



Beginning **Flutter**[®]

A Hands On Guide To App Development

Marco L. Napoli

PRINCIPIANTE FLUTTER®

INTRODUCCIÓN	xxx
PARTE I LOS FUNDAMENTOS DE LA PROGRAMACIÓN FLUTTER	
CAPÍTULO 1 Introducción a Flutter y primeros pasos	3
CAPÍTULO 2 Creación de una aplicación Hello World	25
CAPÍTULO 3 Aprendiendo los conceptos básicos de Dart	43
CAPÍTULO 4 Creación de una plantilla de proyecto inicial.	sesenta y cinco
CAPÍTULO 5 Descripción del árbol de widgets.	77
PARTE II FLUTTER INTERMEDIO: DESARROLLAR UNA APLICACIÓN	
CAPÍTULO 6 Uso de widgets comunes.	103
CAPÍTULO 7 Adición de animación a una aplicación.	151
CAPÍTULO 8 Creación de la navegación de una aplicación	177
CAPÍTULO 9 Creación de listas de desplazamiento y efectos	221
CAPÍTULO 10 Planos de construcción	253
CAPÍTULO 11 Aplicación de la interactividad	267
CAPÍTULO 12 Plataforma de escritura: código nativo.	307
PARTE III CREACIÓN DE APLICACIONES LISTAS PARA LA PRODUCCIÓN	
CAPÍTULO 13 Almacenamiento de datos con persistencia local.	327
CAPÍTULO 14 Adición de Firebase y Firestore Backend	375
CAPÍTULO 15 Agregar administración de estado a la aplicación Firestore Client	411
CAPÍTULO 16 Adición de BLoC a las páginas de la aplicación Firestore Client.	453
ÍNDICE	489

COMIENZO
Flutter®

COMIENZO

Flutter®

UNA GUÍA PRÁCTICA PARA EL DESARROLLO DE APLICACIONES

Marco L. Nápoles



A Wiley Brand

Comienzo de Flutter®: una guía práctica para el desarrollo de aplicaciones

Publicado por

John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianápolis, IN 46256
www.wiley.com

Copyright © 2020 por John Wiley & Sons, Inc., Indianápolis, Indiana

Publicado simultáneamente en Canadá

ISBN: 978-1-119-55082-2

ISBN: 978-1-119-55087-7 (ebk)

ISBN: 978-1-119-55085-3 (ebk)

Fabricado en los Estados Unidos de América

Ninguna parte de esta publicación puede ser reproducida, almacenada en un sistema de recuperación o transmitida de ninguna forma o por ningún medio, ya sea electrónico, mecánico, fotocopiado, grabado, escaneado o de otro modo, excepto según lo permitido por las Secciones 107 o 108 de la Ley de derechos de autor de los Estados Unidos de 1976. Act. sin el permiso previo por escrito del editor o la autorización mediante el pago de la tarifa correspondiente por copia al Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646 -8600. Las solicitudes de permiso al editor deben dirigirse al Departamento de permisos, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, o en línea en <http://www.wiley.com/go/permissions>.

Límite de responsabilidad/Descargo de responsabilidad de la garantía: El editor y el autor no hacen representaciones ni garantías con respecto a la precisión o la integridad del contenido de este trabajo y renuncian específicamente a todas las garantías, incluidas, entre otras, las garantías de idoneidad para un propósito particular. Ninguna garantía pueda ser creada o extendida por ventas o materiales promocionales. Los consejos y estrategias contenidos en este documento pueden no ser adecuados para todas las situaciones. Este trabajo se vende con el entendimiento de que el editor no se dedica a prestar servicios legales, contables u otros servicios profesionales. Si se requiere asistencia profesional, se deben buscar los servicios de una persona profesional competente. Ni el editor ni el autor serán responsables de los daños derivados del mismo. El hecho de que en este trabajo se haga referencia a una organización o sitio web como una cita y/o una posible fuente de información adicional no significa que el autor o el editor respalde la información que la organización o el sitio web pueda proporcionar o las recomendaciones que pueda hacer. . Además, los lectores deben tener en cuenta que los sitios web de Internet enumerados en este trabajo pueden haber cambiado o desaparecido entre el momento en que se escribió este trabajo y el momento en que se leyó.

Para obtener información general sobre nuestros otros productos y servicios, comuníquese con nuestro Departamento de atención al cliente dentro de los Estados Unidos al (877) 762-2974, fuera de los Estados Unidos al (317) 572-3993 o al fax (317) 572-4002.

Wiley publica en una variedad de formatos impresos y electrónicos y por impresión bajo demanda. Es posible que parte del material incluido con las versiones impresas estándar de este libro no se incluya en los libros electrónicos o en la impresión bajo demanda. Si este libro hace referencia a medios como un CD o DVD que no está incluido en la versión que compró, puede descargar este material en <http://booksupport.wiley.com>. Para obtener más información sobre los productos Wiley, visite www.wiley.com.

Número de control de la Biblioteca del Congreso: 2019940772

Marcas comerciales: Wiley, el logotipo de Wiley, Wrox, el logotipo de Wrox, Programmer to Programmer y la imagen comercial relacionada son marcas comerciales o marcas comerciales registradas de John Wiley & Sons, Inc. y/o sus filiales, en los Estados Unidos y otros países, y no se puede utilizar sin permiso por escrito. Flutter es una marca registrada de Google, LLC. Todas las demás marcas comerciales son propiedad de sus respectivos dueños. John Wiley & Sons, Inc., no está asociado con ningún producto o proveedor mencionado en este libro.

A Dios; mi esposa Carla; mis hijos, Michael,
Timothy y Joseph; y usted, el lector.

SOBRE EL AUTOR

Marco L. Napoli es el director ejecutivo de Pixolini, Inc. y un experimentado desarrollador de aplicaciones móviles, web y de escritorio. Tiene un sólido historial comprobado en el desarrollo de sistemas visualmente elegantes y fáciles de usar. Escribió su primera aplicación iOS nativa en 2008. Su trabajo y las aplicaciones publicadas se pueden ver en www.pixolini.com.

Ha amado las computadoras desde una edad temprana. Su papá se dio cuenta y le compró una PC, y desde entonces ha estado desarrollando software. Asistió a la Universidad de Miami para obtener un título en arquitectura, pero ya había comenzado su propio negocio y después de cuatro años decidió que la arquitectura no era para él. Desarrolló sistemas para una combinación diversa de industrias que incluyen banca, atención médica, bienes raíces, educación, transporte por carretera, entretenimiento y mercados horizontales. Más tarde, una empresa líder en software bancario adquirió su empresa MLN Enterprises, Inc. Los principales productos fueron software de banca hipotecaria, procesamiento y marketing.

Luego, comenzó a consultar y luego creó IdeaBlocks, Inc., con el propósito de consultoría de desarrollo de software. Desarrolló para un cliente que vendía software de hospitalidad para plataformas móviles, de escritorio y web. El enfoque principal del producto estaba en las ventas de hoteles, catering, espacio web, servicio al huésped y software de mantenimiento. Los productos se sincronizaron a través de servidores en la nube usando Microsoft SQL Server con cifrado aplicado a datos confidenciales. Los clientes de sus clientes incluyen Hyatt Place y Summerfield, Hilton Hotel, Holiday Inn, Hampton Inn, Marriott, Best Western, Radisson Hotel, Sheraton Hotels, Howard Johnson, Embassy Suites y muchos más. Una vez hecho su contrato, cerró IdeaBlocks.

Hoy, su enfoque es ejecutar Pixolini. Desarrolla aplicaciones móviles, de escritorio y web para iOS, Mac, Android, Windows y la Web. También imparte un curso en Udemy utilizando una aplicación web que desarrolló para analizar los cálculos de inversión inmobiliaria. Ha desarrollado y publicado más de 10 aplicaciones en cada tienda respectiva.

Fue entrevistado por Hillel Coren para "It's All Widgets Flutter Podcast" el 27 de noviembre de 2018, y el episodio se puede encontrar en <https://itsallwidgets.com/podcast/episodes/1/marco-napoli>.

"No puedo codificar sin espresso, capuchino o café, y me encantan las artes marciales".

Marco está casado con Carla y tienen tres hijos maravillosos.

SOBRE EL EDITOR TÉCNICO

Zeeshan Chawdhary es un tecnólogo ávido, con 14 años de experiencia en la industria. Habiendo comenzado su carrera con el desarrollo móvil con J2ME, pronto incursionó en el desarrollo web, creando aplicaciones web robustas y escalables. Como director de tecnología, ha dirigido equipos para crear aplicaciones web y móviles para empresas como Nokia, Motorola, Mercedes, GM, American Airlines y Marriott. Actualmente es director de desarrollo en un equipo internacional, sirviendo a clientes con tecnologías como Magento, WordPress, WooCommerce, Laravel, NodeJS, Google Puppeteer, ExpressJS, ReactJS y .NET. También es autor de libros sobre iOS, Windows Phone e iBooks.

Zeeshan tiene su sede en Bombay, India. Se le puede contactar en imzeeshanc@gmail.com o en Twitter [@imzeeshan](https://twitter.com/imzeeshan), y mantiene una publicación en Medium en <https://medium.com/@imzeeshan>.

EXPRESIONES DE GRATITUD

Quiero agradecer al talentoso equipo de Wiley, incluidos todos los editores, gerentes y muchas personas detrás de escena que ayudaron a publicar este libro. Mi agradecimiento a Devon Lewis por reconocer desde el principio que Flutter está teniendo un gran impacto en la industria, a Candace Cunningham por sus habilidades de edición de proyectos y sus conocimientos, a Zeeshan Chawdhary por su aporte técnico y sugerencias, a Barath Kumar Rajasekaran y su equipo por preparar la producción del libro, ya Pete Gaughan por estar siempre disponible.

Un agradecimiento especial al equipo de Flutter en Google, especialmente a Tim Sneath, Ray Rischpater y Filip Hráček, por su amabilidad y comentarios invaluosables.

Un agradecimiento a mi esposa e hijos que han escuchado pacientemente y han dado su opinión sobre los proyectos creados en este libro.

