siguiente método del observador cada dos segundos para indicar un cambio en los datos o el estado de la aplicación:

```
título$ = nuevo Observable(observador => {
    establecerIntervalo(() => {
        observador.siguiente();
    }, 2000);
});
```



Al definir una variable observable, solemos añadir el símbolo \$ al nombre de la variable. Esta es una convención que seguimos para identificar observables en nuestro código de forma eficiente y rápida.

3. Modifique el componente constructor para utilizar la propiedad title\$ recién creada :

```
constructor() {
    este.title$.subscribe(este.setTitle);
}
```

Usamos el método de suscripción para registrarnos en el observable title\$ y recibir notificaciones de cualquier cambio. Si no llamamos a este método, el método setTitle nunca se ejecutará.



Un observable no hará nada a menos que un suscriptor se suscriba a él.

Si ejecutas la aplicación, notarás que la marca de tiempo cambia cada dos segundos. ¡Felicidades! ¡Has entrado al mundo de los observables y la programación reactiva!

Los observables devuelven un flujo de eventos, y nuestros suscriptores reciben notificaciones inmediatas de dichos eventos para que puedan actuar en consecuencia. No realizan una operación asíncrona ni finalizan (aunque podemos configurarlos para que lo hagan), sino que inician un flujo de eventos continuos para... al que podemos suscribirnos.

Pero eso no es todo. Este flujo puede combinar numerosas operaciones antes de llegar a los observadores suscritos. Así como podemos manipular arrays con métodos como map o filter para transformarlos, podemos hacer lo mismo con el flujo de eventos emitidos por observables. Se trata de un patrón conocido como programación reactiva, y Angular aprovecha al máximo este paradigma para gestionar la asincrónica. información.

Programación reactiva en Angular

El patrón observador es fundamental en la programación reactiva. La implementación más básica de un script reactivo abarca varios conceptos con los que debemos familiarizarnos:

- Un observable
- Un observador
- Una línea de tiempo
- Un flujo de acontecimientos
- Un conjunto de operadores componibles

Puede parecer abrumador, pero no lo es. El gran reto aquí es cambiar nuestra mentalidad y aprender a pensar de forma reactiva, que es el objetivo principal de esta sección.



La programación reactiva implica aplicar suscripciones asincrónicas y transformaciones. información a flujos observables de eventos.

Expliquemos esto con un ejemplo más descriptivo. Imaginemos un dispositivo de interacción, como un teclado. Este tiene teclas que el usuario presiona. Cada pulsación desencadena un evento específico, como "keyUp". Este evento contiene una amplia gama de metadatos, incluyendo, entre otros, el código numérico de la tecla específica que el usuario presionó en un momento dado. A medida que el usuario continúa presionando las teclas, se activan más eventos "keyUp" que se transmiten a través de una línea de tiempo imaginaria. Esta línea de tiempo es un flujo continuo de datos donde el evento "keyUp" puede ocurrir en cualquier momento; después de todo, el usuario decide cuándo presionar esas teclas.

Recuerde el ejemplo con observables de la sección anterior. Ese código podría notificar a un observador que cada dos segundos se emitía otro valor. Sabemos con qué frecuencia se activa un intervalo de temporizador. En el caso de los eventos keyUp, no lo sabemos porque no están bajo nuestro control.

Intentemos explicarlo más detalladamente implementando un key logger en nuestra aplicación:

1. Cree un nuevo componente Angular llamado key-logger:

```
ng generar registrador de teclas de componentes
```

2. Abra el archivo key-logger.component.html y reemplace su contenido con lo siguiente

Plantilla HTML:

```
< tipo de entrada="texto" #keyContainer />  
Presionaste : {{keys}}
```

En la plantilla anterior, agregamos un elemento HTML `<input>` y adjuntamos la variable de referencia de plantilla `keyContainer`.



Se puede agregar una variable de referencia de plantilla a cualquier elemento HTML, no solo componentes.

También mostramos una propiedad de teclas que representa todas las teclas del teclado que el usuario ha presionado.

3. Abra el archivo key-logger.component.ts e importe OnInit, ViewChild y

Artefactos ElementRef del paquete npm @angular/core :

```
importar { Componente, ElementRef, OnInit, ViewChild } desde '@angular/  
centro';
```

4. Cree las siguientes propiedades en la clase KeyLoggerComponent :

```
entrada = ViewChild<ElementRef>('keyContainer');  
claves = '';
```

La propiedad de entrada se utiliza para consultar el elemento HTML `<input>` mediante `keyContainer` variable de referencia de plantilla.

5. Agregue la siguiente declaración de importación para importar el artefacto fromEvent desde el npm rxjs paquete:

```
importar { fromEvent } de 'rxjs';
```

La biblioteca RxJS cuenta con varios artefactos útiles, llamados operadores, que podemos usar con observables. El operador fromEvent crea un observable a partir del evento DOM de un objeto nativo. Elemento HTML.

6. Implemente el método ngOnInit de la interfaz OnInit para escuchar eventos de pulsación de tecla en el elemento <input> y guardar las teclas presionadas en la propiedad keys :

```
clase de exportación KeyLoggerComponent implementa OnInit {
    entrada = ViewChild<ElementRef>('keyContainer'); claves = '';
}

ngOnInit(): vacío {
    const logger$ = fromEvent<KeyboardEvent>(this.input()!.nativeElement,
    'keyup');
    logger$.subscribe(evt => this.keys += evt.key);
}
}
```

Observe que accedemos al elemento de entrada HTML nativo mediante la propiedad nativeElement de la variable de referencia de plantilla. El resultado de la consulta con la función ViewChild es un objeto ElementRef , que envuelve el elemento HTML.

7. Abra el archivo app.component.ts e importe la clase KeyLoggerComponent :

```
importar { Componente, inyectar } desde '@angular/core'; importar
{ RouterOutlet } desde '@angular/router';
importar { ProductListComponent } de './product-list/product-list. component'; importar

{ CopyrightDirective } de './copyright.directive';
importar { APP_SETTINGS, appSettings } desde './app.settings'; importar
{ Observable } desde 'rxjs';
importar { KeyLoggerComponent } desde './key-logger/key-logger.component';

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

```

proveedores: [
  { proporcionar: APP_SETTINGS, valor de uso: appSettings }
]
})
}

```

8. Abra el archivo app.component.html y agregue el selector <app-key-logger> en la plantilla:

```

<header>{{ título }}</header>
<main class="principal">
  <div class="contenido">
    <lista-de-productos-de-aplicaciones></lista-de-productos-de-aplicaciones>
  </div>
</principal>
<footer appCopyright> - v{{ settings.version }}</footer>
<enrutador-de-salida />
<registrar de teclas de la aplicación></registrar de teclas de la aplicación>

```

Ejecute la aplicación usando el comando ng serve y comience a presionar teclas para verificar el uso del key logger que acabamos de crear:



angular You pressed: angular

Figura 6.1: Salida del registrador de teclas

Un aspecto esencial de los observables es el uso de operadores y su encadenamiento, lo que permite una composición rica. Los operadores observables se asemejan a métodos de array cuando se utilizan. Por ejemplo, un operador de mapa para observables se utiliza de forma similar al método de mapa de un array. En la siguiente sección, aprenderemos sobre la biblioteca RxJS, que proporciona estos operadores, y algunos de ellos se explican mediante ejemplos.

La biblioteca RxJS

Como se mencionó anteriormente, Angular viene con una dependencia de pares en RxJS, la versión de JavaScript de la biblioteca ReactiveX , que nos permite crear observables a partir de una gran variedad de escenarios, incluidos los siguientes:

- Eventos de interacción
- Promesas
- Funciones de devolución de llamada
- Eventos

La programación reactiva no tiene como objetivo reemplazar patrones asíncronos como promesas o devoluciones de llamadas.

En general, también puede aprovecharlos para crear secuencias observables.

RxJS incorpora compatibilidad con diversos operadores componibles para transformar, filtrar y combinar los flujos de eventos resultantes. Su API proporciona métodos prácticos para que los observadores se suscriban a estos flujos, de modo que nuestros componentes puedan responder según los cambios de estado o la interacción con la entrada. Veamos algunos de estos operadores en acción en las siguientes subsecciones.

Creando observables

Ya hemos aprendido cómo crear un observable a partir de un evento DOM usando fromEvent operador. Otros dos operadores populares relacionados con la creación de observables son of y from operadores.

El operador of se utiliza para crear un observable a partir de valores como números:

```
valores constantes = de(1, 2, 3);
valores.subscribe(valor => console.log(valor));
```

El fragmento anterior imprime los números 1, 2 y 3 en la ventana de la consola del navegador en orden.

El operador from se utiliza para convertir una matriz en un observable:

```
valores constantes = de([1, 2, 3]);
valores.subscribe(valor => console.log(valor));
```

El operador from también es muy útil al convertir promesas o devoluciones de llamada en observables. Podríamos encapsular el método onComplete en el constructor de la clase AppComponent de la siguiente manera:

```
constructor() {
  const complete$ = from(this.onComplete());
  complete$.subscribe(this.setTitle);
}
```



¡El operador from es una excelente manera de migrar a observables si usa promesas en una aplicación existente!

Además de crear observables, la biblioteca RxJS también contiene un par de operadores útiles para manipular y transformar datos emitidos desde los observables.

Transformando observables

Ya aprendimos a crear una directiva solo numérica en el Capítulo 4, Enriquecimiento de aplicaciones mediante tuberías y directivas. Ahora usaremos operadores RxJS para lograr lo mismo en nuestro componente de registrador de teclas:

1. Abra el archivo key-logger.component.ts e importe el operador tap desde el npm rxjs paquete:

```
importar { fromEvent, tap } desde 'rxjs';
```

2. Refactorice el método ngOnInit de la siguiente manera:

```
ngOnInit(): vacío {
  const logger$ = fromEvent<KeyboardEvent>(this.input()!.elemento nativo, 'keyup');
  registrador$.pipe(
    tap(evt => this.keys += evt.key)
  ).suscribir();
}
```

El operador de canalización enlaza y combina varios operadores separados por comas. Podemos considerarlo como una receta que define los operadores que deben aplicarse a un observable.

Uno de ellos es el operador tap , que se utiliza cuando queremos hacer algo con los datos emitidos sin modificarlos.

3. Queremos excluir los valores no numéricos que emite el observable logger\$. Ya obtenemos la tecla pulsada de la propiedad evt , pero devuelve valores alfanuméricos. No sería eficiente listar todos los valores no numéricos y excluirlos manualmente. En su lugar, usaremos el operador map para obtener el valor Unicode real de la clave. Su comportamiento es similar al del método map de un array, ya que devuelve un observable con una versión modificada de los datos iniciales. Importe el operador map desde el paquete npm rxjs :

```
importar { fromEvent, tap, map } de 'rxjs';
```

4. Agregue el siguiente fragmento encima del operador tap en el método ngOnInit :

```
mapa(evt => evt.key.charCodeAt(0))
```

5. Ahora podemos agregar el operador de filtro , que funciona de forma similar al método de filtro de un array para excluir valores no numéricos. Importe el operador de filtro desde rxjs.
paquete npm:

```
importar { fromEvent, tap, map, filter } de 'rxjs';
```

6. Agregue el siguiente fragmento después del operador de mapa en el método ngOnInit :

```
filtro(código => (código > 31 && (código < 48 || código > 57)) === falso)
```

7. El observable actualmente emite códigos de caracteres Unicode. Debemos convertirlos de nuevo a caracteres de teclado para mostrarlos en la plantilla HTML. Refactorice el operador tap para adaptarlo a este cambio:

```
tap(dígito => this.keys += String.fromCharCode(dígito))
```

Como toque final, agregaremos un enlace de entrada en el componente para activar y desactivar la función solo numérica de manera condicional:

1. Agregue la función de entrada en la declaración de importación del paquete npm @angular/core :

```
importar { Componente, ElementRef, OnInit, ViewChild, entrada } de '@ angular/core';
```

2. Agregue una propiedad de entrada numérica en la clase KeyLoggerComponent :