CONTENIDO

INTRODUCCIÓN

xxx

PARTE I: LOS FUNDAMENTOS DE LA PROGRAMACIÓN FLUTTER

CAPÍTULO 1: PRESENTACIÓN DE FLUTTER Y PRIMEROS PASOS	3
Introducción a Flutter	4
Definición de widgets y elementos	
Descripción de los eventos del ciclo de vida del widget	5 5
El ciclo de vida del widget sin estado	6
El ciclo de vida de StatefulWidget	
Descripción del árbol de widgets y el árbol de elementos	6 8
Widget sin estado y árboles de elementos	9
Widget con estado y árboles de elementos	10
Instalación del SDK de Flutter	13
Instalación en macOS	13
Requisitos del sistema	13
Obtenga el SDK de Flutter	13
Buscar dependencias	14
Configuración de iOS: Instalar Xcode	14
Configuración de Android: instalar Android Studio	14
Configurar el emulador de Android	15
Instalación en Windows	15
Requisitos del sistema	15
Obtenga el SDK de Flutter	
Comprobar dependencias	
Instalar estudio de Android	
Configurar el emulador de Android	17
Instalación en Linux	17
Requisitos del sistema	17
Obtenga el SDK de Flutter	18
Comprobar dependencias	19
Instalar estudio de Android	19
Configurar el emulador de Android	19
Configuración del editor de estudio de Android	20
Resumen	20

CONTENIDO

CAPÍTULO 2: CREAR UNA APLICACIÓN HELLO WORLD	25
Configuración del proyecto	25
Uso de recarga en caliente	30
Uso de temas para darle estilo a su aplicación	33
Uso de un tema de aplicación global	33
Uso de un tema para parte de una aplicación	35
Descripción de los widgets sin estado y con estado	37
Uso de paquetes externos	38
Búsqueda de paquetes	39
Uso de paquetes	40
Resumen	41
CAPÍTULO 3: APRENDIENDO LOS BÁSICOS DE DART	43
¿Por qué usar dardo?	43
Comentar código Ejecutar	44
el punto de entrada principal() Referenciar	45
variables Declarar variables	45
Números Cadenas	46
Booleanos	47
Listas	47
Mapas	47
Runas	47
	48
	48
Uso de operadores	49
Uso de sentencias de flujo if y else operador	51
ternario para bucles	52
while y do-	52
while while y break	53
continue switch y	54
case	54
	55
Uso de funciones	55
Importar paquetes	57
Uso de clases	57
Herencia de clase	60
Mezclas de clase	60
Implementación de la programación asíncrona	61
Resumen	62

CAPÍTULO 4: CREAR UNA PLANTILLA DE PROYECTO INICIAL

Creación y organización de carpetas y archivos	65
Widgets de estructuración	69
Resumen	74

CAPÍTULO 5: COMPRENSIÓN DEL ÁRBOL DE WIDGET

Introducción a los widgets	77
Creación del árbol de widgets completo	79
Construcción de un árbol de widgets poco profundo	85
Refactorización con una constante	86
Refactorización con un método	86
Refactorización con una clase de widget	91
Resumen	99

PARTE II: FLUTTER INTERMEDIO: DESARROLLAR UNA APLICACIÓN**CAPÍTULO 6: USO DE WIDGETS COMUNES**

Uso de widgets básicos	103
Área segura	107
Envase	108
Texto	112
Texto rico	112
Columna	114
Fila	115
Anidamiento de columnas y filas	115
Botones	119
Botón de acción flotante	119
botón plano	121
Botón elevado	121
IconoBotón	122
BotónMenúEmergente	123
Barra de botones	126
Uso de imágenes e íconos	130
paquete de activos	130
Imagen	131
Icono	132
Uso de decoradores	135
Uso del widget de formulario para validar campos de texto	139
Comprobación de la orientación	143
Resumen	149

CONTENIDO

CAPÍTULO 7: AGREGAR ANIMACIÓN A UNA APLICACIÓN	151
Usando AnimatedContainer	152
Usando AnimatedCrossFade	155
Usando AnimatedOpacity	160
Uso de AnimationController	164
Uso de animaciones escalonadas	170
Resumen	175
CAPÍTULO 8: CREAR LA NAVEGACIÓN DE UNA APLICACIÓN	177
Usando el Navegador	178
Uso de la ruta del navegador con nombre	188
Uso de animación de héroe	188
Uso de la barra de navegación inferior	193
Uso de la barra de aplicaciones inferior	199
Uso de TabBar y TabBarView	203
Usando el Cajón y ListView	207
Resumen	217
CAPÍTULO 9: CREACIÓN DE LISTAS DE DESPLAZAMIENTO Y EFECTOS	221
Uso de la tarjeta	222
Uso de ListView y ListTile Uso de	223
GridView Uso de	230
GridView.count Uso de	230
GridView.extent Uso de	232
GridView.builder Uso de la pila	233
Personalización de	237
CustomScrollView con Slivers Resumen	243
	250
CAPÍTULO 10: DISEÑOS DE EDIFICIOS	253
Una vista de alto nivel del diseño	253
Diseño de la sección meteorológica	256
Diseño de etiquetas	256
Diseño de imágenes de pie de página	257
Diseño final	257
Crear el diseño	257
Resumen	265

CAPÍTULO 11: APLICACIÓN DE LA INTERACTIVIDAD	267
Configuración de GestureDetector: conceptos básicos	267
Implementación de los widgets Draggable y Dragtarget	275
Uso de GestureDetector para mover y escalar	278
Uso de los gestos InkWell y InkResponse	289
Uso del widget descartable	296
Resumen	303
CAPÍTULO 12: PLATAFORMA DE ESCRITURA - CÓDIGO NATIVO	307
Comprender los canales de la plataforma	307
Implementación de la aplicación de canal de la plataforma del cliente	309
Implementación del canal de plataforma de host de iOS	313
Implementación del canal de plataforma de host de Android	318
Resumen	322
PARTE III: CREACIÓN DE APLICACIONES LISTAS PARA LA PRODUCCIÓN	
CAPÍTULO 13: GUARDAR DATOS CON PERSISTENCIA LOCAL	327
Comprender el formato JSON	328
Uso de clases de base de datos para escribir, leer y serializar JSON	330
Formateo de fechas	331
Ordenar una lista de fechas	332
Recuperación de datos con FutureBuilder	333
Creación de la aplicación Diario	335
Adición de las clases de base de datos de revistas	339
Adición de la página de entrada del diario	344
Finalización de la página de inicio del diario	359
Resumen	371
CAPÍTULO 14: AGREGAR EL BACKEND DE FIREBASE Y FIRESTORE	375
¿Qué son Firebase y Cloud Firestore?	376
Estructuración y modelado de datos Cloud Firestore	377
Ver las capacidades de autenticación de Firebase	380
Ver las reglas de seguridad de Cloud Firestore	381
Configuración del proyecto Firebase	383
Agregar una base de datos de Cloud Firestore e implementar seguridad	391
Creación de la aplicación Client Journal	395

CONTENIDO

Agregar paquetes de autenticación y Cloud Firestore a la aplicación cliente	395
Adición de un diseño básico a la aplicación del cliente	403
Adición de clases a la aplicación del cliente	406
Resumen	409
 CAPÍTULO 15: AGREGAR LA GESTIÓN ESTATAL A LA APLICACIÓN CLIENTE DE FIRESTORE	 411
Implementación de la gestión estatal	412
Implementando una clase abstracta	414
Implementando el widget heredado	415
Implementando la clase modelo	416
Implementación de la clase de servicio	417
Implementando el Patrón BLoC	417
Implementación de StreamController, Streams, Sinks y StreamBuilder	419
Gestión del estado del edificio	421
Adición de la clase de modelo de revista	422
Adición de las clases de servicio	424
Agregar la clase de validadores	430
Agregar el patrón BLoC	432
Aregar el bloque de autenticación	432
Adición de AuthenticationBlocProvider	435
Aregar el bloque de inicio de sesión	436
Agregando el HomeBloc	441
Aregar el HomeBlocProvider	443
Agregando el JournalEditBloc	444
Adición de JournalEditBlocProvider	447
Resumen	449
 CAPÍTULO 16: AGREGAR BLOQUES A LAS PÁGINAS DE LA APLICACIÓN CLIENTE DE FIRESTORE	 453
Aregar la página de inicio de sesión	454
Modificación de la página principal	460
Modificación de la página de inicio	465
Adición de la página Editar diario	472
Resumen	484
 ÍNDICE	 489

INTRODUCCIÓN

Flutter se presentó en la Dart Developer Summit de 2015 con el nombre de Sky. Eric Seidel (ingeniero director de Flutter en Google) abrió su charla diciendo que estaba allí para hablar sobre Sky, que era un proyecto experimental presentado como "Dart on mobile". Había creado y publicado una demostración en Android Play Store, y comenzó la demostración afirmando que no había Java dibujando esta aplicación, lo que significa que era nativa. La primera característica que mostró Eric fue un cuadrado girando. Dart conducía el dispositivo a 60 Hertz, que era el primer objetivo del sistema: ser rápido y receptivo. (Quería ir mucho más rápido [es decir, 120 Hertz], pero estaba restringido por la capacidad del dispositivo que estaba usando).

Eric pasó a mostrar funciones multitáctiles, de desplazamiento rápido y otras. Sky proporcionó la mejor experiencia móvil (para usuarios y desarrolladores); los desarrolladores aprendieron lecciones de trabajar en la Web y pensaron que podían hacerlo mejor. La interfaz de usuario (IU) y la lógica comercial se escribieron en Dart.

El objetivo era ser independiente de la plataforma.

Avance rápido hasta 2019, y Flutter ahora está impulsando la plataforma de pantalla inteligente de Google, incluido Google Home Hub, y es el primer paso para admitir aplicaciones de escritorio con Chrome OS. El resultado es que Flutter admite aplicaciones de escritorio que se ejecutan en Mac, Windows y Linux. Flutter se describe como un marco de interfaz de usuario portátil para todas las pantallas, como dispositivos móviles, web, de escritorio e integrados desde una única base de código.

Este libro le enseña cómo desarrollar aplicaciones móviles para iOS y Android a partir de una única base de código utilizando el marco Flutter y Dart como lenguaje de programación. A medida que Flutter se expande más allá de los dispositivos móviles, puede tomar el conocimiento que aprende en este libro y aplicarlo a otras plataformas. No necesitas tener experiencia previa en programación; el libro comienza con los conceptos básicos y avanza hacia el desarrollo de aplicaciones listas para producción.

Escribí este libro en un estilo simple y práctico para enseñarte cada concepto. Puede seguir los ejercicios de estilo de práctica "Pruébelo" para implementar lo que aprende y crear aplicaciones centradas en funciones.

Cada capítulo se basa en los anteriores y agrega nuevos conceptos para mejorar su conocimiento para crear aplicaciones rápidas, hermosas, animadas y funcionales. Al final de este libro, podrá tomar el conocimiento y las técnicas que ha aprendido y aplicarlos para desarrollar sus propias aplicaciones.

En los últimos cuatro capítulos del libro, creará una aplicación de diario con la capacidad de guardar datos localmente y una segunda aplicación de diario que agrega seguimiento de estado de ánimo con gestión de estado, autenticación y capacidades de sincronización en la nube de datos de múltiples dispositivos, incluida la sincronización sin conexión, que es una necesidad para las aplicaciones móviles de hoy. He hecho todo lo posible para enseñarle las técnicas utilizando un enfoque amigable y de sentido común para que pueda aprender los conceptos básicos hasta los conceptos avanzados necesarios para el lugar de trabajo.

Desde la primera vez que vi a Google presentando Flutter, me llamó la atención. Lo que me atrajo especialmente de Flutter fue el concepto de widgets. Usted toma los widgets y los anida (composición) para crear la interfaz de usuario necesaria y, lo mejor de todo, puede crear fácilmente sus propios widgets personalizados. El otro elemento importante que me atrajo de Flutter fue la capacidad de desarrollar para iOS y Android desde una única base de código; esto es algo que había estado necesitando durante mucho tiempo y nunca encontré una gran solución hasta Flutter. Flutter es declarativo; es un marco reactivo moderno donde los widgets manejan el aspecto que debería tener la interfaz de usuario según su estado actual.

Introducción

Mi pasión por desarrollar con Flutter y Dart sigue creciendo, y decidí escribir este libro para compartir mis experiencias y conocimientos con los demás. Creo firmemente que el libro enseña a todos, desde principiantes hasta desarrolladores expertos, brindándoles las herramientas y el conocimiento para construir y avanzar como desarrollador multiplataforma. Este libro está lleno de consejos, ideas, escenarios hipotéticos, diagramas, capturas de pantalla, código de muestra y ejercicios. Todo el código fuente del proyecto está disponible para su descarga en la página web de este libro en www.wiley.com/go/beginningflutter.

PARA QUIEN ES ESTE LIBRO

Este libro es para todos los que quieran aprender a programar aplicaciones móviles multiplataforma utilizando Flutter y Dart. Es para principiantes absolutos que quieren aprender a desarrollar aplicaciones móviles modernas, de rendimiento nativo rápido y reactivas para iOS y Android. Sin embargo, también lo lleva desde un principiante absoluto hasta el aprendizaje de los conceptos avanzados necesarios para desarrollar aplicaciones listas para la producción. También es para personas familiarizadas con la programación que desean aprender el marco de trabajo de Flutter y el lenguaje Dart.

Este libro está escrito con el supuesto de no tener experiencia previa en programación, Flutter o Dart. Si has programado en otros lenguajes o estás familiarizado con Flutter y Dart, simplemente obtendrás una comprensión más profunda de cada concepto y técnica.

LO QUE CUBRE ESTE LIBRO

Los primeros capítulos introducen y cubren la arquitectura del marco Flutter, el lenguaje Dart y los pasos para crear un nuevo proyecto. Usará ese conocimiento para crear nuevos proyectos para cada ejercicio del libro. Cada capítulo está escrito para avanzar en su conocimiento centrándose en nuevos conceptos.

Los capítulos también están escritos como material de referencia para refrescar su conocimiento de cada concepto.

A partir del Capítulo 2, a medida que aprenda cada concepto y técnica, seguirá los ejercicios de estilo de práctica "Pruébelo" y creará nuevos proyectos de aplicación para poner en práctica lo que ha aprendido.

A medida que avanza, cada capítulo está diseñado para enseñarle temas más avanzados. Los últimos cuatro capítulos se centran en la creación de dos aplicaciones listas para producción mediante la aplicación de materiales aprendidos previamente y la implementación de nuevos conceptos avanzados. También puede encontrar estos ejercicios como parte de las descargas de códigos en la página de este libro en www.wiley.com/go/beginningflutter.

CÓMO ESTÁ ESTRUCTURADO ESTE LIBRO

Este libro está dividido en 16 capítulos. Aunque cada capítulo se basa en los conceptos anteriores, también son independientes y están escritos para que usted pueda saltar a un interés particular para aprender o actualizar ese tema.

Parte I: Los fundamentos de la programación de Flutter En la primera parte del libro, conocerá los aspectos centrales de Flutter para que tenga una base sólida sobre la cual construir.

Capítulo 1: Introducción a Flutter y primeros pasos: aprenderá cómo funciona el marco de Flutter detrás de escena y sobre los beneficios del lenguaje Dart. Verá cómo se relacionan Widget, Element y RenderObject , y comprenderá cómo forman el árbol de widgets, el árbol de elementos y el árbol de renderizado. Obtendrá una introducción a StatelessWidget y StatefulWidget y sus eventos de ciclo de vida. Aprenderá que Flutter es declarativo, lo que significa que Flutter crea la interfaz de usuario para reflejar el estado de la aplicación. Aprenderá a instalar el marco Flutter, Dart, el editor y los complementos en macOS, Windows o Linux.

Capítulo 2: Creación de una aplicación Hello World: aprenderá cómo crear su primer proyecto de Flutter para familiarizarse con el proceso. Al escribir este ejemplo mínimo, aprenderá la estructura básica de una aplicación, cómo ejecutar la aplicación en el simulador de iOS y el emulador de Android, y cómo realizar cambios en el código. En este punto, no te preocupes por entender el código todavía; Lo guiaré paso a paso en capítulos posteriores.

Capítulo 3: Aprendizaje de los conceptos básicos de Dart: Dart es la base para aprender a desarrollar aplicaciones de Flutter, y en este capítulo comprenderá la estructura básica de Dart. Aprenderá cómo comentar su código, cómo la función main() inicia la aplicación, cómo declarar variables y cómo usar la Lista para almacenar una matriz de valores. Aprenderá sobre los símbolos de los operadores y cómo usarlos para realizar notaciones aritméticas, de igualdad, lógicas, condicionales y en cascada.

Aprenderá cómo usar paquetes y clases externos y cómo usar la declaración de importación . Aprenderá a implementar la programación asíncrona mediante el uso de un objeto Future . Aprenderá cómo crear clases para agrupar la lógica del código y usar variables para almacenar datos y cómo definir funciones para ejecutar la lógica.

Capítulo 4: Creación de una plantilla de proyecto inicial: aprenderá los pasos para crear un nuevo proyecto que usará y replicará para crear todos los ejercicios de este libro. Aprenderá a organizar archivos y carpetas en su proyecto. Creará los nombres más utilizados para agrupar sus widgets, clases y archivos por el tipo de acción necesaria. Aprenderá a estructurar widgets e importar paquetes y bibliotecas externos.

Capítulo 5: Descripción del árbol de widgets: el árbol de widgets es el resultado de componer (anidar) widgets para crear diseños simples y complejos. A medida que comienza a anidar widgets, el código puede volverse más difícil de seguir, por lo que una buena práctica es tratar de mantener el árbol de widgets lo más superficial posible. Obtendrá una introducción a los widgets que utilizará en este capítulo. Obtendrá una comprensión de los efectos de un árbol de widgets profundo y aprenderá a refactorizarlo en un árbol de widgets poco profundo, lo que dará como resultado un código más manejable. Aprenderá tres formas de crear un árbol de widgets superficial refactorizando con una constante, con un método y con una clase de v

Aprenderás los beneficios y las contras de cada técnica.

Parte II: Flutter intermedio: desarrollar una aplicación

En la Parte II del libro, se ensuciará las manos y explicará paso a paso cómo agregar funciones que crean excelentes experiencias de usuario.

Introducción

Capítulo 6: Uso de widgets comunes: aprenderá a usar los widgets más comunes, que son los componentes básicos para crear una interfaz de usuario y una experiencia de usuario (UX) atractivas. Aprenderá a cargar imágenes desde el paquete de activos de la aplicación y en la Web a través de un localizador uniforme de recursos (URL). Aprenderá a usar los íconos de componentes de material incluidos y cómo aplicar decoradores para mejorar la apariencia de los widgets o usarlos como guías de entrada para los campos de entrada. Aprenderá a usar el widget de formulario para validar widgets de entrada de campos de texto como grupo. Aprenderá diferentes formas de detectar la orientación para diseñar los widgets según corresponda, dependiendo de si el dispositivo está en modo vertical u horizontal.

Capítulo 7: Adición de animación a una aplicación: aprenderá cómo agregar animación a una aplicación para transmitir acción. Cuando la animación se usa adecuadamente, mejora la UX, pero demasiadas animaciones o innecesarias pueden empeorar la UX. Aprenderá a crear animaciones Tween . Aprenderá a utilizar las animaciones integradas mediante los widgets AnimatedContainer, AnimatedCrossFade y AnimatedOpacity . Aprenderá a crear animaciones personalizadas mediante las clases AnimationController y AnimatedBuilder . Aprenderá a crear animaciones escalonadas mediante el uso de varias clases de animación . Aprenderá a usar la clase CurvedAnimation para efectos no lineales.

Capítulo 8: Creación de la navegación de una aplicación: aprenderá que una buena navegación crea una gran UX, lo que facilita el acceso a la información. Aprenderá que agregar animación mientras navega a otra página también puede mejorar la UX siempre que transmita una acción, en lugar de ser una distracción. Aprenderá a usar el widget Navigator para administrar una pila de rutas para moverse entre páginas. Aprenderá a usar el widget Hero para transmitir una animación de navegación para mover y cambiar el tamaño de un widget de una página a otra. Aprenderá diferentes formas de agregar navegación mediante los widgets BottomNavigationBar, BottomAppBar, TabBar, Tab BarView y Drawer . También aprenderá a usar el widget ListView junto con el widget Drawer para crear una lista de elementos del menú de navegación.

Capítulo 9: Creación de listas de desplazamiento y efectos: aprenderá a utilizar diferentes widgets para crear una lista de desplazamiento para ayudar a los usuarios a ver y seleccionar información. Aprenderá a usar el widget Tarjeta para agrupar la información utilizada junto con los widgets de lista de desplazamiento. Aprenderá a usar el widget ListView para crear una lista lineal de widgets desplazables. Aprenderá a usar GridView para mostrar mosaicos de widgets desplazables en formato de cuadrícula. Aprenderá a utilizar el widget Stack para superponer, colocar y alinear sus widgets secundarios junto con una lista de desplazamiento. Aprenderá a implementar CustomScrollView para crear efectos de desplazamiento personalizados, como animación de paralaje, mediante el uso de widgets como SliverSafeArea, SliverAppBar, SliverList, SliverGrid y más.

Capítulo 10: Creación de diseños: aprenderá a anidar widgets para crear diseños profesionales. Este concepto es una parte importante de la creación de hermosos diseños y se conoce como composición. Los diseños básicos y complejos se basan principalmente en widgets verticales u horizontales o en una combinación de ambos. El objetivo de este capítulo es crear una página de entrada de diario que muestre detalles como una imagen de encabezado, título, detalles del diario, clima, dirección (ubicación del diario), etiquetas e imágenes de pie de página. Para diseñar la página, utilizará widgets como SingleChildScrollView, SafeArea, Padding, Column, Row, Image, Divider, Text, Icon, SizedBox, Wrap , Chip y CirculoAvatar.

Capítulo 11: Aplicación de interactividad: aprenderá cómo agregar interactividad a una aplicación mediante gestos. En las aplicaciones móviles, los gestos son el corazón para escuchar la interacción del usuario, y hacer uso de los gestos puede dar como resultado una aplicación con una gran experiencia de usuario. Sin embargo, el uso excesivo de gestos

sin agregar valor al transmitir una acción puede crear una experiencia de usuario deficiente. Aprenderá a usar los gestos de GestureDetector , como tocar, tocar dos veces, presionar prolongadamente, desplazarse, arrastrar verticalmente, arrastrar horizontalmente y escalar. Aprenderá a usar el widget Draggable para arrastrar sobre el widget DragTarget para crear un efecto de arrastrar y soltar para cambiar el color del widget. Aprenderá a implementar los widgets InkWell e InkResponse que responden al tacto y muestran visualmente una animación de bienvenida. Aprenderá a implementar el widget Descartable que se descarta arrastrando. Aprenderá a usar el widget Transform y la clase Matrix4 para escalar y mover widgets.