```
numérico = entrada(falso);
```

3. Refactorice el operador de filtro en el método ngOnInit para que tenga en cuenta la propiedad numérica :

```
filtro(código => {
    si (este.numeric()) {
        return (código > 31 && (código < 48 || código > 57)) === falso;
    }
    devuelve
    verdadero; })
```

El observable logger\$ filtrará valores no numéricos solo si la propiedad de entrada numérica es cierto

4. El método ngOnInit debería finalmente verse así:

```
ngOnInit(): vacío {
    const logger$ = fromEvent<KeyboardEvent>(this.input()!.
    elemento nativo, 'keyup');
    logger$.pipe(
        mapa(evt => evt.key.charCodeAt(0)),
        filtro(código => {
            si (este.numeric()) {
```

```

        return (código > 31 && (código < 48 || código > 57)) === falso;
    }
    devuelve verdadero;
}),
tap(dígito => this.keys += String.fromCharCode(dígito))
.suscribir();
}

```

5. Abra el archivo app.component.html y agregue un enlace a la propiedad numérica en el Selector <app-key-logger> :

```
<app-key-logger [numérico]="verdadero"></app-key-logger>
```

6. Ejecute la aplicación usando el comando ng serve e ingrese Angular 19 dentro de la entrada caja:



Figura 6.2: Registrador de teclas numéricas

Hemos visto operadores RxJS que manipulan observables que devuelven tipos de datos primitivos como números, cadenas y matrices. En la siguiente sección, aprenderemos a usar observables en nuestra aplicación de tienda online.

Suscribirse a observables

Ya hemos aprendido que un observador necesita suscribirse a un observable para obtener los datos emitidos. En nuestro caso, el observador será el componente de lista de productos y el observable residirá en el archivo products.service.ts . Por lo tanto, primero debemos convertir la clase ProductsService para que use observables en lugar de matrices simples, de modo que los componentes puedan suscribirse para obtener datos:

1. Abra el archivo products.service.ts y agregue la siguiente declaración de importación :

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

2. Extraiga los datos del producto utilizados en el método getProducts en una propiedad de servicio independiente

Para mejorar la legibilidad del código:

```

productos privados : Producto[] = [
{
    identificación: 1,
    Título: 'Teclado',
    precio: 100,
}
]

```

```

categorías: {
  1: 'Computación',
  2: 'Periféricos'
},
{
  id: 2,
  título: 'Micrófono',
  precio: 35,
  categorías: { 3: 'Multimedia' }
},
{
  id: 3,
  título: 'Cámara web',
  precio: 79,
  categorías: {
    1: 'Computación',
    3: 'Multimedia'
  }
},
{
  identificación: 4,
  Título: 'Tableta',
  precio: 500,
  categorías: { 4: 'Entretenimiento' }
}
];

```

3. Modifique el método `getProducts` para que devuelva la propiedad de productos como un observable:

```

obtenerProductos(): Observable<Producto[]> {
  retorno de(este.productos);
}

```

En el fragmento anterior, utilizamos el operador `of` para crear un nuevo observable a partir del gama de productos

La clase `ProductsService` ahora emite datos de producto mediante observables. Debemos modificar el componente para suscribirse y obtener estos datos:

1. Abra el archivo product-list.component.ts y cree un método getProducts en el Clase ProductListComponent :

```
privado obtenerProductos() {  
    este.productService.getProducts().subscribe(productos => {  
        este.productos = productos;  
    });  
}
```

En el método anterior, nos suscribimos al método getProducts del ProductsService Clase porque devuelve un observable en lugar de un array simple. El array de productos se devuelve dentro del método subscribe , donde asignamos la propiedad del componente de productos al array emitido por el observable.

2. Modifique el método ngOnInit para que llame al método getProducts recién creado :

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



Podríamos haber agregado el cuerpo del método getProducts dentro de ngOnInit Método directamente. No lo hicimos, ya que los métodos de eventos del ciclo de vida del componente deben ser lo más claros y concisos posible. Siempre intente extraer su lógica en un método separado para mayor claridad.

Ejecute la aplicación usando el comando ng serve y debería ver la lista de productos mostrada en la página correctamente:

Products (4)

Keyboard

Microphone

Tablet

Web camera

Figura 6.3: Lista de productos

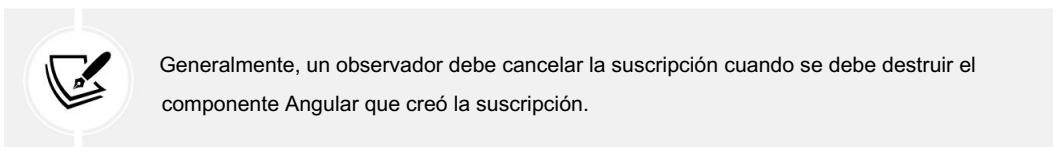
Como se muestra en la imagen anterior, hemos logrado el mismo resultado al mostrar la lista de productos que en el Capítulo 5, Gestión de Tareas Complejas con Servicios, pero utilizando observables. Puede que no sea evidente a primera vista, pero hemos sentado las bases para trabajar con el cliente HTTP de Angular, que se basa en observables. En el Capítulo 8, Comunicación con Servicios de Datos a través de HTTP, exploraremos...

El cliente HTTP con más detalle.

Al suscribirnos a observables, corremos el riesgo de sufrir fugas de memoria si no los limpiamos a tiempo. En la siguiente sección, aprenderemos diferentes maneras de lograrlo.

Darse de baja de observables

Cuando nos suscribimos a un observable, creamos un observador que escucha los cambios en un flujo de datos. El observador observa la transmisión continuamente mientras la suscripción permanece activa. Cuando una suscripción está activa, reserva memoria en el navegador y consume ciertos recursos. Si no le indicamos al observador que cancele su suscripción en algún momento y limpie los recursos, la suscripción al observable podría provocar una fuga de memoria.



Generalmente, un observador debe cancelar la suscripción cuando se debe destruir el componente Angular que creó la suscripción.

Algunas de las técnicas más conocidas para cancelar la suscripción a observables son las siguientes:

- Cancelar la suscripción a un observable manualmente
- Utilice la tubería asíncrona en una plantilla de componente

Veamos ambas técnicas en acción en las siguientes subsecciones.

Destrucción de un componente

Un componente tiene eventos de ciclo de vida que podemos usar para conectar y ejecutar lógica personalizada, como aprendimos en el Capítulo 3, "Estructura de interfaces de usuario con componentes". Uno de ellos es el evento `ngOnDestroy`, que se llama cuando el componente se destruye y deja de existir.

Recuerde `ProductListComponent` y `ProductViewComponent`, que usamos anteriormente en nuestros ejemplos. Se suscriben a los métodos correspondientes de `ProductsService` y `ProductViewService` al inicializar el componente. Cuando se destruyen los componentes, la referencia de las suscripciones permanece activa, lo que puede provocar un comportamiento impredecible. Es necesario cancelar la suscripción manualmente cuando se destruyen los componentes para limpiar los recursos correctamente.

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

```
importar { Suscripción } de 'rxjs';
```

2. Cree la siguiente propiedad en la clase ProductListComponent :

```
productos privadosSub : Suscripción | undefined;
```

3. Asigne la propiedad productsSub al resultado de la suscripción en el método getProducts :

```
privado obtenerProductos() {
    este.productsSub = este.productService.getProducts().
    suscribirse(productos =>
        { this.productos = productos; });

}
```

4. Importe el gancho del ciclo de vida OnDestroy desde el paquete npm @angular/core :

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

5. Agregue OnDestroy a la lista de interfaces implementadas de la clase ProductListComponent :

La clase de exportación ProductListComponent implementa OnInit, OnDestroy

6. Implemente el método ngOnDestroy de la siguiente manera:

```
ngOnDestroy(): void
{
    este.productsSub?.unsubscribe();
}
```

El método de cancelación de suscripción elimina un observador de los oyentes activos de una suscripción y limpia todos los recursos reservados.

Se necesita mucho código repetitivo para cancelar una sola suscripción. Si tenemos muchas suscripciones, podría volverse ilegible e inmantenible rápidamente.

Como alternativa, podemos usar un tipo particular de operador llamado takeUntilDestroyed, disponible en el paquete @angular/core/rxjs-interop . Exploraremos cómo cancelar la suscripción a observables usando este operador en el componente de lista de productos:

1. Abra el archivo product-list.component.ts e importe los artefactos inject, DestroyRef y takeUntilDestroyed de la siguiente manera:

```
importar { Componente, DestroyRef, inyectar, OnInit } desde '@angular/ core';
```

```
importar {takeUntilDestroyed} desde '@angular/core/rxjs-interop';
```

El artefacto takeUntilDestroyed es un operador que cancela la suscripción a un observable cuando se destruye el componente.

2. Declare la siguiente propiedad para injectar el servicio DestroyRef :

```
privado destroyRef = inyectar(DestroyRef);
```

3. Modifique el método getProducts de la siguiente manera:

```
privado obtenerProductos() {
    este.productService.getProducts().pipe(
        tomar hasta que se destruya (this.destroyRef)
    ).subscribe(productos => {
        este.productos = productos;
    });
}
```

En el método anterior, utilizamos el operador de tubería para encadenar el método takeUntilDestroyed.

Operador con la suscripción del método getProducts del ProductsService

clase. El operador takeUntilDestroyed acepta un parámetro del servicio DestroyRef .

4. Elimine cualquier código relacionado con el método ngOnDestroy .

¡Listo! Hemos convertido nuestra suscripción para que sea más declarativa y legible. Sin embargo, el problema de mantenimiento persiste. Nuestros componentes ahora cancelan la suscripción a sus observables manualmente.

Podemos solucionarlo usando una tubería Angular específica, la tubería asíncrona , que nos permite cancelar la suscripción automáticamente con menos código.

Usando la tubería asíncrona

La tubería asíncrona es una tubería integrada de Angular que se utiliza junto con observables y tiene una doble función : nos ayuda a escribir menos código y nos ahorra tener que configurar y desactivar una suscripción.

Se suscribe automáticamente a un observable y cancela la suscripción cuando se destruye el componente.

Lo usaremos para simplificar el código del componente de lista de productos:

1. Abra el archivo product-list.component.ts y agregue las siguientes declaraciones de importación :

```
importar { AsyncPipe } desde '@angular/common';
importar { Observable } de 'rxjs';
```

2. Agregue la clase AsyncPipe a la matriz de importaciones del decorador @Component :

```
@Component({
  selector: 'lista-de-productos-de-aplicaciones',
  imports: [ProductDetailComponent, SortPipe, AsyncPipe],
  templateUrl: './lista-de-productos.componente.html',
  styleUrls: ['./lista-de-productos.componente.css']
})
```

3. Convierte la propiedad del componente productos en un observable:

```
productos$: Observable<Product[]> | undefined;
```

4. Asigne el método getProducts de la clase ProductsService al componente products\$

propiedad nonte:

```
privado obtenerProductos() {
  este.productos$ = este.productoServicio.getProducts();
}
```

El cuerpo del método getProducts ahora se ha reducido a una línea y se ha vuelto más legible.

5. Abra el archivo product-list.component.html y agregue el siguiente fragmento al comienzo del archivo:

```
@let productos = (productos$ | async);
```

En el fragmento anterior, nos suscribimos al observable products\$ mediante la tubería async y creamos una variable de plantilla con la palabra clave @let . Esta variable de plantilla tiene el mismo nombre que la propiedad del componente correspondiente que teníamos previamente, por lo que no es necesario modificar la plantilla del componente.

¡Listo! Ya no necesitamos suscribirnos ni cancelar la suscripción al observable manualmente.

La tubería asíncrona se encarga de todo por nosotros.

Hemos aprendido que los observables reaccionan a los eventos de la aplicación y emiten valores asíncrónicamente en observadores registrados. Podríamos visualizar los observables como objetos contenedores de los valores emitidos. Angular enriquece el campo de reactividad de las aplicaciones web al proporcionar un contenedor similar que funciona asíncrónicamente y reacciona a los cambios de estado de la aplicación.

Resumen

Se necesita mucho más que un solo capítulo para cubrir en detalle todas las ventajas que podemos lograr con la reactividad en Angular. La buena noticia es que hemos cubierto todas las herramientas y clases necesarias para el desarrollo básico en Angular.

Aprendimos qué es la programación reactiva y cómo se puede usar en Angular. Vimos cómo aplicar técnicas reactivas, como los observables, para interactuar con flujos de datos. Exploramos la biblioteca RxJS y cómo usar algunos operadores para manipular observables. Aprendimos diferentes maneras de suscribirse y cancelar la suscripción a observables en componentes de Angular.

El resto queda a tu imaginación, así que siéntete libre de ir más allá y poner en práctica todo este conocimiento en tus aplicaciones Angular. Las posibilidades son infinitas, y tienes estrategias que abarcan desde promesas hasta observables. Puedes aprovechar las increíbles funcionalidades de los patrones reactivos y crear experiencias reactivas increíbles para tus aplicaciones Angular.

Como ya hemos destacado, el cielo es el límite. Sin embargo, aún nos queda un largo y emocionante camino por delante. En el próximo capítulo, exploraremos las señales, un patrón reactivo alternativo integrado en el framework Angular. Aprenderemos a usar las señales Angular para gestionar el estado de una aplicación Angular.

7

Seguimiento del estado de la aplicación con Señales

Angular permite a los desarrolladores usar la reactividad integrada en sus aplicaciones mediante señales.

Las señales Angular son un enfoque síncrono para la programación reactiva que mejora eficientemente el rendimiento de las aplicaciones y gestiona su estado.

En capítulos anteriores, vimos señales, donde usamos el método de entrada para intercambiar datos entre componentes y el método viewChild para consultar los componentes secundarios. La API de señales se puede usar en diferentes partes de una aplicación Angular, por lo que su uso se encuentra disperso en los capítulos de este libro.

En este capítulo cubriremos los siguientes temas:

- Comprensión de señales
- Lectura y escritura de señales
- Señales calculadas
- Cooperando con RxJS

Requisitos técnicos

El capítulo contiene varios ejemplos de código para guiarlo a través del concepto de señales angulares.

Puede encontrar el código fuente relacionado en la carpeta ch07 del siguiente repositorio de GitHub:

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

Entendiendo las señales

Como aprendimos en el Capítulo 3, "Estructura de Interfaces de Usuario con Componentes", Zone.js desempeña un papel fundamental en el rendimiento de una aplicación Angular. Activa el mecanismo de detección de cambios de Angular cuando ocurren eventos específicos dentro de la aplicación. El framework verifica cada componente de la aplicación en cada ciclo de detección y evalúa sus enlaces, lo que reduce el rendimiento de la aplicación.

La razón de ser de la detección de cambios con Zone.js reside en que Angular no puede saber cuándo ni dónde se ha producido un cambio dentro de la aplicación. Inevitablemente, los desarrolladores de Angular intentan limitar los ciclos de detección de cambios mediante las siguientes técnicas:

- Configuración de componentes con la estrategia de detección de cambios OnPush
- Interactuar manualmente con el mecanismo de detección de cambios utilizando ChangeDetectorRef servicio

Las señales mejoran la forma en que los desarrolladores interactúan con el mecanismo de detección de cambios Angular al simplificar y mejorar las técnicas anteriores según las necesidades de la aplicación.

Las señales angulares proporcionan una gestión más robusta y ergonómica del ciclo de detección de cambios basada en la reactividad. Monitorean cómo cambia el estado de la aplicación y permiten que el framework reaccione activando la detección de cambios solo en las partes afectadas por el cambio.



Las señales son una característica innovadora del marco Angular que permitirá mejoras adicionales en el rendimiento de la aplicación al introducir aplicaciones sin zonas y componentes basados en señales en el futuro.

Las señales también actúan como contenedores de valores, que el mecanismo de detección de cambios debe verificar. Cuando un valor cambia, las señales notifican al framework sobre dicho cambio. El framework es responsable de activar la detección de cambios y actualizar a los consumidores de señales. El valor de una señal puede cambiar directamente mediante señales de escritura o indirectamente mediante señales de solo lectura o computadas .

En la siguiente sección, aprenderemos cómo funcionan las señales escribibles.

Lectura y escritura de señales

Una señal escribible se indica mediante el tipo de señal del paquete npm @angular/core .



Necesitará el código fuente de la aplicación Angular que creamos en el Capítulo 6, Patrones Reactivos en Angular, para continuar con el resto del capítulo. Después de obtener el código, le sugerimos eliminar la carpeta key-logger para mayor simplicidad.

Comencemos y aprendamos cómo podemos escribir un valor en una señal:

1. Abra el archivo app.component.ts e importe el artefacto de señal desde @angular/core paquete npm:

```
importar { Componente, inyectar, señal } de '@angular/core';
```

2. Declare la siguiente propiedad en la clase AppComponent como una señal e inicialícela:

```
fechaActual = señal(nueva Fecha());
```

3. Reemplace la variable de marca de tiempo en la propiedad setTitle con el siguiente fragmento:

```
this.currentDate.set(nueva Fecha());
```

En el fragmento anterior, usamos el método `set` para escribir un nuevo valor en la señal. Este método notifica al framework Angular que el valor ha cambiado y debe ejecutar el mecanismo de detección de cambios.

4. Modifique la propiedad del título para utilizar el valor de la señal currentDate :

```
este.título = `${este.configuración.título} (${este.fechaActual()})`;
```

En el fragmento anterior, llamamos al método getter currentDate para leer el valor de la señal.

Las señales son una excelente opción en los casos donde la velocidad y el rendimiento de una aplicación son importantes. como:

- Una página de panel con widgets y datos en vivo que deben actualizarse periódicamente, como un Solicitud de bolsa de valores.
- Un componente que necesita mostrar propiedades de un objeto grande o complejo, como el siguiente:

```
orden constante = {
  no: '1',
  fecha: nueva Fecha(),
  productos: [
    {
      identificación: 1,
      Título: 'Teclado',
      precio: 100
    },
  ],
};
```

```
{
  identificación: 2,
  Título: 'Micrófono',
  precio: 35
}
],
Código de cliente: '0002',
isCompleted: false
};
```

En este caso, podemos extraer las propiedades del objeto que queremos en una señal sin involucrar a todo el objeto en el ciclo de detección de cambios, como por ejemplo:

```
constante orderDetails = señal({
  no: '1',
  Código de cliente: '0002',
  isCompleted: false
});
```

Un método similar de señales que también activa la detección de cambios es el método de actualización . Se utiliza cuando queremos establecer un nuevo valor en una señal basándose en su valor actual:

```
esta.fechaActual.actualizar(d => {
  devuelve nueva Fecha(d.getFullYear(), d.getMonth(), d.getDate(), 0, 0);
});
```

El fragmento anterior obtendrá el valor de la señal currentDate en la variable d y lo usará para devolver un nuevo objeto Date .

En la siguiente sección, exploraremos cómo se comportan las señales calculadas en una aplicación Angular.

Señales calculadas

Una señal calculada o de solo lectura depende de otras señales, ya sean de escritura o calculadas. El valor de una señal calculada no puede cambiar directamente mediante el método de configuración o actualización ; solo puede cambiar indirectamente cuando cambia el valor de cualquiera de las otras señales.

Veamos cómo funciona:

1. Abra el archivo app.component.ts e importe los artefactos calculados y de señal desde el Paquete npm de @angular/core :

```
importar {  
    Componente,  
    injectar,  
    Señal,  
    calculado,  
    señal  
} de '@angular/core';
```

2. Cambie el tipo de propiedad del componente de título a Señal:

```
título: Señal<string> = señal(");
```

El tipo de señal indica que la señal es calculada.

3. Elimine la asignación de título del método setTitle y agréguela dentro del constructor como sigue:

```
constructor()  
{ este.título$.suscribirse(este.setTitle); este.título  
= calculado(() => {  
    devuelve `${this.settings.title} (${this.currentDate()})`; });  
}
```

En el fragmento anterior, utilizamos la función calculada para establecer el valor de la señal del título .

El valor de la señal de título depende de la señal currentDate . Se actualiza cada 2 segundos cuando cambia el valor de la señal currentDate .

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

```
<header>{{ título() }}</header>
```

5. Ejecute la aplicación usando ng serve y verifique que el título se actualice correctamente.

Las señales calculadas tienen un gran rendimiento cuando se trata de cálculos más complicados que el anterior debido a las siguientes razones:

- La función calculada se ejecuta cuando el valor de la señal se lee por primera vez en la plantilla
- Se calcula un nuevo valor de señal solo cuando cambian las señales derivadas
- Las señales calculadas utilizan un mecanismo de caché para memorizar valores y devolverlos sin recalculando

Aunque las señales son un enfoque reactivo moderno para Angular, son relativamente nuevas en el ecosistema Angular en comparación con RxJS. En la siguiente sección, aprenderemos cómo pueden cooperar con RxJS en una aplicación Angular.

Cooperando con RxJS

Signals y RxJS dotan a las aplicaciones Angular de capacidades reactivas. Estas bibliotecas se complementan para ofrecer reactividad y aprovechar al máximo las ventajas del framework Angular.

Signals no fue creado para reemplazar a RxJS, sino para proporcionar un enfoque reactivo alternativo a los desarrolladores con las siguientes características adicionales:

- Reactividad de grano fino
- Programación imperativa
- Uso mejorado del mecanismo de detección de cambios

Sin embargo, existen componentes esenciales del framework Angular que aún utilizan RxJS y observables, como el cliente HTTP y el enrutador. Además, muchos desarrolladores prefieren el enfoque declarativo que la biblioteca RxJS ofrece de fábrica.



Al momento de escribir este artículo, el equipo de Angular está investigando y experimentando para que RxJS sea opcional para las aplicaciones Angular en un futuro próximo. También están trabajando para convertir las API integradas, como el cliente HTTP y el enrutador, en señales.

Angular Signals proporciona una API integrada para interactuar con RxJS y observables. La API de señales proporciona una función que convierte un observable en una señal:

1. Abra el archivo product-list.component.ts e importe los artefactos inject y toSignal :

```
importar { Component, injectar } desde '@angular/core';
importar { toSignal } desde '@angular/core/rxjs-interop';
```

El paquete npm @angular/core/rxjs-interop incluye todos los métodos de utilidad para gestionar la cooperación entre señales y observables. La función toSignal puede convertir un observable en una señal.



El paquete rxjs-interop también contiene métodos de utilidad para convertir una señal en un observable. Puede leer más en " Patrones reactivos con RxJS y señales angulares" de Lamis Chebbi (Packt Publishing).

2. Cree la siguiente señal en la clase ProductListComponent :

```
productos = toSignal(inyectar(ProductosServicio).obtenerProductos(), {
    valor inicial: []
});
```

Pasamos dos parámetros a la función toSignal : el observable que queremos convertir y, opcionalmente, un valor inicial. En este caso, pasamos el método getProducts de la clase ProductService , que devuelve un observable, y también establecemos el valor inicial de la señal en un array vacío.

3. Abra el archivo product-list.component.html y modifique su contenido de la siguiente manera:

```
@if (productos().longitud > 0) {
    Productos ({{products().length}})
}

<ul class="grupo de píldoras">
    @for (producto de productos() | sort; seguimiento producto.id) {
        <li class="pill" (clic)="productoSeleccionado = producto">
            @switch (producto.título) {
                @case ('Teclado') { @case ☖ }
                ('Micrófono') { @default { 🎤 }
                    🔍 }
                }
                {{producto.título}}
            </li>
        } @vacío {
            <p>¡No se encontraron productos!</p>
        }
    </ul>