Capítulo 12: Escritura de código nativo de la plataforma: en algunos casos, debe acceder a la funcionalidad específica de la API de iOS o Android, y aprenderá a usar los canales de la plataforma para enviar y recibir mensajes entre la aplicación Flutter y la plataforma host. Aprenderá a usar Method Channel para enviar mensajes desde la aplicación Flutter (lado del cliente) y FlutterMethodChannel en iOS y MethodChannel en Android para recibir llamadas (lado del host) y devolver los resultados.

Parte III: Creación de aplicaciones listas para producción

Para los últimos cuatro capítulos del libro, se moverá a un territorio más avanzado y se preparará para lanzar sus aplicaciones de muestra a producción.

Capítulo 13: Guardar datos con persistencia local: aprenderá a crear una aplicación de diario. Aprenderá a conservar los datos durante los inicios de aplicaciones utilizando el formato de archivo JSON y guardando el archivo en el sistema de archivos local de iOS y Android. La notación de objetos de JavaScript (JSON) es un formato de datos de archivo estándar abierto e independiente del idioma común con los beneficios de ser texto legible por humanos. Aprenderá a crear clases de bases de datos para escribir, leer y serializar archivos JSON. Aprenderá cómo dar formato a una lista y ordenarla por fecha.

En una aplicación móvil, es importante no bloquear la interfaz de usuario durante el procesamiento y aprenderá a usar la clase Future y el widget FutureBuilder . Aprenderá cómo presentar un calendario de selección de fechas, validar los datos de entrada del usuario y mover el foco entre los campos de entrada.

También aprenderá a eliminar registros utilizando el widget Descartable arrastrando o arrojando una entrada. Para ordenar las entradas por fecha, aprenderá a usar el método List().sort y la función Comparator . Para navegar entre páginas, usará el widget Navigator y aprenderá a usar el widget CircularProgressIndicator para mostrar que se está ejecutando una acción.

Capítulo 14: Adición del backend de Firebase y Firestore. A lo largo de este capítulo, el Capítulo 15 y el Capítulo 16, utilizará las técnicas que aprendió en los capítulos anteriores junto con nuevos conceptos y los unirá para crear una aplicación de registro de estados de ánimo a nivel de producción.. En una aplicación de nivel de producción, ¿cómo combinaría lo que aprendió, mejoraría el rendimiento al volver a dibujar solo los widgets con cambios de datos, pasar el estado entre páginas y subir el árbol de widgets, manejar las credenciales de autenticación del usuario, sincronizar datos entre dispositivos y la nube, y crear clases que manejen la lógica independiente de la plataforma entre aplicaciones móviles y web? Estas son las razones por las que estos últimos tres capítulos le enseñarán cómo aplicar las técnicas anteriores que aprendió junto con otras nuevas e importantes para desarrollar una aplicación móvil de nivel de producción.

En estos últimos tres capítulos, aprenderá cómo implementar la gestión de estado local y en toda la aplicación y maximizar el uso compartido de código de plataforma mediante la implementación del patrón Business Logic Component (BLoC).

Introducción

Aprenderá a usar la autenticación y conservar los datos en una base de datos en la nube mediante la infraestructura del servidor de back-end de Firebase de Google, la autenticación de Firebase y Cloud Firestore. Aprenderá que Cloud Firestore es una base de datos de documentos NoSQL para almacenar, consultar y sincronizar datos con soporte sin conexión para aplicaciones móviles y web. Podrás sincronizar datos entre múltiples dispositivos. Aprenderá a configurar y crear aplicaciones sin servidor.

Capítulo 15: Agregar administración de estado a la aplicación Firestore Client: continuará editando la aplicación de registro de estado de ánimo creada en el Capítulo 14. Aprenderá a crear una administración de estado local y de toda la aplicación que usa la clase InheritedWidget como proveedor para administrar y pasar el estado entre widgets y páginas.

Aprenderá a usar el patrón BLoC para crear clases BLoC, por ejemplo, administrar el acceso a las clases de servicio de base de datos de Firebase Authentication y Cloud Firestore. Aprenderá a usar la clase InheritedWidget para pasar una referencia entre el BLoC y las páginas. Aprenderá a usar el enfoque reactivo usando StreamBuilder, StreamCon troller y Stream para completar y actualizar datos.

Aprenderá a crear clases de servicio para administrar la API de autenticación de Firebase y la API de la base de datos de Cloud Firestore. Creará y aprovechará una clase abstracta para administrar las credenciales de usuario. Aprenderá a crear una clase de modelo de datos para manejar la asignación de Cloud Firestore QuerySnapshot a registros individuales. Aprenderá a crear una clase para administrar una lista de iconos de estado de ánimo, descripción y posición de rotación de iconos según el estado de ánimo seleccionado. Utilizará el paquete intl y aprenderá a crear una clase de formato de fecha.

Capítulo 16: Adición de BLoC a las páginas de la aplicación Firestore Client: continuará editando la aplicación de registro de estado de ánimo creada en el Capítulo 14 con las adiciones del Capítulo 15.

Aprenderá a aplicar las clases BLoC, servicio, proveedor, modelo y utilidad a las páginas de widgets de la interfaz de usuario. El beneficio de usar el patrón BLoC permite separar los widgets de la interfaz de usuario de la lógica comercial. Aprenderá a usar la inyección de dependencia para inyectar clases de servicio en las clases de BLoC. Mediante el uso de inyección de dependencia, los BLoC siguen siendo independientes de la plataforma. Este concepto es extremadamente importante ya que el marco Flutter se está expandiendo más allá de los dispositivos móviles y hacia la web, el escritorio y los dispositivos integrados.

Aprenderá a aplicar la administración del estado de autenticación en toda la aplicación mediante la implementación de clases que aplican el patrón BLoC. Aprenderá a crear una página de inicio de sesión que implemente la clase de patrón BLoC para validar correos electrónicos, contraseñas y credenciales de usuario. Aprenderá a pasar el estado entre las páginas y el árbol de widgets mediante la implementación de clases de proveedores (Inherit edWidget). Aprenderá a modificar la página de inicio para implementar y crear clases de patrón BLoC para manejar la validación de credenciales de inicio de sesión, crear una lista de entradas de diario y agregar y eliminar entradas individuales. Aprenderá a crear la página de edición del diario que implementa las clases de patrón BLoC para agregar, modificar y guardar entradas existentes.

QUE NECESITAS PARA UTILIZAR ESTE LIBRO

Deberá instalar Flutter Framework y Dart para crear los proyectos de ejemplo. Este libro usa Android Studio como su principal herramienta de desarrollo y todos los proyectos se compilan para iOS y Android. Para compilar aplicaciones de iOS, necesitará una computadora Mac con Xcode instalado. También puedes usar otros

editores como Microsoft Visual Studio Code o IntelliJ IDEA. Para el último gran proyecto, deberá crear una cuenta gratuita de Google Firebase para aprovechar la autenticación en la nube y la sincronización de datos, incluido el soporte sin conexión.

El código fuente de las muestras está disponible para su descarga desde www.wiley.com/go/beginningflutter.

CONVENCIONES

Para ayudarlo a aprovechar al máximo el texto y realizar un seguimiento de lo que sucede, hemos utilizado una serie de convenciones a lo largo del libro.

PRUÉBALO Debes resolver todos los ejercicios de Pruébalo del libro.

1. Estos ejercicios consisten en un conjunto de pasos numerados.
2. Siga los pasos con su copia de la base de datos.

CÓMO FUNCIONA

Al final de cada Pruébalo, se le explicará en detalle el código que ha escrito.

En cuanto a los estilos en el texto:

Ponemos en cursiva los términos nuevos y las palabras importantes cuando los presentamos.

Mostramos pulsaciones de teclado como esta: Ctrl+A.

Mostramos nombres de archivo, URL y código dentro del texto de la siguiente manera:

`persistencia.propiedades`

Presentamos el código de dos maneras diferentes:

Usamos un tipo monofuente sin resaltado para la mayoría de los ejemplos de código.

Usamos negrita para enfatizar el código que es particularmente importante en el contexto actual o para mostrar cambios de un fragmento de código anterior.

FE DE ERRATAS

Hacemos todo lo posible para asegurarnos de que no haya errores en el texto o en el código. Sin embargo, nadie es perfecto y los errores ocurren. Si encuentra un error en uno de nuestros libros, como un error de ortografía o una pieza de código defectuosa, estaremos muy agradecidos por sus comentarios. Al enviar erratas, puede ahorrarle a otro lector horas de frustración y, al mismo tiempo, nos ayudará a brindar información de mayor calidad.

Introducción

Para encontrar la página de errata de este libro, vaya a la página del libro en www.wiley.com/go/beginningflutter y haga clic en el enlace Errata. En esta página, puede ver todas las erratas enviadas para este libro y publicadas por los editores de Wrox.

Si no detecta "su" error en la página de errata del libro, envíe un correo electrónico a nuestro equipo de atención al cliente a wileysupport@wiley.com con el asunto "Posible envío de errata del libro". Verificaremos la información y, si corresponde, publicaremos un mensaje en la página de errata del libro y solucionaremos el problema en ediciones posteriores del libro.

PARTE I

Los fundamentos del aleteo Programación

CAPÍTULO 1: Introducción a Flutter y primeros pasos

CAPÍTULO 2: Creación de una aplicación Hello World

CAPÍTULO 3: Aprendiendo los conceptos básicos de Dart

CAPÍTULO 4: Creación de una plantilla de proyecto inicial

CAPÍTULO 5: Comprender el árbol de widgets

1

Presentamos Flutter y Empezando

LO QUE APRENDERÁS EN ESTE CAPÍTULO

- Qué es el framework Flutter
- Cuáles son los beneficios de Flutter
- Cómo funcionan juntos Flutter y Dart
- Qué es un widget Flutter
- Qué es un elemento
- Qué es un RenderObject
- Qué tipo de widgets Flutter están disponibles
- Cuál es el ciclo de vida del widget sin estado y con estado
- Cómo funcionan juntos el árbol de widgets y el árbol de elementos
- Cómo instalar el SDK de Flutter
- Cómo instalar Xcode en macOS y Android Studio en macOS, Windows y Linux
- Cómo configurar un editor
- Cómo instalar los complementos Flutter y Dart

En este capítulo, aprenderá cómo funciona el marco Flutter detrás de escena. Flutter usa widgets para crear la interfaz de usuario (UI), y Dart es el lenguaje que se usa para desarrollar las aplicaciones. Una vez que comprenda cómo Flutter maneja e implementa los widgets, lo ayudará a diseñar sus aplicaciones.

Aprenderá a instalar Flutter SDK en macOS, Windows y Linux. Configurará Android Studio para instalar el complemento Flutter para ejecutar, depurar y usar la recarga en caliente. Instalará el complemento Dart para el análisis de código, la validación de código y la finalización de código.

INTRODUCCIÓN AL FLUTTER

Flutter es el marco de interfaz de usuario portátil de Google para crear aplicaciones modernas, nativas y reactivas para iOS y Android. Google también está trabajando en la incrustación de escritorio Flutter y Flutter para la Web (Hummingbird) y dispositivos integrados (Raspberry Pi, hogar, automóvil y más). Flutter es un proyecto de código abierto alojado en GitHub con contribuciones de Google y la comunidad. Flutter usa Dart, un lenguaje moderno orientado a objetos que se compila en código ARM nativo y código JavaScript listo para producción. Flutter usa el motor de renderizado Skia 2D que funciona con diferentes tipos de plataformas de hardware y software y también lo usan Google Chrome, Chrome OS, Android, Mozilla Firefox, Firefox OS y otros. Skia está patrocinado y administrado por Google y está disponible para que cualquiera lo use bajo la licencia de software libre BSD. Skia utiliza un procesamiento de ruta basado en CPU y también es compatible con el backend acelerado por OpenGL ES2.

Dart es el lenguaje que usará para desarrollar sus aplicaciones Flutter, y aprenderá más sobre él en el Capítulo 3, "Aprender los conceptos básicos de Dart". Dart se compila con anticipación (AOT) en código nativo, lo que hace que su aplicación Flutter sea más rápida. Dart también se compila justo a tiempo (JIT), lo que agiliza la visualización de los cambios de código, como a través de la función de recarga en caliente con estado de Flutter.

Flutter usa Dart para crear su interfaz de usuario, eliminando la necesidad de usar lenguajes separados como Marcado o diseñadores visuales. Flutter es declarativo; en otras palabras, Flutter crea la interfaz de usuario para reflejar el estado de la aplicación. Cuando el estado (datos) cambia, la interfaz de usuario se vuelve a dibujar y Flutter construye una nueva instancia del widget. En la sección "Comprender el árbol de widgets y el árbol de elementos" de este capítulo, aprenderá cómo se configuran y montan (renderizan) los widgets creando el árbol de widgets y el árbol de elementos, pero debajo del capó, el árbol de renderizado (un tercer árbol) utiliza RenderObject, que calcula e implementa los protocolos básicos de diseño y pintura. (No necesitará interactuar directamente con el árbol de renderizado o el RenderObject, y no los discutiré más en este libro).

Flutter es rápido y el renderizado se ejecuta a 60 cuadros por segundo (fps) y 120 fps para dispositivos compatibles. Cuanto más altos sean los fps, más suaves serán las animaciones y las transiciones.

Las aplicaciones creadas en Flutter se construyen a partir de una única base de código, se compilan en código ARM nativo, usan la unidad de procesamiento de gráficos (GPU) y pueden acceder a API específicas de iOS y Android (como ubicación GPS, biblioteca de imágenes) comunicándose a través de los canales de la plataforma. Aprenderá más sobre los canales de la plataforma en el Capítulo 12, "Escribir código nativo de la plataforma".

Flutter proporciona al desarrollador herramientas para crear aplicaciones hermosas y de aspecto profesional y la capacidad de personalizar cualquier aspecto de la aplicación. Podrá agregar animaciones fluidas, detección de gestos y comportamiento de retroalimentación de salpicaduras a la interfaz de usuario. Las aplicaciones Flutter dan como resultado un rendimiento nativo para las plataformas iOS y Android. Durante el desarrollo, Flutter utiliza la recarga en caliente para actualizar la aplicación en ejecución en milisegundos cuando cambia el código fuente para agregar nuevas funciones o modificar las existentes. El uso de la recarga en caliente es una excelente manera de ver los cambios que realiza en su código en el simulador o dispositivo mientras mantiene el estado de la aplicación, los valores de datos, en la pantalla.

Definición de widgets y elementos La interfaz de usuario

de Flutter se implementa mediante el uso de widgets de un marco reactivo moderno. Flutter usa su propio motor de renderizado para dibujar widgets. En el Capítulo 5, "Comprensión del árbol de widgets", obtendrá una introducción a los widgets, y en el Capítulo 6, "Uso de widgets comunes", aprenderá a implementar widgets.

Te estarás preguntando, ¿qué es un widget? Los widgets se pueden comparar con bloques LEGO; al agregar bloques, crea un objeto, y al agregar diferentes tipos de bloques, puede alterar la apariencia y el comportamiento del objeto. Los widgets son los componentes básicos de una aplicación Flutter, y cada widget es una declaración de tabla immu de la interfaz de usuario. En otras palabras, los widgets son configuraciones (instrucciones) para diferentes partes de la interfaz de usuario. Al colocar los widgets juntos, se crea el árbol de widgets. Por ejemplo, digamos que un arquitecto dibuja un plano de una casa; todos los objetos como paredes, ventanas y puertas de la casa son los widgets, y todos ellos trabajan juntos para crear la casa o, en este caso, la aplicación.

Dado que los widgets son las piezas de configuración de la interfaz de usuario y juntos crean el árbol de widgets, ¿cómo usa Flutter estas configuraciones? Flutter usa el widget como configuración para construir cada elemento, lo que significa que el elemento es el widget que se monta (representa) en la pantalla. Los elementos que se montan en la pantalla crean el árbol de elementos. Aprenderá más sobre el árbol de widgets y el árbol de elementos en la siguiente sección, "Comprensión del árbol de widgets y el árbol de elementos". También aprenderá a manipular el árbol de widgets en detalle en el Capítulo 5.

He aquí un breve vistazo a la amplia gama de widgets a su disposición:

Widgets con elementos estructurantes como una lista, cuadrícula, texto y botón

Widgets con elementos de entrada como un formulario, campos de formulario y detectores de teclado

Widgets con elementos de estilo como tipo de fuente, tamaño, grosor, color, borde y sombra

Widgets para diseñar la interfaz de usuario, como fila, columna, pila, centrado y relleno

Widgets con elementos interactivos que responden al tacto, gestos, arrastrar y descartar

Widgets con elementos de animación y movimiento, como animación de héroe, contenedor animado, fundido cruzado animado, transición de fundido, rotación, escala, tamaño, deslizamiento y opacidad

Widgets con elementos como activos, imágenes e íconos

Widgets que se pueden anidar para crear la interfaz de usuario necesaria

Widgets personalizados que puede crear usted mismo

COMPRENSIÓN DE LOS EVENTOS DEL CICLO DE VIDA DEL WIDGET

En la programación, tiene diferentes eventos del ciclo de vida que generalmente ocurren en un modo lineal, uno tras otro a medida que se completa cada etapa. En esta sección, aprenderá los eventos del ciclo de vida del widget y su propósito.

Para crear la interfaz de usuario, utiliza dos tipos principales de widgets, StatelessWidget y StatefulWidget. Un widget sin estado se usa cuando los valores (estado) no cambian, y el widget con estado se usa cuando

los valores (estado) cambian. En el Capítulo 2, "Creación de una aplicación Hello World", aprenderá en detalle cuándo usar un StatelessWidget o un StatefulWidget. Cada widget sin estado o con estado tiene un método de compilación con un BuildContext que maneja la ubicación del widget en el árbol de widgets. Los objetos BuildContext son en realidad objetos Element , una instancia del Widget en una ubicación en el árbol.

El ciclo de vida del widget sin estado

Un StatelessWidget se construye en base a su propia configuración y no cambia dinámicamente. Por ejemplo, la pantalla muestra una imagen con una descripción y no cambiará. El widget sin estado se declara con una clase, y aprenderá sobre las clases en el Capítulo 3. El método de compilación (las partes de la interfaz de usuario) del widget sin estado se puede llamar desde tres escenarios diferentes. Se puede llamar la primera vez que se crea el widget, cuando cambia el padre del widget y cuando cambia un InheritedWidget . En el Capítulo 15, "Agregar administración de estado a la aplicación Firestore Client", aprenderá a implementar InheritedWidget.

El siguiente código de ejemplo muestra una estructura base StatelessWidget y la Figura 1.1 muestra el ciclo de vida del widget.

```
clase JournalList extiende StatelessWidget { @override

    Compilación del widget (contexto BuildContext) {
        devolver Contenedor();

    }
}
```



FIGURA 1.1: Ciclo de vida del widget sin estado

El ciclo de vida de StatefulWidget

Un StatefulWidget se crea en función de su propia configuración, pero puede cambiar dinámicamente. Por ejemplo, la pantalla muestra un ícono con una descripción, pero los valores pueden cambiar según la interacción del usuario, como elegir un ícono o descripción diferente. Este tipo de widget tiene un estado mutable que puede cambiar con el tiempo. El widget con estado se declara con dos clases, la clase StatefulWidget y la clase State . La clase StatefulWidget se reconstruye cuando cambia la configuración del widget, pero la clase State puede persistir (permanecer), mejorando el rendimiento. Por ejemplo, cuando cambia el estado, el widget se reconstruye. Si StatefulWidget se elimina del árbol y luego se vuelve a insertar en el futuro, se crea un nuevo objeto State . Tenga en cuenta que bajo ciertas circunstancias y restricciones, puede usar una GlobalKey (clave única en toda la aplicación) para reutilizar (no volver a crear) el objeto State ; sin embargo, las claves globales son costosas y, a menos que sean necesarias, es posible que desee considerar no usarlas. Llama al método setState() para notificar al marco que este objeto tiene cambios, y se llama al método de compilación del widget (programado). Establecería los nuevos valores de estado dentro del método setState() . En el Capítulo 2, aprenderá cómo llamar al método setState() .

El siguiente ejemplo muestra una estructura base StatefulWidget y la Figura 1.2 muestra el ciclo de vida del widget. Tiene dos clases, la clase JournalEdit StatefulWidget y la clase _JournalEditState .

```
clase JournalEdit extiende StatefulWidget { @override

    _ JournalEditState createState() =>
        _ DiarioEditarEstado();
```

```

}

class _JournalEditState extiende Estado<JournalEdit> {
    @anular
    Compilación del widget (contexto BuildContext) {
        devolver Contenedor();
    }
}

```

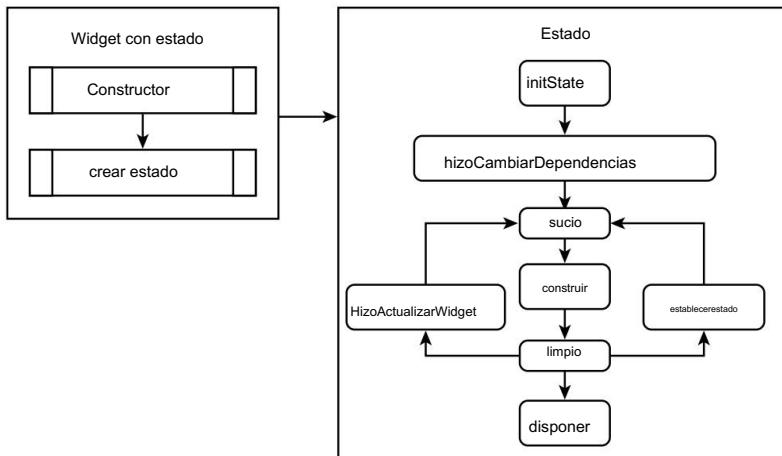


FIGURA 1.2: Ciclo de vida de StatefulWidget

Puede anular diferentes partes de StatefulWidget para personalizar y manipular datos en diferentes puntos del ciclo de vida del widget. La Tabla 1.1 muestra algunas de las anulaciones principales del widget con estado, y la mayoría de las veces utilizará los métodos `initState()`, `didChangeDependencies()` y `dispose()`. Usarás el método `build()` todo el tiempo para construir tu interfaz de usuario.