<detalle-del-producto-de-la-aplicación
    [producto]="ProductoSeleccionado"
    (añadido)="onAdded()">
</detalle-del-producto-de-la-aplicación>
```

En la plantilla anterior, eliminamos el bloque @if superior y convertimos los productos propiedad en una señal. No necesitamos la tubería asíncrona porque las señales se suscriben automáticamente a un observable.

4. Para depurar aún más nuestro componente, podemos eliminar cualquier código relacionado con la tubería asíncrona y los observables, ya que ya no son necesarios. El archivo product-list.component.ts resultante debería ser el siguiente:

```
importar { Componente, inyectar } de '@angular/core'; importar
{ toSignal } de '@angular/core/rxjs-interop'; importar { Producto } de '../
product';
importar { ProductDetailComponent } de './product-detail/product-detail.component';
importar { SortPipe } de
'./sort.pipe'; importar { ProductsService } de './
products.service';

@Component({ selector: 'lista-de-productos-de-la-aplicación',
importaciones: [ProductDetailComponent, SortPipe],
templateUrl: './product-list.component.html',
styleUrl: './product-list.component.css' })

clase de exportación ProductListComponent {
    selectedProduct: Producto | indefinido;
    productos = toSignal(inyectar(ProductosServicio).obtenerProductos(), {
        valor inicial: []
    });

    al agregarlo() {
        alert(` ${this.selectedProduct?.title} | agregado al carrito!`);
    }
}
```

5. Ejecute la aplicación usando ng serve y observe que la salida de la aplicación muestra la lista de productos.

El fragmento anterior parece mucho más sencillo. Las señales angulares mejoran la experiencia del desarrollador y la ergonomía, además del rendimiento de nuestras aplicaciones.

Resumen

En este capítulo, exploramos las señales, un nuevo patrón reactivo en Angular que se utiliza para gestionar el estado de la aplicación. Aprendimos su fundamento y cómo se comparan con Zone.js.

Exploramos ejemplos de cómo leer y escribir valores en señales. También aprendimos a crear señales computacionales que dependen de valores de otras señales.

En el próximo capítulo, aprenderemos cómo usar el cliente HTTP Angular y consumir datos desde un punto final remoto.

8

Comunicarse con datos Servicios a través de HTTP

Un escenario real para las aplicaciones Angular empresariales es la conexión a servicios y API remotos para intercambiar datos. El cliente HTTP de Angular ofrece compatibilidad inmediata para la comunicación con servicios a través de HTTP. La interacción de una aplicación Angular con el cliente HTTP se basa en flujos observables de RxJS, lo que ofrece a los desarrolladores un amplio conjunto de funciones para el acceso a los datos.

Hay muchas maneras de conectarse a las API mediante HTTP. En este libro, solo cubriremos algunos aspectos. Sin embargo, los conocimientos de este capítulo le brindarán todo lo necesario para conectar sus aplicaciones Angular a servicios HTTP rápidamente, dejando todo lo que pueda hacer con ellas a su propia creatividad.

En este capítulo, exploraremos los siguientes conceptos:

- Comunicación de datos a través de HTTP
- Presentación del cliente HTTP Angular
- Configuración de una API de backend
- Manejo de datos CRUD en Angular
- Autenticación y autorización con HTTP

Requisitos técnicos

Este capítulo contiene varios ejemplos de código que explican el concepto del cliente HTTP de Angular. Puede encontrar el código fuente relacionado en la carpeta ch08 del siguiente repositorio de GitHub:

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

Comunicación de datos a través de HTTP

Antes de profundizar en la descripción del cliente HTTP de Angular y cómo usarlo para comunicarse con servidores, hablemos primero de las implementaciones HTTP nativas. Actualmente, si queremos comunicarnos con un servidor a través de HTTP usando JavaScript, podemos usar la API de búsqueda nativa de JavaScript . Esta contiene todos los métodos necesarios para conectar con un servidor e intercambiar datos.

Puedes ver un ejemplo de cómo obtener datos en el siguiente código:

```
obtener(url)
  .then(respuesta => {
    devolver respuesta.ok ? respuesta.texto() : '';
  })
  .then(resultado => {
    si (resultado) {
      console.log(resultado);
    } demás {
      console.error(' Se ha producido un error');
    }
  });
});
```

Aunque la API de búsqueda se basa en promesas, la promesa que devuelve no se rechaza si hay un error. En cambio, la solicitud no tiene éxito cuando la propiedad ok no está en el objeto de respuesta .

Si se completa la solicitud a la URL remota, podemos utilizar el método text() de la respuesta Objeto para devolver el texto de respuesta dentro de una nueva promesa. Finalmente, en la segunda devolución de llamada "then" , mostramos el texto de respuesta o un mensaje de error específico en la consola del navegador.



Para obtener más información sobre la API de búsqueda , consulte la documentación oficial en <https://desarrollador.mozilla.org/docs/Web/API/fetch>.

Ya hemos aprendido que los observables son flexibles para gestionar operaciones asincrónicas. Probablemente se esté preguntando cómo podemos aplicar este patrón al consumir información de un servicio HTTP. Hasta ahora, se estará acostumbrando a enviar solicitudes asíncronas a servicios AJAX y luego pasar la respuesta a una devolución de llamada o una promesa. Ahora, gestionaremos la llamada devolviendo un observable. El observable emitirá la respuesta del servidor como un evento en el contexto de un flujo, que puede canalizarse mediante operadores RxJS para procesar mejor la respuesta.

Convertamos el ejemplo anterior con la API de búsqueda en un observable. Usamos el Observable. clase para envolver la llamada de búsqueda en un flujo observable y reemplazar los métodos de consola con los métodos del objeto observador apropiados :

```
constante solicitud$ = nuevo Observable(observador => {
    obtener(url)
        .then(respuesta => {
            devolver respuesta.ok ? respuesta.texto() : '';
        })
        .then(resultado => {
            si (resultado) {
                observador.siguiente(resultado);
                observador.complete();
            } demás {
                observer.error('Se ha producido un error');
            }
        });
});
```

En el fragmento anterior, utilizamos los siguientes métodos de observador :

- siguiente: Esto devuelve los datos de respuesta a los suscriptores cuando llegan
- completo: Esto notifica a los suscriptores que no habrá otros datos disponibles en la transmisión
- error: Esto alerta a los suscriptores de que se ha producido un error.

¡Listo! Hemos creado un cliente HTTP personalizado. Claro que no es mucho. Nuestro cliente HTTP personalizado solo maneja una operación GET para obtener datos de un punto final remoto. No manejamos muchas otras operaciones del protocolo HTTP, como POST, PUT y DELETE. Sin embargo, era esencial comprender todo el trabajo pesado que el cliente HTTP en Angular hace por nosotros. Otra lección importante es lo fácil que es convertir una API asíncrona en una API observable que se integra perfectamente con el resto de nuestros conceptos asíncronos. Así que, continuemos con la implementación de un servicio HTTP en Angular.

Presentamos el cliente HTTP Angular

El cliente HTTP del framework Angular es una biblioteca independiente de Angular que reside en el paquete npm @angular/common, dentro del espacio de nombres http . La CLI de Angular instala este paquete por defecto al crear un nuevo proyecto Angular.



Necesitará el código fuente de la aplicación Angular que creamos en el Capítulo 6, Patrones Reactivos en Angular, para continuar con el resto del capítulo. Después de obtener el código, le sugerimos eliminar la carpeta key-logger para mayor simplicidad.

Para comenzar a utilizar el cliente HTTP Angular, necesitamos importar el método provideHttpClient en el archivo app.config.ts :

```
importar { provideHttpClient } desde '@angular/common/http';
importar { ApplicationConfig, provideZoneChangeDetection } desde '@angular/
centro';
importar { provideRouter } desde '@angular/router';

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

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



Supongamos que queremos usar el cliente HTTP en aplicaciones creadas con versiones anteriores de Angular . En ese caso, necesitamos importar un módulo de Angular, llamado HttpClientModule, desde el espacio de nombres @angular/common/http a uno de los módulos de nuestra aplicación .

El método provideHttpClient expone varios servicios de Angular que podemos usar para gestionar la comunicación HTTP asíncrona. El más básico es el servicio HttpClient , que proporciona una API robusta y abstrae todas las operaciones necesarias para gestionar conexiones asíncronas mediante los siguientes métodos HTTP:

- obtener: Esto realiza una operación GET para obtener datos
- post: Esto realiza una operación POST para agregar nuevos datos
- put/patch: Realiza una operación PUT/PATCH para actualizar datos existentes
- eliminar: realiza una operación ELIMINAR para eliminar datos existentes

Los métodos HTTP anteriores constituyen las operaciones principales para crear, leer, actualizar y eliminar.

Aplicaciones CRUD . Todos los métodos anteriores del cliente HTTP de Angular devuelven un flujo de datos observable. Los componentes de Angular pueden usar la biblioteca RxJS para suscribirse a estos métodos e interactuar con una API remota.



El equipo de Angular está investigando y experimentando para ver si es posible hacer que el uso de RxJS sea opcional en el framework. En ese caso, podríamos ver una implementación HTTP basada en señales. En el resto de este capítulo, nos centraremos en los observables, ya que el cliente HTTP de Angular no admite señales de fábrica.

En la siguiente sección, exploraremos cómo utilizar estos métodos y comunicarnos con un API remota.

Configuración de una API de backend

Una aplicación web CRUD suele conectarse a un servidor y utiliza una API HTTP de backend para realizar operaciones con los datos. Obtiene datos existentes, los actualiza, crea nuevos datos o los elimina.

En un escenario real, lo más probable es que interactúes con un servicio API backend real mediante HTTP. En este libro, usaremos una API falsa llamada Fake Store API.



La documentación oficial de la API de Fake Store se puede encontrar en <https://fakestoreapi.com>.

La API de Tienda Falsa es una API REST de backend disponible en línea que puede usar cuando necesita datos falsos para una aplicación web de comercio electrónico o tienda virtual. Permite gestionar productos, carritos de compra y usuarios disponibles en formato JSON. Expone los siguientes endpoints principales:

- **productos:** Gestiona un conjunto de artículos de productos
- **carrito:** Gestiona el carrito de compras de un usuario
- **usuario:** administra una colección de usuarios de la aplicación
- **inicio de sesión:** Esto maneja la autenticación del usuario



En este capítulo, trabajaremos únicamente con los productos y los puntos de acceso. Sin embargo, volveremos a abordar el punto de acceso del carrito más adelante.

Todas las operaciones que modifican datos no los conservan físicamente en una base de datos. Sin embargo, devuelven una indicación de si la operación se realizó correctamente. Todas las operaciones que obtienen datos devuelven un conjunto predefinido de elementos.

Manejo de datos CRUD en Angular

Las aplicaciones CRUD se utilizan ampliamente en el mundo Angular. Es difícil encontrar una aplicación web que no siga este patrón. Angular ofrece un excelente soporte para este tipo de aplicaciones al proporcionar el servicio HttpClient . En esta sección, exploraremos el cliente HTTP de Angular interactuando con el endpoint de productos de la API de Fake Store.

Obteniendo datos a través de HTTP

La clase ProductListComponent utiliza la clase ProductsService para obtener y mostrar datos de productos. Actualmente, los datos están codificados en la propiedad products de la clase ProductsService . En esta sección, modificaremos nuestra aplicación Angular para que funcione con datos en tiempo real de la API de Fake Store:

1. Abra el archivo app.component.ts y elimine la propiedad de proveedores de @Component Decorador. Proporcionaremos APP_SETTINGS directamente a través del archivo de configuración de la aplicación.

2. En este punto, también podemos eliminar la propiedad del título , el observable title\$, el setTitle propiedad y el constructor de la clase del componente:

```
clase de exportación AppComponent {  
    configuraciones = inyectar(APP_SETTINGS);  
}
```

3. Abra el archivo app.component.html y modifique el elemento HTML <header> para que utilice el objeto de configuración directamente:

```
<header>{{ configuración.título }}</header>
```

4. Abra el archivo app.config.ts y agregue el proveedor APP_SETTINGS de la siguiente manera:

```
importar { provideHttpClient } desde '@angular/common/http';  
importar { ApplicationConfig, provideZoneChangeDetection } desde '@  
angular/núcleo';  
importar { provideRouter } desde '@angular/router';  
  
importar { rutas } desde './app.routes';
```

```
importar { APP_SETTINGS, appSettings } desde './app.settings';

exportar const AppConfig: ApplicationConfig = { proveedores:
  [
    proporcionarZoneChangeDetection({ eventCoalescing: true }),
    proporcionarRouter(rutas),
    provideHttpClient(),
    { proporcionar: APP_SETTINGS, useValue: appSettings }
  ]
};
```

Proporcionamos APP_SETTINGS desde el archivo de configuración de la aplicación porque queremos que sea accesible globalmente en la aplicación.

5. Abra el archivo app.settings.ts y agregue una nueva propiedad en la interfaz AppSettings que Representa la URL de la API de Fake Store:

```
importar { InjectionToken } desde '@angular/core';

interfaz de exportación AppSettings {
  título: cadena;
  versión: cadena;
  apiUrl: cadena;
}

export const appSettings: AppSettings = { title: 'Mi
tienda online',
versión: '1.0', apiUrl:
'https://fakestoreapi.com'
};

exportar const APP_SETTINGS = nuevo InjectionToken<AppSettings>('app. settings');
```



La URL de una API de backend también se puede agregar en los archivos de entorno, como aprenderemos en el Capítulo 14, Llevar aplicaciones a producción.

6. Abra el archivo products.service.ts y modifique las declaraciones de importación según corresponda:

```
importar { HttpClient } desde '@angular/common/http';
importar { Inyectable, inyectar } desde '@angular/core';
importar { Producto } desde './producto';
importar { Observable, de } de 'rxjs';
importar { APP_SETTINGS } desde './app.settings';
```

7. Cree la siguiente propiedad en la clase ProductsService que representa el producto API.

punto final de productos:

```
productos privadosUrl = inyectar(APP_SETTINGS).apiUrl + '/productos';
```

8. Modifique el constructor para inyectar el servicio HttpClient :

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

9. Modifique el método getProducts para que utilice el servicio HttpClient para obtener la lista de productos:

```
obtenerProductos(): Observable<Producto[]> {
    devuelve este.http.get<Product[]>(this.productsUrl);
}
```

En el método anterior, utilizamos el método get de la clase HttpClient y pasamos el endpoint de productos de la API como parámetro. También definimos el producto como un tipo genérico en el método get para indicar que la respuesta de la API contiene una lista de objetos de producto .

10. Convierta la propiedad de productos en una matriz vacía:

```
productos privados : Producto[] = [];
```

Usaremos esto para propósitos de caché local más adelante, en la sección Modificación de datos a través de HTTP .

11. Abra el archivo product-list.component.html y modifique el bloque @if para que marque

Si existe la variable de plantilla de productos :

```
@if (productos) {
    Productos ({{products.length}})
}
```

Necesitamos verificar si la variable existe porque los datos ahora se obtienen del almacén falso API y habrá un retraso en la red antes de que la variable tenga un valor.

Si ejecutamos la aplicación usando el comando `ng serve` , deberíamos ver una lista extendida de productos de la API similar a la siguiente:

Products (20)

- 👉 Acer SB220Q bi 21.5 inches Full HD (1920 x 1080) IPS Ultra-Thin
- 👉 BIYLACLESEN Women's 3-in-1 Snowboard Jacket Winter Coats
- 👉 DANVOUY Womens T Shirt Casual Cotton Short
- 👉 Fjallraven - Foldsack No. 1 Backpack, Fits 15 Laptops
- 👉 John Hardy Women's Legends Naga Gold & Silver Dragon Station Chain Bracelet
- 👉 Lock and Love Women's Removable Hooded Faux Leather Moto Biker Jacket
- 👉 MBJ Women's Solid Short Sleeve Boat Neck V
- 👉 Mens Casual Premium Slim Fit T-Shirts
- 👉 Mens Casual Slim Fit
- 👉 Mens Cotton Jacket
- 👉 Opna Women's Short Sleeve Moisture
- 👉 Pierced Owl Rose Gold Plated Stainless Steel Double
- 👉 Rain Jacket Women Windbreaker Striped Climbing Raincoats
- 👉 Samsung 49-Inch CHG90 144Hz Curved Gaming Monitor (LC49HG90DMNXZA) – Super Ultrawide Screen QLED
- 👉 SanDisk SSD PLUS 1TB Internal SSD - SATA III 6 Gb/s
- 👉 Silicon Power 256GB SSD 3D NAND A55 SLC Cache Performance Boost SATA III 2.5
- 👉 Solid Gold Petite Micropave

Figura 8.1: Lista de productos de la API de Fake Store

El punto final de productos permite pasar un parámetro de solicitud para limitar los resultados devueltos por la API. Como se indica en <https://fakestoreapi.com/docs#p-limit>, podemos usar un parámetro de consulta llamado límite para lograr esta tarea. Veamos cómo podemos pasar parámetros de consulta en el cliente HTTP de Angular:

1. Abra el archivo products.service.ts e importe la clase HttpParams desde @angular/http :

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

La clase HttpParams se utiliza para pasar parámetros de consulta en una solicitud HTTP.

2. Cree la siguiente variable dentro del método getProducts :

```
const opciones = new HttpParams().set('limit', 10);
```

 La clase HttpParams es inmutable. Lo siguiente no funcionaría porque cada operación devuelve una nueva instancia:

```
const opciones = nuevos HttpParams();
opciones.set('limit', 10);
```

El método set de la clase HttpParams crea un nuevo parámetro de consulta. Si quisieramos pasar parámetros adicionales, deberíamos encadenar más métodos set , como:

```
opciones constantes = nuevos HttpParams()
.set('límite', 10)
.set('pagina', 1);
```

3. Usamos el segundo parámetro del método get para pasar parámetros de consulta usando el propiedad params :

```
devuelve este.http.get<Product[]>(este.productsUrl, {
    parámetros: opciones
});
```

4. Guarde los cambios, espere a que la aplicación se vuelva a cargar y observe el resultado de la aplicación:

Products (10)

- 👉 Fjallraven - Foldsack No. 1 Backpack, Fits 15 Laptops
- 👉 John Hardy Women's Legends Naga Gold & Silver Dragon Station Chain Bracelet
- 👉 Mens Casual Premium Slim Fit T-Shirts
- 👉 Mens Casual Slim Fit
- 👉 Mens Cotton Jacket
- 👉 Pierced Owl Rose Gold Plated Stainless Steel Double
- 👉 SanDisk SSD PLUS 1TB Internal SSD - SATA III 6 Gb/s
- 👉 Solid Gold Petite Micropave
- 👉 WD 2TB Elements Portable External Hard Drive - USB 3.0
- 👉 White Gold Plated Princess

Figura 8.2: Lista de productos

En la lista anterior, todos los productos se muestran con el mismo ícono de etiqueta, que es el predeterminado según el bloque @switch en el archivo product-list.component.html :

```
<li class="pill" (clic)="productoSeleccionado = producto">  
  @switch (producto.título) {  
    @case ('Teclado')  }  
    { @case ('Micrófono')  }  
    { @default {  }  
  }  
  {{producto.título}}  
</li>
```

El bloque @switch se basa en la propiedad del título del producto . Lo modificaremos para que se base en la propiedad de categoría , que proviene del punto final de productos de la API.

5. Abra el archivo product.ts y reemplace la propiedad categorías con la siguiente propiedad:

categoría: cadena;

6. Abra el archivo product-list.component.html y modifique el bloque @switch de la siguiente manera:

```
@switch (producto.categoría) {  
    @case ('electrónica') { @case  }  
    ('joyería') {  }  
    @por defecto {  }  
}
```

7. También necesitamos modificar el archivo product-detail.component.html porque reemplazamos la propiedad categorías en el paso 1:

```
@if (producto()) {  
    <p>Usted seleccionó:  
        <strong>{{producto()!.título}}</strong>  
    </p>  
    <p>{{producto()!.precio | moneda:'EUR'}}</p>  
    <div class="grupo-de-píldoras">  
        <p class="pill">{{ producto()!.categoría }}</p>  
    </div>  
    Añadir al carrito  
}
```

8. Guarde los cambios, espere a que la aplicación se vuelva a cargar y observe el resultado de la aplicación:

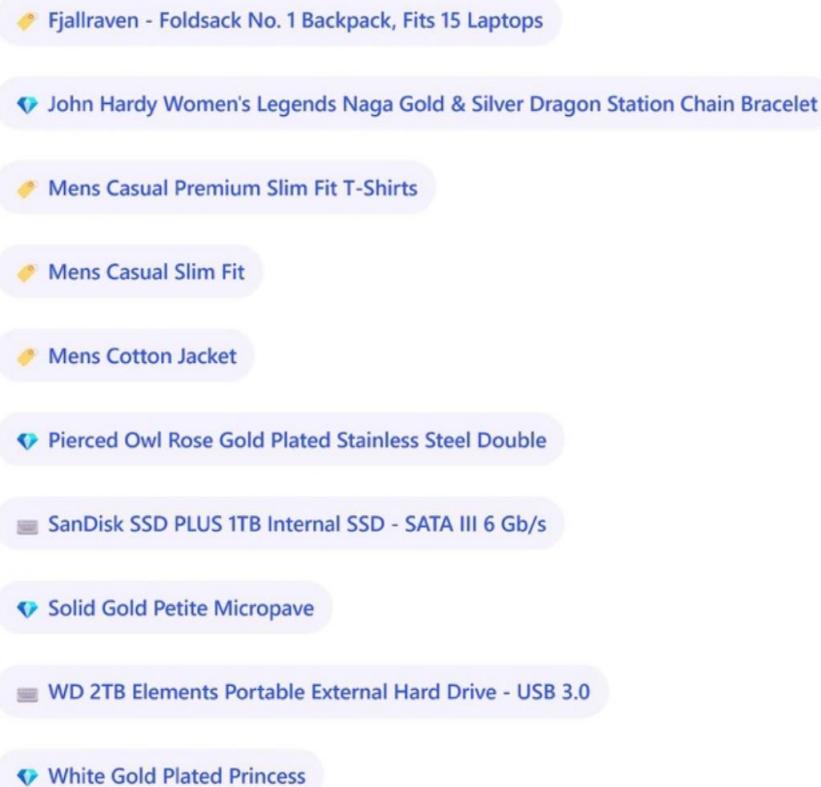


Figura 8.3: Lista de productos con categorías

Si hace clic en un producto de la lista, notará que los detalles del producto se muestran correctamente:

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

€109.00

electronics

Add to cart

Figura 8.4: Detalles del producto

El componente de detalles del producto continúa funcionando como se esperaba porque pasamos el producto seleccionado como una propiedad de entrada de la lista de productos:

```
<detalle-del Producto-de-la-aplicación  
  [producto] = "ProductoSeleccionado"  
  (añadido) = "onAdded()"  
></detalle-del Producto-de-la-aplicación>
```

Cambiaremos el comportamiento anterior y obtendremos los detalles del producto directamente de la API mediante una solicitud HTTP GET. La API de Fake Store contiene un método de punto final que podemos usar para obtener los detalles de un producto específico según su ID:

1. Abra el archivo products.service.ts y cree un nuevo método getProduct que acepte el ID del producto como parámetro e inicie una solicitud GET a la API en función de ese ID:

```
obtenerProducto(id: número): Observable<Producto> {  
  devuelve esto.http.get<Product>(`${this.productsUrl}/${id}`);  
}
```

El método anterior utiliza el método get del servicio HttpClient . Acepta la URL del punto final del producto seguida del ID del producto como parámetro.