TABLA 1.1: Ciclo de vida de StatefulWidget

MÉTODO	DESCRIPCIÓN	CÓDIGO DE MUESTRA
<code>initState()</code>	Llamado una vez cuando este objeto se inserta en el árbol.	@anular void initState() { super.initState(); imprimir('estadoinicial'); }
<code>disponer()</code>	Se llama cuando este objeto se elimina del árbol de forma permanente.	@anular anular disponer () { imprimir('desechar'); super.dispose(); }

continúa

TABLA 1.1 (continuación)

MÉTODO	DESCRIPCIÓN	CÓDIGO DE MUESTRA
hizoCambiarDependencias()	Se llama cuando cambia este objeto State .	@anular void hizoCambiarDependencias() { super.didChangeDependencies(); print('didChangeDependencies'); }
HizoActualizarWidget (Antiguo widget de contactos)	Se llama cuando cambia la configuración del widget.	@override void didUpdateWidget(Contactos oldWidget) { super.didUpdateWidget(antiguoWidget); print('hizoActualizarWidget: \$antiguoWidget'); }
desactivar()	Se llama cuando este objeto se elimina del árbol.	@anular void desactivar() { imprimir('desactivar'); super.desactivar(); }
build(BuildContext contexto)	Se puede llamar varias veces para crear la interfaz de usuario y BuildContext controla la ubicación de este widget en el árbol de widgets.	@anular Widget compilación (contextoBuildContext) { print ('construir'); devolver Contenedor(); }
establecerEstado()	Notifica al marco que el estado de este objeto ha cambiado para programar la llamada a la compilación para este objeto State .	setState(() { nombre = _nuevoValor; });

ENTENDIENDO EL ÁRBOL DE WIDGET Y EL ÁRBOL DE ELEMENTOS

En la sección anterior, aprendió que los widgets contienen las instrucciones para crear la interfaz de usuario, y cuando compone (anida) widgets juntos, crean el árbol de widgets. El marco Flutter usa los widgets como configuraciones para cada elemento que se monta (representa) en la pantalla. el montado

los elementos que se muestran en la pantalla crean el árbol de elementos. Ahora tiene dos árboles, el árbol de widgets que tiene las configuraciones de widgets y el árbol de elementos que representa los widgets representados en la pantalla (Figura 1.3).

Cuando se inicia la aplicación, la función main() llama al método runApp() , por lo general tomando un estado lessWidget como argumento, y se monta como el elemento raíz de la aplicación. El marco Flutter procesa todos los widgets y se monta cada elemento correspondiente.

El siguiente es un código de muestra que inicia una aplicación Flutter, y el método runApp() infla MyApp StatelessWidget, lo que significa que la aplicación principal en sí misma es un widget. Como puedes ver, casi todo en Flutter es un widget.

```
void main() => runApp(MiAplicación());  
  
clase MyApp extiende StatelessWidget { @override  
  
    Complilación del widget (contexto BuildContext) {  
        devolver MaterialApp(  
            título: 'Flutter App', tema:  
            ThemeData  
                (primarySwatch: Colors.blue, ), inicio:  
  
            MyHomePage (título: 'Inicio'), );  
  
    }  
}
```

Los elementos tienen una referencia al widget y son responsables de comparar las diferencias del widget. Si un widget es responsable de crear widgets secundarios, se crean elementos para cada widget secundario. Cuando ve el uso de objetos BuildContext , son los objetos Element .

Para desalentar la manipulación directa de los objetos Element , se utiliza en su lugar la interfaz BuildContext . El marco Flutter usa objetos BuildContext para disuadirte de manipular los objetos Element . En otras palabras, usará widgets para crear sus diseños de interfaz de usuario, pero es bueno saber cómo está diseñado el marco Flutter y cómo funciona detrás de escena.

Como se señaló anteriormente, hay un tercer árbol llamado árbol de renderizado que es un sistema de diseño y pintura de bajo nivel que hereda del RenderObject. RenderObject calcula e implementa los protocolos básicos de diseño y pintura . Sin embargo, no necesitará interactuar directamente con el árbol de representación y utilizará los widgets.

Widget sin estado y árboles de elementos

Un widget sin estado tiene la configuración para crear un elemento sin estado. Cada widget sin estado tiene un elemento sin estado correspondiente. El marco Flutter llama al método createElement (crear una instancia), y el elemento sin estado se crea y se monta en el árbol de elementos. En otra

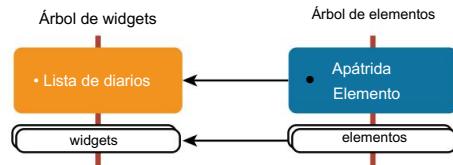


FIGURA 1.3: Árbol de widgets y árbol de elementos

En otras palabras, el marco Flutter realiza una solicitud desde el widget para crear un elemento y luego monta (agrega) el elemento al árbol de elementos. Cada elemento contiene una referencia al widget. El elemento llama al método de compilación del widget para buscar widgets secundarios, y cada widget secundario (como un ícono o texto) crea su propio elemento y se monta en el árbol de elementos. Este proceso da como resultado dos árboles: el árbol de widgets y el árbol de elementos.

La Figura 1.4 muestra el `JournalList StatelessWidget` que tiene widgets de Fila, Ícono y Texto que representan el árbol de widgets. El marco Flutter le pide a cada widget que cree el elemento, y cada elemento tiene una referencia al widget. Este proceso ocurre para cada widget en el árbol de widgets y crea el árbol de elementos. El widget contiene las instrucciones para construir el elemento montado en la pantalla. Tenga en cuenta que usted, el desarrollador, crea los widgets, y el marco Flutter maneja el montaje de los elementos, creando el árbol de elementos.

```
// La clase de código de muestra
simplificada JournalList extiende StatelessWidget { @override

    Compilación del widget (contexto BuildContext) {
        return Fila(niños:
            <Widget>[
                Ícono(),
                Texto(), ], );
```

```
}
```

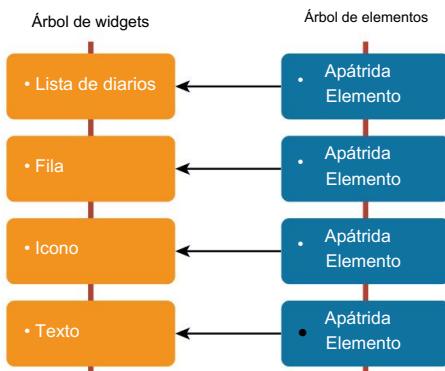


FIGURA 1.4: Árbol de widgets y árbol de elementos
 `StatelessWidget`

Widget con estado y árboles de elementos

Un widget con estado tiene la configuración para crear un elemento con estado. Cada widget con estado tiene un elemento con estado correspondiente. El marco Flutter llama al método `createElement` para crear el elemento con estado, y el elemento con estado se monta en el árbol de elementos. Dado que se trata de un widget con estado, el elemento con estado solicita que el widget cree un objeto de estado llamando al método `createState` de la clase `StatefulWidget`.

El elemento con estado ahora tiene una referencia al objeto de estado y al widget en la ubicación dada en el árbol de elementos. El elemento con estado llama al método de compilación del widget de objeto de estado para buscar widgets secundarios, y cada widget secundario crea su propio elemento y se monta en el árbol de elementos. Este proceso da como resultado dos árboles: el árbol de widgets y el árbol de elementos. Tenga en cuenta que si un widget secundario que muestra el estado (nota del diario) es un widget sin estado como el widget de texto, entonces el elemento creado para este widget es un elemento sin estado. El objeto de estado mantiene una referencia al widget (`StatefulWidget`)

class) y también maneja la construcción del widget Text con el valor más reciente. La figura 1.5 muestra el árbol de widgets, el árbol de elementos y el objeto de estado. Tenga en cuenta que el elemento con estado tiene una referencia al widget con estado y al objeto de estado.

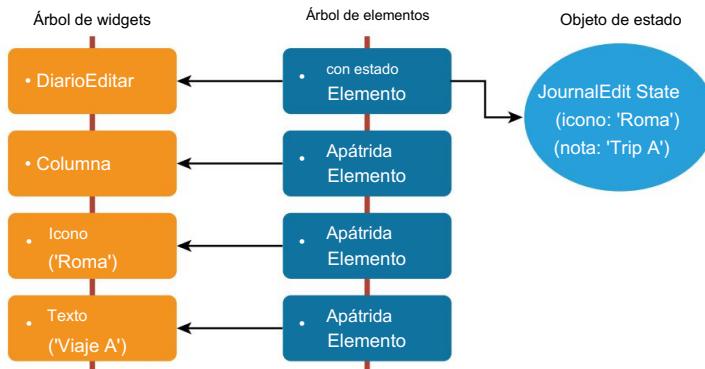


FIGURA 1.5: Árbol de widgets StatefulWidget, el árbol de elementos y el objeto de estado

Para actualizar la interfaz de usuario con nuevos datos, llame al método `setState()` que aprendió en la sección "El ciclo de vida de StatefulWidget". Para establecer nuevos valores de datos (propiedades/variables), llame al método `setState()` para actualizar el objeto de estado, y el objeto de estado marca el elemento como sucio (ha cambiado) y hace que la interfaz de usuario se actualice (programe). El elemento con estado llama al método de compilación del objeto de estado para reconstruir los widgets secundarios. Se crea un nuevo widget con el nuevo valor de estado y se elimina el antiguo widget.