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,  
  aporte,  
  producción,  
  OnChanges  
} de '@angular/core';  
importar { Producto } de './producto';  
importar { Observable } de 'rxjs';  
importar { ProductsService } desde './products.service';
```

3. Agregue la siguiente propiedad en la clase ProductDetailComponent :

```
id = entrada<número>();
```

La propiedad del componente id se utilizará para pasar el ID del producto seleccionado de la lista.

4. Reemplace la propiedad de entrada del producto con el siguiente observable:

```
producto$: Observable<Producto> | indefinido;
```

La propiedad product\$ se utilizará para llamar al método getProduct desde el servicio.

5. Agregue un constructor en la clase ProductDetailComponent e inyecte ProductsService:

```
constructor( productoServicioPrivado: ProductosServicio ) { }
```

6. Agregue OnChanges en la lista de interfaces implementadas:

La clase de exportación ProductDetailComponent implementa OnChanges

7. Implemente el método ngOnChanges de la siguiente manera:

```
ngOnChanges(): void {
  este.producto$ = este.productoServicio.getProduct(este.id());
}
```

En el método anterior, asignamos el valor del método getProduct de ProductsService a la propiedad del componente producto\$ cada vez que se pasa un nuevo id usando el enlace de entrada.

8. Abra el archivo product-detail.component.html y modifique su contenido para que utilice el observable product\$:

```
@let producto = (producto$ | async); @if
(producto) {
  <p>Usted seleccionó:
    <strong>{{producto.título}}</strong> </p>

  <p>{{producto.precio | moneda:'EUR'}}</p> <div
  class="pill-group"> <p
    class="pill">{{ producto.categoría }}</p>
  </div>
  Añadir al carrito
}
```

9. Finalmente, abra el archivo product-list.component.html y vincule el id de la propiedad selectedProduct al enlace de entrada id del componente <app-product-detail> :

```
<detalle-del Producto-de-la-aplicación
  [id] = "ProductoSeleccionado?.id"
```

```
(añadido)="onAdded()"

></detalle-del-producto-de-la-aplicación>
```

Si ejecutamos la aplicación usando el comando `ng serve` y seleccionamos un producto de la lista, verificaremos que el detalle del producto se muestre correctamente.

Hemos aprendido a obtener una lista de elementos y un solo elemento de una API de backend y hemos cubierto la parte de lectura de una operación CRUD. En la siguiente sección, cubriremos las partes restantes de una operación CRUD, que se centran principalmente en la modificación de datos.

Modificar datos a través de HTTP

Modificar datos en una aplicación CRUD suele implicar añadir nuevos datos y actualizar o eliminar los existentes. Para demostrar cómo implementar esta funcionalidad en una aplicación Angular mediante el cliente HTTP, realizaremos los siguientes cambios en nuestra aplicación:

- Crea un componente Angular para agregar nuevos productos
- Modificar el componente de detalle del producto para cambiar el precio de un producto existente
- Agregue un botón en el componente de detalle del producto para eliminar un producto existente

Ya hemos mencionado que ninguna operación HTTP conserva datos físicamente en el almacén falso.

API, por lo que necesitamos implementar un mecanismo de caché local para los datos de nuestros productos e interactuar con ellos directamente en el servicio de productos:

1. Abra el archivo `products.service.ts` e importe el operador RxJS del mapa :

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

2. Modifique el método `getProducts` de la siguiente manera:

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

El método anterior llena la matriz de productos con datos de la API y devuelve el producto. datos como observables.

3. Modifique el método `getProduct` para que utilice la matriz de productos para devolver un producto. objeto en lugar de la API de Fake Store:

```
obtenerProducto(id: número): Observable<Producto> {
  const producto = this.productos.find(p => p.id === id); return
  of(producto!);
}
```

Ahora tenemos nuestro servicio de productos en funcionamiento y podemos comenzar a construir el componente para agregar nuevos productos.

Añadiendo nuevos productos

Para agregar un nuevo producto a través de nuestra aplicación, necesitamos enviar sus detalles a la API de Fake Store:

1. Abra el archivo `products.service.ts` y agregue el siguiente método:

```
addProduct(newProduct: Partial<Producto>): Observable<Producto> {
  devuelve este.http.post<Producto>(este.productUrl, nuevoProducto).pipe(
    mapa(producto =>
      { this.productos.push(producto);
        return producto; } );
}
```

En el fragmento anterior, utilizamos el método `post` de la clase `HttpClient` y pasamos el punto final de productos de la API junto con un nuevo objeto de producto como parámetros.



El tipo genérico definido en el método `post` indica que el producto devuelto por la API es un objeto "Producto". También añadimos el nuevo producto a la caché local y lo devolvemos.

2. Ejecute el siguiente comando CLI de Angular para crear un nuevo componente:

```
ng generar componente producto-crear
```

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

```
importar { ProductsService } desde '../products.service';
```

4. Cree un constructor e inyecte la clase ProductsService :

```
constructor( productosServicioprivado: ProductosServicio ) {
```

5. Agregue el siguiente método a la clase de componente:

```
createProduct(título: cadena, precio: cadena, categoría: cadena) {
    este.productsService.addProduct({
        título,
        precio: Número(precio),
        categoría })
    .subscribe();
}
```



No necesitamos cancelar la suscripción al interactuar con el cliente HTTP de Angular porque el marco lo hará automáticamente por nosotros.

El método anterior acepta los detalles del producto como parámetros y llama al método addProduct de la clase ProductsService . Usamos la función nativa Number para convertir los valor del precio a un número porque se pasará como una cadena desde la plantilla.

6. Abra el archivo product-create.component.html y reemplace su contenido con lo siguiente
Plantilla HTML:

```
Añadir nuevo producto
<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>
    <seleccionar id="categoría" #categoría>
        <option>Seleccionar una categoría</option>
        <option value="electronics">Electrónica</option>
    </seleccionar>
</div>
```

```
<option value="jewelery">Joyas</option> <option>Otros</  
option>  
</seleccionar>  
</div>  
<div>  
    <button (click)="createProduct(title.value, price.value, category.  
valor)">Crear</button>  
</div>
```

En la plantilla anterior, vinculamos el método `createProduct` al evento de clic del botón Crear y pasamos el valor de los elementos HTML `<input>` y `<select>` utilizando las respectivas variables de referencia de plantilla.

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

```
entrada  
{ radio del borde: 4px;  
relleno: 8px;  
margen inferior: 16px;  
borde: 1px sólido #BDBDBD;  
}
```

Además, mueva los estilos relacionados con los botones del archivo `product-detail.component.css` al archivo de estilos CSS global.

8. Abra el archivo `product-create.component.css` y agregue los siguientes estilos CSS para darle

Una apariencia agradable para nuestro nuevo componente:

```
entrada  
{ ancho: 200px;  
}  
  
seleccionar {  
    radio del borde: 4px;  
    relleno: 8px;  
    margen inferior: 16px;  
    borde: 1px sólido #BDBDBD;  
    ancho: 220px;  
}
```

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

9. Abra el archivo product-list.component.ts e importe la clase ProductCreateComponent :

```
importar { AsyncPipe } de '@angular/common'; importar
{ Component, OnInit } de '@angular/core'; importar { Observable } de
'rxjs'; importar { Producto } de '../product';

importar { ProductDetailComponent } de './product-detail/product-detail.component'; importar
{ SortPipe } de '../
sort.pipe'; importar { ProductsService } de '../
products.service'; importar { ProductCreateComponent } de './product-create/
product-
crear.componente';
```



```
@Component({
 selector: 'app-product-list',
 importaciones: [
 ComponenteDetalleDeProducto,
 SortPipe,
 Tuberia asíncrona,
 ProductoCrearComponente
 ],
 URL de plantilla: './lista-de-productos.componente.html',
 styleUrls: ['./product-list.component.css' })
```

10. Finalmente, abra el archivo product-list.component.html y agregue el siguiente fragmento en

El final de la plantilla:

```
<app-product-create></app-product-create>
```

Si ahora ejecutamos nuestra aplicación Angular usando el comando ng serve , deberíamos ver el componente para agregar nuevos productos al final de la página:

Add new product

Title

Price

Category

Create

Figura 8.5: Crear un producto

Para experimentar, intente agregar un nuevo producto completando sus detalles, haciendo clic en el botón Crear y verificando que el nuevo producto se haya agregado a la lista.

La siguiente característica que agregaremos a nuestra aplicación es modificar datos cambiando el precio de un producto existente.

Actualización del precio del producto. El precio de un producto en una aplicación de comercio electrónico puede tener que cambiar en algún momento. Necesitamos ofrecer a nuestros usuarios una forma de actualizarlo a través de nuestra aplicación:

1. Abra el archivo products.service.ts y agregue un nuevo método para actualizar un producto:

```
updateProduct(id: número, precio: número): Observable<Product> {
    devuelve este.http.patch<Product>(`${this.productsUrl}/${id}`, {
        precio }).pipe( map(producto
            => { const index = this.products.findIndex(p => p.id === id);
            this.products[index].price = precio;
            devolver
            producto; }) );
}
```

En el método anterior, usamos el método patch de la clase HttpClient para enviar a la API los detalles del producto que queremos modificar. También actualizamos el precio del producto seleccionado en la caché local de productos y lo devolvemos.



Como alternativa, podríamos haber usado el método put del cliente HTTP. El método patch debe usarse cuando queremos actualizar solo un subconjunto de un objeto, mientras que el método put interactúa con todas las propiedades del objeto. En este caso, no queremos actualizar el título del producto, por lo que usamos el método patch. Ambos métodos aceptan el punto final de la API y el objeto que queremos actualizar como parámetros.

2. Agregue el siguiente método a la clase ProductDetailComponent :

```
changePrice(producto: Producto, precio: cadena) {
    este.productService.updateProduct(producto.id, Número(precio)).
    suscribir();
}
```

El método anterior acepta un producto existente y su nuevo precio como parámetros y llama al método updateProduct de la clase ProductsService .

3. Abra el archivo product-detail.component.html y agregue un elemento <input> y un elemento <button> . elemento después del párrafo del precio:

```
@let producto = (producto$ | async);
@if (producto) {
<p>Usted seleccionó:
<strong>{{producto.título}}</strong>
</p>
<p>{{producto.precio | moneda:'EUR'}}</p>
<input placeholder="Nuevo precio" #price type="number" />
<botón
    clase="secundaria"
    (clic)="cambiarPrecio(producto, precio.valor)">
    Cambiar
</botón>
<div class="grupo-de-píldoras">
    <p class="pill">{{ producto.categoría }}</p>
```

```
</div>
Añadir al carrito
}
```

El elemento <input> se utiliza para ingresar el nuevo precio del producto y define el precio. Variable de referencia de plantilla. El evento de clic del elemento <button> está vinculado al método changePrice , que pasa el objeto de producto actual y el valor del precio. variable.

- Finalmente, abra el archivo product-detail.component.css y agregue los siguientes estilos CSS:

```
botón.secundario {
  pantalla: en línea;
  margen izquierdo: 5px;
  --button-accent: var(--rosa vivo);
}
```

- Ejecute el comando ng serve para iniciar la aplicación Angular y seleccionar un producto de La lista. Los detalles del producto deberían ser similares a los siguientes:

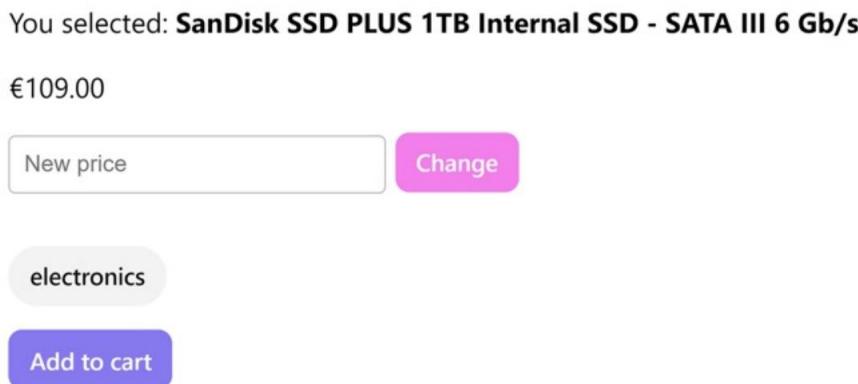


Figura 8.6: Detalles del producto

- Ingrese un precio en el cuadro de entrada Nuevo precio y haga clic en el botón Cambiar . El precio actual... debe actualizarse para reflejar el cambio, por ejemplo:

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

€79.00

79

Change

electronics

Add to cart

Figura 8.7: Detalles del producto con precio modificado

Ahora podemos modificar un producto cambiando su precio.



Recuerda que los cambios en los productos provenientes de la API de Fake Store no se conservan físicamente. Si cambias el precio y actualizas el navegador, se restaurará el precio inicial.

El siguiente y último paso de nuestra aplicación CRUD será eliminar un producto existente.

Eliminar un producto

Eliminar un producto de una aplicación de tienda online no es muy común. Sin embargo, necesitamos ofrecer una función para ello en caso de que los usuarios introduzcan datos incorrectos o no válidos y deseen eliminarlo posteriormente. En nuestra aplicación, la eliminación de un producto existente se realizará con el componente de detalles del producto:

1. Abra el archivo products.service.ts e importe el operador tap desde el paquete rxjs :

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

2. Agregue el siguiente método a la clase ProductsService :

```
deleteProduct(id: número): Observable<void> {
  devuelve esto.http.delete<void>(`${this.productUrl}/${id}`).pipe(
    toque(() => {
      const index = this.products.findIndex(p => p.id === id);
      este.productos.splice(index, 1);
    })
  );
}
```

En el método anterior, usamos el método delete de la clase HttpClient , pasando el endpoint de productos y el ID del producto que queremos eliminar en la API. También usamos el método splice de la matriz de productos para eliminar el producto de la caché local.

El tipo de retorno del método se establece en Observable<void> porque actualmente no nos interesa el resultado de la solicitud HTTP. Solo necesitamos saber si se realizó correctamente. También usamos el operador tap de RxJS porque no modificamos el valor devuelto del observable.

3. Abra el archivo product-detail.component.ts y cree una nueva propiedad de salida en el Clase ProductDetailComponent :

```
eliminado = salida();
```

La propiedad anterior notificará a ProductListComponent que el producto seleccionado Ha sido eliminado.

4. Cree el siguiente método, que llama al método deleteProduct de la Clase ProductsService y activa el evento de salida eliminado :

```
eliminar(producto: Producto) {
    este.productService.deleteProduct(producto.id).subscribe(() => {
        este.eliminado.emit();
    });
}
```

5. Abra el archivo product-detail.component.html , cree un elemento <button> y vincule su evento de clic al método de emisión de la salida eliminada :

```
@let producto = (producto$ | async);
@if (producto) {
    <p>Usted seleccionó:
        <strong>{{producto.título}}</strong>
    </p>
    <p>{{producto.precio | moneda:'EUR'}}</p>
    <input placeholder="Nuevo precio" #price type="number" />
    <botón
        clase="secundaria"
        (clic)="cambiarPrecio(producto, precio.valor)">
        Cambiar
    </botón>
```

```
<div class="pill-group"> <p  
    class="pill">{{ producto.categoría }}</p>  
</div>  
  
<div class="grupo-de-botones">  
    Añadir al carrito  
  
</div>  
}
```

En el fragmento anterior, agrupamos los dos botones en un elemento HTML `<div>` para que aparezcan uno al lado del otro.

6. Agregue un estilo apropiado para el nuevo botón y el grupo de botones en el archivo `product-detail.component.css` :

```
botón.eliminar {  
    pantalla: en línea;  
    margen-izquierdo:  
        5px; --button-accent: var(--hot-red);  
}  
  
.grupo de botones {  
    pantalla: flexible;  
    flex-direction: fila; align-  
    items: inicio;  
    flex-wrap: envolver;  
}
```

7. Abra el archivo `product-list.component.html` y agregue un enlace al evento eliminado de el componente `<app-product-detail>` :

```
<detalle-del Producto-de-la-aplicación  
[id]="ProductoSeleccionado?.id"  
(añadido)="onAdded()"  
(eliminado)="ProductoSeleccionado = indefinido"  
></app-product-detail>
```

Si ejecutamos la aplicación usando el comando `ng serve` y seleccionamos un producto de la lista, deberíamos ver algo como lo siguiente:

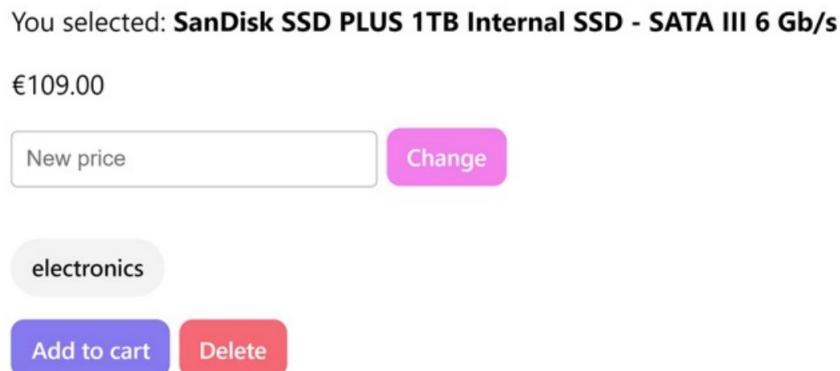
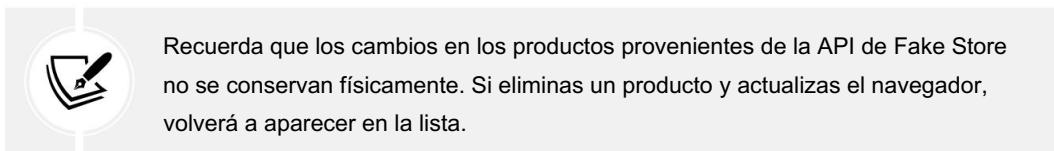
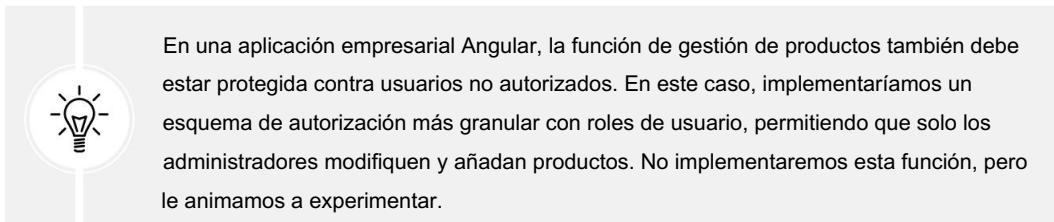


Figura 8.8: Detalles del producto

El componente de detalle del producto ahora tiene un botón Eliminar que elimina el producto y lo quita de la lista cuando se hace clic.



La aplicación de tienda online que hemos creado hasta ahora cuenta con un botón "Añadir al carrito" que permite añadir un producto al carrito de compras. Este botón aún no tiene mucha funcionalidad, pero implementaremos la funcionalidad completa del carrito en los siguientes capítulos. Según la documentación de la API de Fake Store, los carritos de compras solo están disponibles para usuarios autenticados, por lo que debemos asegurarnos de que el botón "Añadir al carrito"... El botón sólo estará disponible para ellos en nuestra aplicación.



En la siguiente sección, aprenderemos sobre la autenticación y autorización en Angular.

Autenticación y autorización con HTTP

La API de Fake Store proporciona un punto final para la autenticación de usuarios. Contiene un método de inicio de sesión que acepta un nombre de usuario y una contraseña como parámetros y devuelve un token de autenticación. Usaremos el token de autenticación en nuestra aplicación para diferenciar entre un usuario conectado y un usuario... invitado.



Un grupo predefinido desde el punto final de los usuarios en <https://fakestoreapi.com/users> proporciona el nombre de usuario y la contraseña.

Exploraremos los siguientes temas de autenticación y autorización en esta sección:

- Autenticación con una API de backend
- Autorizar a los usuarios para determinadas funciones
- Autorizar solicitudes HTTP mediante interceptores

Comencemos con el tema de la autenticación con la API de Fake Store.

Autenticación con una API de backend

En las aplicaciones Angular del mundo real, solemos crear un componente Angular que permite a los usuarios iniciar y cerrar sesión en la aplicación. Un servicio Angular se comunicará con la API y gestionará... todas las tareas de autenticación.

Comencemos creando el servicio de autenticación:

1. Ejecute el siguiente comando para crear un nuevo servicio Angular:

```
ng genera autorización de servicio
```

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

```
importar { Inyectable, calculado, inyectar, señal } de '@angular/
centro';
importar { HttpClient } desde '@angular/common/http';
importar { Observable, tap } de 'rxjs';
importar { APP_SETTINGS } desde './app.settings';
```

3. Cree las siguientes propiedades en la clase AuthService :

```
token de acceso privado = señal(");  
authUrl privado = inyectar(APP_SETTINGS.apiUrl + '/auth';  
isLoggedIn = calculado(() => este.accessToken() != "");
```

En el fragmento anterior, la señal accessToken almacenará el token de autenticación de la API y la señal isLoggedIn indica si el usuario ha iniciado sesión. El estado de sesión iniciada del usuario depende de si la propiedad accessToken tiene un valor.



Las señales se pueden utilizar no sólo en componentes angulares sino también dentro de los servicios.

La propiedad authUrl apunta a la URL del punto final de autenticación de la API de Fake Store.

4. Inyecte la clase HttpClient en el constructor:

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

5. Cree un método de inicio de sesión para permitir que los usuarios inicien sesión en la API de Fake Store:

```
login(nombre de usuario: cadena, contraseña: cadena): Observable<cadena> {  
    devuelve este.http.post<string>(este.authUrl + '/login', {  
        nombre de usuario, contraseña  
    }).pipe(tap(token => this.accessToken.set(token)));  
}
```

El método anterior inicia una solicitud POST a la API, utilizando el punto de acceso de inicio de sesión e introduciendo el nombre de usuario y la contraseña en el cuerpo de la solicitud. El observable devuelto por la solicitud POST se pasa al operador tap , que actualiza la señal accessToken .

6. Cree un método de cierre de sesión que restablezca la señal accessToken :

```
cerrar sesión() {  
    este.accessToken.set("");  
}
```

Ya hemos configurado la lógica empresarial para autenticar usuarios en nuestra aplicación Angular. En la siguiente sección aprenderemos cómo usarlo para controlar la autorización en la aplicación.

Autorizar el acceso de usuarios

Primero, crearemos un componente de autenticación que permitirá a nuestros usuarios iniciar y cerrar sesión en la aplicación:

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

```
ng genera la autenticación del componente
```

2. Abra el archivo auth.component.ts y agregue la siguiente declaración de importación :

```
importar { AuthService } desde './auth.service';
```

3. Inyecte AuthService en el constructor del componente:

```
constructor(public authService: AuthService) {}
```

En el fragmento anterior, usamos el modificador de acceso público para injectar AuthService porque queremos que sea accesible desde la plantilla del componente.

4. Cree los siguientes métodos en la clase AuthComponent :

```
acceso() {
  este.authService.login('david_r', '3478*#54').subscribe();
}

cerrar sesión() {
  este.authService.logout();
}
```

En el fragmento anterior, el método de inicio de sesión utiliza credenciales predefinidas del punto final del usuario.

5. Abra el archivo auth.component.html y reemplace su contenido con el siguiente HTML

plantilla:

```
@if (!authService.isLoggedIn()) {
  Iniciar sesión
} @demás {
  Cerrar sesión
}
```

La plantilla anterior contiene dos elementos HTML <button> para fines de inicio y cierre de sesión.

Cada botón se muestra condicionalmente según el valor de la señal isLoggedIn de la clase AuthService .

Ahora podemos aprovechar la señal isLoggedIn en el componente de detalles del producto y alternar la visibilidad del botón Agregar al carrito :

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

```
importar { AuthService } desde './auth.service';
```

2. Inyecte AuthService en el constructor de la clase ProductDetailComponent :

```
constructor( producto privado : ProductosServicio, autenticación pública:  
ServicioAutorización) { }
```

3. Abra el archivo product-detail.component.html y use un bloque @if para mostrar la opción Agregar.

Botón para añadir al carrito condicionalmente:

```
@if (authService.isLoggedIn()) {  
    Añadir al carrito  
}
```

4. Abra el archivo app.component.ts e importe la clase AuthComponent :

```
importar { Componente, inyectar } desde '@angular/core'; importar  
{ RouterOutlet } desde '@angular/router';  
importar { ProductListComponent } de './product-list/product-list. component'; importar  
  
{ CopyrightDirective } de './copyright.directive';  
importar { APP_SETTINGS } desde './app.settings'; importar  
{ AuthComponent } desde './auth/auth.component';  
  
"@Componente({  
    selector: 'app-root',  
    importaciones: [  
        Salida de enrutador,  
        Componente de lista de productos,  
        Directiva sobre derechos de autor,  
        Componente de autenticación  
    ],  
    URL de plantilla: './app.component.html',  
    estiloUrl: './app.component.css' })
```

5. Abra el archivo app.component.html y agregue la etiqueta <app-auth> dentro del HTML <header> elemento:

```
<encabezado>
{{configuraciones.título}}
<app-auth></app-auth>
</encabezado>
```

Para probar la función de autenticación en la aplicación, siga estos pasos:

1. Ejecute el comando ng serve para iniciar la aplicación y navegue a <http://localhost:4200>.
2. Seleccione un producto de la lista y verifique que el botón Agregar al carrito no esté visible.
3. Haga clic en el botón Iniciar sesión en la esquina superior izquierda de la página. El texto debería cambiar a Cerrar sesión. después de haber iniciado sesión exitosamente en la API de Fake Store, debería aparecer el botón Agregar al carrito .

¡Felicitaciones! Has agregado patrones básicos de autenticación y autorización a tu aplicación Angular.

Es común en las aplicaciones empresariales realizar la autorización en la capa de lógica de negocio durante la comunicación con la API de backend. Esta suele requerir ciertas llamadas a métodos para pasar el token de autenticación en cada solicitud a través de los encabezados. En la siguiente sección, aprenderemos a trabajar con encabezados HTTP .

Autorizar solicitudes HTTP

La API de Fake Store no requiere autorización para comunicarse con sus endpoints. Sin embargo, supongamos que trabajamos con una API de backend que espera que todas las solicitudes HTTP contengan un token de autenticación mediante encabezados HTTP. Un patrón común en aplicaciones web es incluir el token en un encabezado de autorización . Podemos usar encabezados HTTP en una aplicación Angular importando la clase HttpHeaders del espacio de nombres @angular/common/http y modificando nuestros métodos según corresponda. A continuación, se muestra un ejemplo de cómo debería verse el método getProducts :

```
obtenerProductos(): Observable<Producto[]> {
  opciones constantes = {
    parámetros: nuevos HttpParams().set('limit', 10),
    encabezados: nuevos HttpHeaders({Autorización: 'myToken' })
  };
  devuelve este.http.get<Product[]>(this.productsUrl, opciones).
  tubería(mapa(productos => {
    este.productos = productos;
```

```

    devolver productos;
});

}

```



Para simplificar, usamos un valor predefinido para el token de autenticación. En un escenario real, podríamos obtenerlo del almacenamiento local del navegador o de algún otro modo.

Todos los métodos de HttpClient aceptan un objeto opcional como parámetro para pasar opciones adicionales a una solicitud HTTP, incluyendo encabezados HTTP. Para establecer los encabezados, usamos la propiedad headers del objeto options y creamos una nueva instancia de la clase HttpHeaders como valor. HttpHeaders es un objeto que define encabezados HTTP personalizados.

Ahora, imaginemos qué sucedería si necesitáramos pasar el token de autenticación en todos los métodos de la clase ProductsService. Deberíamos acceder a cada uno de ellos y escribir el mismo código repetidamente. Nuestro código podría volverse rápidamente desordenado y difícil de probar. Afortunadamente, el cliente HTTP de Angular cuenta con otra función que podemos usar para ayudarnos en esta situación: los interceptores.

Un interceptor HTTP es un servicio de Angular que intercepta las solicitudes y respuestas HTTP que pasan a través del cliente HTTP de Angular. Se puede utilizar en los siguientes escenarios:

- Cuando queremos pasar encabezados HTTP personalizados en cada solicitud, como una autenticación simbólica
- Cuando queremos mostrar un indicador de carga mientras esperamos una respuesta del servidor
- Cuando queremos proporcionar un mecanismo de registro para cada comunicación HTTP

En nuestro caso, podemos crear un interceptor para pasar el token de autenticación a cada solicitud HTTP:

1. Ejecute el siguiente comando para crear un nuevo interceptor:

```
ng genera la autenticación del interceptor
```

2. Abra el archivo app.config.ts e importe la función withInterceptors desde el Espacio de nombres @angular/common/http :

```
importar { provideHttpClient, withInterceptors } desde '@angular/
común/http';
```

La función withInterceptors se utiliza para registrar un interceptor con el cliente HTTP.

3. Importa el interceptor que creamos en el paso anterior usando la siguiente declaración:

```
importar { authInterceptor } desde './auth.interceptor';
```

4. Modifique el método provideHttpClient para registrar el authInterceptor:

```
export const appConfig: ApplicationConfig =
  { proveedores: [
    proporcionarZoneChangeDetection({ eventCoalescing: true }),
    proporcionarRouter(rutas),
    proporcionarHttpClient(withInterceptors([authInterceptor])),
    { proporcionar: APP_SETTINGS, useValue: appSettings }
  ]
};
```

La función withInterceptors acepta una lista de interceptores registrados, y su orden es importante.

En el siguiente diagrama, se puede ver cómo los interceptores procesan las solicitudes y respuestas HTTP según su orden:

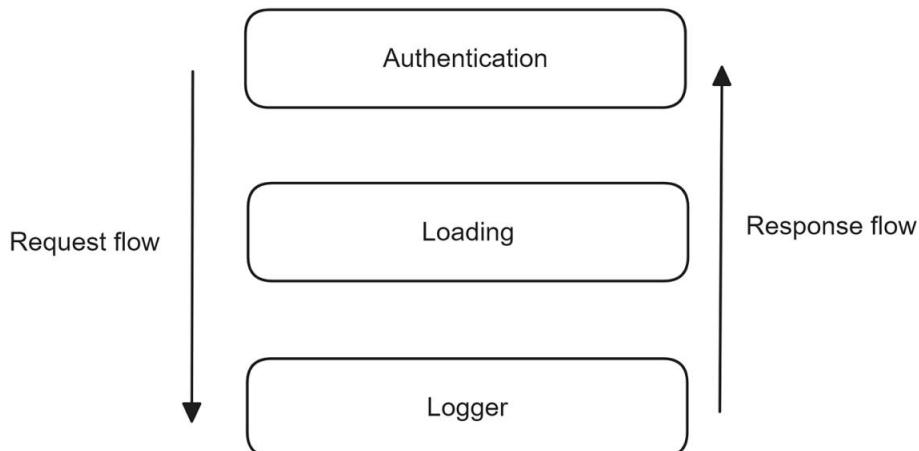


Figura 8.9: Orden de ejecución de los interceptores angulars



De forma predeterminada, el último interceptor antes de enviar la solicitud al servidor es un servicio Angular integrado llamado HttpBackend.

5. Abra el archivo auth.interceptor.ts y modifique la función de flecha de authInterceptor.

funciona de la siguiente manera:

```
export const authInterceptor: HttpInterceptorFn = (req, next) => {
  constante authReq = req.clone({
    setHeaders: { Autorización: 'myToken' }
```

```
});  
devolver siguiente(authReq);  
};
```

La función de flecha acepta los siguientes parámetros: req, que indica la solicitud actual, y next, que representa el siguiente interceptor en la cadena. En el fragmento anterior, usamos el método clone para modificar la solicitud existente, ya que las solicitudes HTTP son inmutables por defecto. De igual forma, debido a la naturaleza inmutable de los encabezados HTTP, usamos el método setHeaders para actualizarlos. Finalmente, delegamos la solicitud al siguiente interceptor mediante el método handle .

Los interceptores pueden usar el método inject para obtener las dependencias que necesiten del mecanismo de DI de Angular. Por ejemplo, si quisieramos usar la clase AuthService dentro del interceptor, podríamos modificarla de la siguiente manera:

```
importar { inyectar } desde '@angular/core';  
importar { HttpInterceptorFn } desde '@angular/common/http';  
importar { AuthService } desde './auth.service';  
  
exportar const authInterceptor: HttpInterceptorFn = (req, next) => {  
  const authService = inyectar(AuthService);  
  constante authReq = req.clone(  
    setHeaders: { Autorización: 'myToken' }  
  );  
  devolver siguiente(authReq);  
};
```

En aplicaciones creadas con versiones anteriores del framework Angular, puede observar que los interceptores son clases de TypeScript en lugar de funciones puras. Para registrar un interceptor con el cliente HTTP, necesitamos agregar el siguiente literal de objeto "provider" en la matriz de proveedores del módulo, que también proporciona el HttpClientModule:



```
{  
  proporcionar: HTTP_INTERCEPTORS,  
  useClass: AuthInterceptor,  
  multi: verdadero  
}
```

En el fragmento anterior, HTTP_INTERCEPTORS es un token de inyección que se puede proporcionar varias veces como lo indica la propiedad multi .

Los interceptores angulares tienen múltiples usos, y la autorización es uno de los más básicos. El paso de tokens de autenticación durante las solicitudes HTTP es un escenario común en las aplicaciones web empresariales.

Resumen

Las aplicaciones web empresariales deben intercambiar información con una API de backend casi a diario. El framework Angular permite que las aplicaciones se comuniquen con una API mediante HTTP mediante el cliente HTTP Angular. En este capítulo, exploramos los componentes esenciales del cliente HTTP Angular.

Aprendimos a alejarnos de la API de búsqueda tradicional y a usar observables para comunicarnos mediante HTTP. Exploramos los componentes básicos de una aplicación CRUD utilizando la API de Fake Store como backend. Investigamos cómo implementar la autenticación y la autorización en aplicaciones Angular. Finalmente, aprendimos qué son los interceptores Angular y cómo usarlos para autorizar llamadas HTTP.

Ahora que sabemos cómo consumir datos de una API de backend en nuestros componentes, podemos mejorar aún más la experiencia de usuario de nuestra aplicación. En el siguiente capítulo, aprenderemos a cargar nuestros componentes mediante la navegación usando el enrutador Angular.

9

Navegando a través de Aplicaciones con enrutamiento

En capítulos anteriores, hicimos un excelente trabajo separando las preocupaciones y añadiendo diferentes capas de abstracción para aumentar la facilidad de mantenimiento de una aplicación Angular. Sin embargo, apenas nos hemos centrado en la experiencia de usuario (UX) de la aplicación.

Nuestra interfaz de usuario está sobrecargada, con componentes dispersos en una sola pantalla. Debemos ofrecer una mejor experiencia de navegación a los usuarios y una forma lógica de cambiar la vista de la aplicación de forma intuitiva. Ahora es el momento adecuado para incorporar el enrutamiento y dividir las diferentes áreas de interés en páginas, conectadas mediante una cuadrícula de enlaces y URL.

Entonces, ¿cómo implementamos un esquema de navegación entre los componentes de una aplicación Angular? Usamos el enrutador Angular y creamos enlaces personalizados para que nuestros componentes reaccionen a ellos.

Este capítulo contiene las siguientes secciones:

- Presentamos el enrutador angular
- Configuración de las rutas principales
- Organizar rutas de aplicaciones
- Pasar parámetros a las rutas
- Mejora de la navegación con funciones avanzadas