Por ejemplo, tiene una clase StatefulWidget JournalEntry y en la clase de objeto State llama al método `setState()` para cambiar la descripción del widget Text de 'Trip A' a 'Trip B' configurando el valor de la variable de nota en 'Trip B'. La variable de nota del objeto de estado se actualiza al valor 'Trip B', el objeto de estado marca el elemento como sucio y el método de compilación reconstruye los widgets secundarios de la interfaz de usuario. El nuevo widget de Texto se crea con el nuevo valor 'Viaje B' y el antiguo widget de Texto con el valor 'Viaje A' se elimina (Figura 1.6).

```

// Clase de código de ejemplo
simplificado JournalEdit extiende StatefulWidget
{ @override
    _JournalEditState createState() => _JournalEditState();
}

class _JournalEditState extiende Estado<JournalEdit> {
    Nota de cadena = 'Viaje A';

    void _onPressed()
    { setState(() {
        nota = 'Viaje B';
    });
}

```

```

@anular
Compilación del widget (contexto BuildContext) { columna
de retorno (hijos:
<Widget>[
  Icono(),
  Texto('$nota'),
FlatButton( onPressed:
  _onPressed, ), ], );
}

```

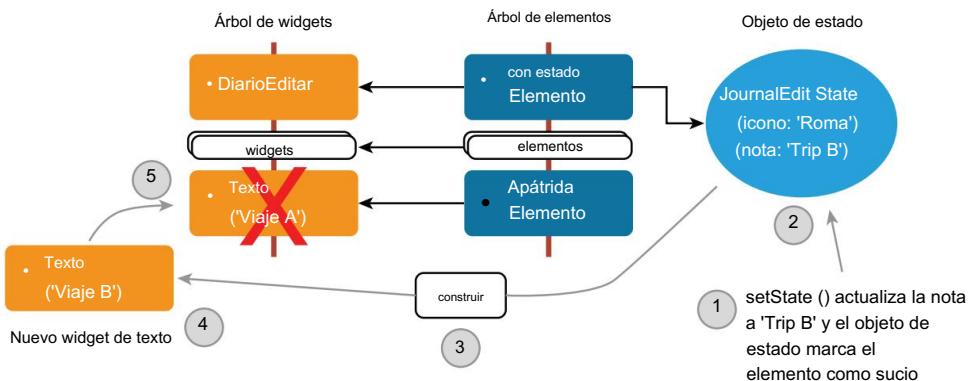


FIGURA 1.6: Actualización del proceso de estado

Dado que tanto el widget antiguo como el nuevo son widgets de texto , el elemento existente actualiza su referencia al nuevo widget y permanece en el árbol de elementos. El widget de texto es un widget sin estado y el elemento correspondiente es un elemento sin estado; aunque se reemplazó el widget de texto , el estado se mantiene (persiste). Los objetos de estado tienen una larga vida útil y permanecen adjuntos al árbol de elementos siempre que el nuevo widget sea del mismo tipo que el antiguo.

Sigamos con el ejemplo anterior; el antiguo widget de Texto con el valor 'Viaje A' se elimina y se reemplaza por el nuevo widget de Texto con el valor 'Viaje B' . Dado que tanto el widget antiguo como el nuevo son del mismo tipo de widgets de texto , el elemento permanece en el árbol de elementos con la referencia actualizada al nuevo widget de texto 'Trip B' (Figura 1.7).

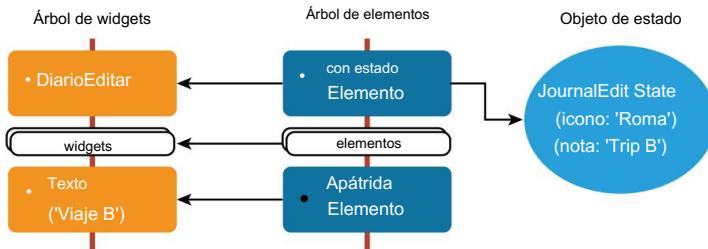


FIGURA 1.7: Estado actualizado para el Árbol de widgets y el Árbol de elementos

INSTALACIÓN DEL SDK DE FLUTTER

La instalación del SDK de Flutter requiere la descarga de la versión actual, que es 1.5.4 en el momento de escribir este artículo (su versión puede ser superior), del SDK del sitio web de Flutter. Este capítulo incluye secciones para la instalación en macOS, Windows y Linux. (Tenga en cuenta que la orientación y compilación para la plataforma iOS requiere una computadora Mac y Xcode, el entorno de desarrollo de Apple). No te desanimes por la cantidad de pasos; harás esto la primera vez que instales solamente.

Utilizará la ventana Terminal para ejecutar los comandos de instalación y configuración.

Instalación en macOS

Antes de comenzar la instalación, debe asegurarse de que su Mac admita los requisitos mínimos de hardware y software.

Requisitos del sistema macOS

(64 bits)

700 MB de espacio en disco (no incluye espacio en disco para el entorno de desarrollo integrado
ment y otras herramientas)

Las siguientes herramientas de línea de comandos:

```
golpe
mkdir
rm
git
rizo
descomprimir
que
```

Obtenga el SDK de Flutter

Los últimos detalles de instalación están disponibles en línea en el sitio web de Flutter en <https://flutter.dev/docs/get-started/install/macos/>. Ejecute los pasos 2 y siguientes desde la ventana de Terminal de su Mac.

1. Descargue el siguiente archivo de instalación para obtener la versión más reciente, v1.7.8 (es posible que su versión sea mayor), del SDK de Flutter:

```
https://storage.googleapis.com/flutter\_infra/releases/stable/macos/flutter\_macos\_v1.7.8+hotfix.4-stable.zip.
```

2. Extraiga el archivo en la ubicación deseada utilizando la ventana Terminal.

```
cd ~/descomprimir
desarrollo ~/Descargas/flutter_macos_v1.7.8+hotfix.4-stable.zip
```

3. Agregue la herramienta Flutter a su ruta (pwd significa directorio de trabajo actual; en su caso, será ser la carpeta de desarrollo).

exportar RUTA="\$RUTA:`pwd`/flutter/bin"

4. Actualice la ruta de forma permanente.

a. Obtenga la ruta que usó en el paso 3. Por ejemplo, reemplace MacUserName con su ruta a la carpeta de desarrollo.

/Usuarios/NombreUsuarioMac/desarrollo/flutter/bin

b. Abra o cree \$HOME/.bash_profile. Su ruta de archivo o nombre de archivo puede ser diferente en tu ordenador.

C. Edite .bash_profile (abrirá un editor de línea de comandos). d.

Escriba nano .bash_profile.

mi. Escriba export PATH=/Users/MacUserName/development/flutter/bin:\$PATH.

F. Guarde presionando ^X (Control+X), confirme presionando Y (Sí) y presione Entrar para aceptar el nombre del archivo.

gramo. Cierra la ventana de Terminal.

H. Vuelva a abrir la ventana de Terminal y escriba echo \$PATH para verificar que se haya agregado la ruta. Luego escriba flutter para asegurarse de que la ruta funcione. Si no se reconoce el comando, algo salió mal con la RUTA. Verifique que tenga el nombre de usuario correcto para su computadora en la ruta.

Comprobar dependencias

Ejecute el siguiente comando desde la ventana de Terminal para verificar las dependencias que deben instalarse para finalizar la configuración:

médico aleteo

Vea el informe y busque otro software que pueda ser necesario.

Configuración de iOS: Instale Xcode

Se requiere una Mac con Xcode 10.0 o posterior.

1. Abra App Store e instale Xcode.

2. Configure las herramientas de línea de comandos de Xcode ejecutando sudo xcode-select --switch / Aplicaciones/Xcode.app/Contents/Developer desde la ventana Terminal.

3. Confirme que el acuerdo de licencia de Xcode esté firmado ejecutando sudo xcodebuild -licencia de la terminal.

Configuración de Android: instalar Android Studio

Los detalles completos de la instalación están disponibles en <https://flutter.dev/docs/get-started/install/macos#android-setup>. Flutter requiere una instalación completa de Android Studio para las dependencias de la plataforma Android. Tenga en cuenta que las aplicaciones de Flutter se pueden escribir en diferentes editores, como Visual Code o IntelliJ IDEA.

1. Descargue e instale Android Studio desde <https://developer.android.com/studio/>.
2. Inicie Android Studio y siga el asistente de configuración, que instala todos los SDK de Android requerido por Flutter. Si le pide que importe la configuración anterior, puede hacer clic en Sí para usar la configuración actual o en No para comenzar con la configuración predeterminada.

Configurar el emulador de Android Puede ver

detalles sobre cómo crear y administrar dispositivos virtuales en <https://developer.android.com/studio/run/managing-avds>.

1. Habilite la aceleración de VM en su máquina. Las instrucciones están disponibles en <https://developer.android.com/studio/run/emulator-acceleration>.
2. Si es la primera vez que instala Android Studio, para acceder al Administrador de AVD, debe crear un nuevo proyecto. Inicie Android Studio, haga clic en Iniciar un nuevo proyecto de Android Studio, asignele cualquier nombre y acepte los valores predeterminados. Una vez creado el proyecto, continúa con estos pasos.

Actualmente, Android Studio requiere un proyecto de Android abierto antes de poder acceder al submenú de Android.

3. Desde Android Studio, seleccione Herramientas → Android → AVD Manager → Crear dispositivo virtual.
4. Seleccione un dispositivo de su elección y haga clic en Siguiente.
5. Seleccione una imagen x86 o x86_64 para emular la versión de Android.
6. Si está disponible, asegúrese de seleccionar Hardware: GLES 2.0 para habilitar la aceleración de hardware. Esta aceleración de hardware hará que el emulador de Android funcione más rápido.
7. Haga clic en el botón Finalizar.
8. Para verificar que la imagen del emulador se haya instalado correctamente, seleccione Herramientas → AVD Manager y luego haga clic en el botón Ejecutar (ícono de reproducción).

Instalación en Windows

Antes de comenzar la instalación, debe asegurarse de que su Windows admita los requisitos mínimos de hardware y software.

Requisitos del sistema → Windows 7

SP1 o posterior (64 bits)

400 MB de espacio en disco (no incluye espacio en disco para un entorno de desarrollo integrado
ment y otras herramientas)

Las siguientes herramientas de línea de comandos:

PowerShell 5.0 o posterior

Git para Windows (Git desde el símbolo del sistema de Windows)

Obtenga el SDK de Flutter

Los últimos detalles de instalación están disponibles en <https://flutter.dev/docs/get-started/install/windows/>.

1. Descargue el siguiente archivo de instalación para obtener la versión más reciente, v1.7.8 en el momento de escribir este artículo (tu versión puede ser superior), del SDK de Flutter:

https://storage.googleapis.com/flutter_infra/releases/stable/windows/flutter_windows_v1.7.8+hotfix.4-stable.zip.

2. Extraiga el archivo en la ubicación deseada.

Reemplace WindowsUserName con su propio nombre de usuario de Windows. No instales Flutter en directorios que requieran privilegios elevados como C:\Program Files\). Usé la siguiente ubicación de carpeta:

C:\Users\WindowsUserName\flutter

3. Busque en C:\Users\WindowsUserName\flutter y haga doble clic en flutter_con archivo sole.bat .

4. Actualice la ruta de forma permanente (la siguiente es para Windows 10):

a. Abra el Panel de control y navegue hasta Aplicación de escritorio Cuentas de usuario
Cuentas de usuario Cambiar mis variables de entorno.

b. En Variables de usuario para WindowsUserName, seleccione la variable Ruta y haga clic en el botón Editar.
Si falta la variable Ruta , salte al siguiente subpaso, c. i. En la variable de entorno Editar, haga clic en el botón Nuevo. ii. Escriba la ruta C:
\Users\WindowsUserName\flutter\bin.
iii. Haga clic en el botón Aceptar y luego cierre la pantalla Variables de entorno.

C. En Variables de usuario para WindowsUserName, si falta la variable Path , haga clic en el botón New y escriba Path para el nombre de la variable y C:\Users\ WindowsUserName\flutter\bin para el valor de la variable.

5. Reinicie Windows para aplicar los cambios.

Comprobar dependencias

Ejecute el siguiente comando desde el símbolo del sistema de Windows para verificar las dependencias que deben instalarse para finalizar la configuración:

médico aleteo

Vea el informe y compruebe si es necesario instalar otro software.

Instalar estudio de Android

Los detalles completos de instalación están disponibles en <https://flutter.dev/docs/get-started/install/windows#android-setup>. Flutter requiere una instalación completa de Android Studio para las dependencias de la plataforma Android. Tenga en cuenta que las aplicaciones de Flutter se pueden escribir en diferentes editores como Visual Code o IntelliJ IDEA.

1. Descargue e instale Android Studio desde <https://developer.android.com/studio/>.
2. Inicie Android Studio y siga el asistente de configuración, que instala todos los SDK de Android requerido por Flutter. Si le pide que importe la configuración anterior, puede hacer clic en Sí para usar la configuración actual o en No para comenzar con la configuración predeterminada.

Configurar el emulador de Android Puede

ver detalles sobre cómo crear y administrar dispositivos virtuales en <https://developer.android.com/studio/run/managing-avds>.

1. Habilite la aceleración de VM en su máquina. Las instrucciones están disponibles en <https://developer.android.com/studio/run/emulator-acceleration>.
2. Si es la primera vez que instala Android Studio, para acceder al Administrador de AVD, debe crear un nuevo proyecto. Inicie Android Studio y haga clic en Iniciar un nuevo proyecto de Android Studio y asigne cualquier nombre y acepte los valores predeterminados. Una vez creado el proyecto, continúa con estos pasos.

Actualmente, Android Studio requiere un proyecto de Android abierto antes de poder acceder al submenú de Android.
3. Desde Android Studio, seleccione Herramientas → Android → AVD Manager → Crear dispositivo virtual.
4. Seleccione un dispositivo de su elección y haga clic en Siguiente.
5. Seleccione una imagen x86 o x86_64 para emular la versión de Android.
6. Si está disponible, asegúrese de seleccionar Hardware: GLES 2.0 para habilitar la aceleración de hardware. Esta aceleración de hardware hará que el emulador de Android funcione más rápido.
7. Haga clic en el botón Finalizar.
8. Para verificar que la imagen del emulador se haya instalado correctamente, seleccione Herramientas → AVD Manager y luego haga clic en el botón Ejecutar (ícono de reproducción).

Instalación en Linux

Antes de comenzar la instalación, debe asegurarse de que su Linux admite los requisitos mínimos de hardware y software.

Requisitos del sistema → Linux

(64 bits)

600 MB de espacio en disco (no incluye espacio en disco para un entorno de desarrollo integrado
ment y otras herramientas)

Las siguientes herramientas de línea de comandos:

golpe

rizo

git 2.x

```
mkdir
```

```
rm
```

```
descomprimir
```

```
que
```

```
xz-utils
```

La biblioteca compartida libGLU.so.1 proporcionada por los paquetes mesa (p. ej., libglu1-mesa en Ubuntu/Debian)

Obtenga el SDK de Flutter

Los últimos detalles de instalación están disponibles en <https://flutter.dev/docs/get-started/install/linux/>.

1. Descargue el siguiente archivo de instalación para obtener la última versión, que es v1.7.8 en este momento de escritura (su versión puede ser superior), del SDK de Flutter:

```
https://storage.googleapis.com/flutter_infra/releases/stable/linux/flutter_linux_v1.7.8+hotfix.4-stable.tar.xz.
```

2. Extraiga el archivo en la ubicación deseada usando la ventana Terminal:

```
cd ~/desarrollo tar xf
```

```
~/Downloads/flutter_linux_v1.7.8+hotfix.4-stable.tar.xz 3. Agregue la herramienta
```

Flutter a su ruta (pwd significa directorio de trabajo actual; en su caso, será ser la carpeta de desarrollo).

```
export RUTA="$RUTA:`pwd`/flutter/bin"
```

4. Actualice la ruta de forma permanente.

- a. Obtenga la ruta que usó en el paso 3. Por ejemplo, reemplace PathToDev con su ruta a la carpeta de desarrollo.

```
/PathToDev/desarrollo/flutter/bin
```

- b. Abra o cree \$HOME/.bash_profile. Su ruta de archivo o nombre de archivo puede ser diferente en tu ordenador.

C. Edite .bash_profile (abrirá un editor de línea de comandos). d.

Agregue la siguiente línea y asegúrese de reemplazar PathToDev con su ruta:

```
export PATH="$PATH :/PathToDev/desarrollo/flutter/bin" .
```

mi. Ejecute source \$HOME/.bash_profile para actualizar la ventana actual.

F. En la ventana Terminal, escriba echo \$PATH para verificar que se agregó la ruta. Luego escriba flutter para asegurarse de que la ruta funcione. Si no se reconoce el comando, algo salió mal con la RUTA. Compruebe que tiene la ruta correcta.

Comprobar dependencias

Ejecute el siguiente comando desde la ventana de Terminal para verificar las dependencias que deben instalarse para finalizar la configuración:

médico aleteo

Vea el informe y busque otro software que pueda ser necesario.

Instalar estudio de Android

Los detalles completos de instalación están disponibles en <https://flutter.dev/docs/get-started/install/linux#android-setup>. Flutter requiere una instalación completa de Android Studio para las dependencias de la plataforma Android. Tenga en cuenta que las aplicaciones de Flutter se pueden escribir en diferentes editores como Visual Code o IntelliJ IDEA.

1. Descargue e instale Android Studio desde <https://developer.android.com/studio/>.
2. Inicie Android Studio y siga el asistente de configuración, que instala todos los SDK de Android.
requerido por Flutter. Si le pide que importe la configuración anterior, puede hacer clic en Sí para usar la configuración actual o en No para comenzar con la configuración predeterminada.

Configurar el emulador de Android Puede

ver detalles sobre cómo crear y administrar dispositivos virtuales en <https://developer.android.com/studio/run/managing-avds>.

1. Habilite la aceleración de VM en su máquina. Las instrucciones están disponibles en <https://developer.android.com/studio/run/emulator-acceleration>.
2. Si es la primera vez que instala Android Studio, para acceder al Administrador de AVD, debe crear un nuevo proyecto. Inicie Android Studio, haga clic en Iniciar un nuevo proyecto de Android Studio, asignele cualquier nombre y acepte los valores predeterminados. Una vez creado el proyecto, continúa con estos pasos.

Actualmente, Android Studio requiere un proyecto de Android abierto antes de poder acceder al submenú de Android.

3. Desde Android Studio, seleccione Herramientas → Android → AVD Manager → Crear dispositivo virtual.
4. Seleccione un dispositivo de su elección y haga clic en Siguiente.
5. Seleccione una imagen x86 o x86_64 para emular la versión de Android.
6. Si está disponible, asegúrese de seleccionar Hardware: GLES 2.0 para habilitar la aceleración de hardware.
Esta aceleración de hardware hará que el emulador de Android funcione más rápido.
7. Haga clic en el botón Finalizar.
8. Para verificar que la imagen del emulador se haya instalado correctamente, seleccione Herramientas → AVD Manager y luego haga clic en el botón Ejecutar (ícono de reproducción).

CONFIGURAR EL EDITOR DE ESTUDIO DE ANDROID

El editor que utilizará es Android Studio. Android Studio es el entorno de desarrollo integrado oficial para el sistema operativo Android de Google y está diseñado específicamente para el desarrollo de Android. También es un excelente entorno de desarrollo para desarrollar aplicaciones con Flutter. Antes de comenzar a crear una aplicación, el editor necesita que se instalen los complementos Flutter y Dart para que sea más fácil escribir código.

(Otros editores que admiten estos complementos son IntelliJ o Visual Studio Code). Los complementos del editor brindan finalización de código, resaltado de sintaxis, compatibilidad con ejecución y depuración, y más. Usar un editor de texto sin formato para escribir su código sin complementos también funcionaría, pero no se recomienda el uso de funciones de complementos.

Las instrucciones para configurar diferentes editores de código están disponibles en <https://flutter.dev/docs/get-started/editor/>.

Para admitir el desarrollo de Flutter, instale los siguientes complementos:

Complemento Flutter para flujos de trabajo de desarrolladores, como ejecución, depuración y recarga en caliente

Complemento Dart para análisis de código, como validación instantánea de código y finalización de código

Siga estos pasos para instalar los complementos Flutter y Dart:

1. Inicie Android Studio.
2. Haga clic en Preferencias Complementos (en macOS) o Archivo Configuración Complementos (en Windows y Linux).
3. Haga clic en Examinar repositorios, seleccione el complemento Flutter y haga clic en el botón Instalar.
4. Haga clic en Sí cuando se le solicite instalar el complemento Dart.
5. Haga clic en Reiniciar cuando se le solicite.

RESUMEN

En este capítulo, aprendiste cómo funciona la arquitectura del marco Flutter detrás de escena. Aprendiste que Flutter es un excelente marco de interfaz de usuario portátil para crear aplicaciones móviles para iOS y Android. Flutter también planea respaldar el desarrollo para dispositivos integrados, de escritorio y web. Aprendió que las aplicaciones de Flutter se construyen a partir de una única base de código que usa widgets para crear la interfaz de usuario y que usted desarrolla con el lenguaje Dart. Aprendiste que Flutter usa el motor de renderizado Skia 2D que funciona con diferentes tipos de hardware y software.

Aprendió que el lenguaje Dart es AOT compilado en código nativo, lo que resulta en un rendimiento rápido para sus aplicaciones. Aprendiste que Dart está compilado con JIT, lo que agiliza la visualización de cambios en el código con la recarga en caliente con estado de Flutter.

Aprendió que los widgets son los componentes básicos para componer la interfaz de usuario, y cada widget es una declaración de tabla immu de la interfaz de usuario. Los widgets son la configuración para crear elementos. Los elementos son los widgets hechos concretos, es decir, montados y pintados en la pantalla. Aprendió que RenderObject implementa los protocolos básicos de diseño y pintura.

Aprendió sobre los eventos del ciclo de vida de los widgets sin estado y con estado. Aprendiste que el widget sin estado es declarado por una sola clase que se extiende (hereda) de la clase StatelessWidget .

Aprendió que el widget con estado se declara con dos clases, la clase StatefulWidget y la clase State .

Aprendiste que Flutter es declarativo y que la IU se reconstruye sola cuando cambia el estado. Aprendió que los widgets son los componentes básicos de una aplicación Flutter y que los widgets son las configuraciones para la interfaz de usuario.

Aprendió que anidar (componer) los widgets da como resultado la creación del árbol de widgets. El marco de trabajo de Flutter usa el widget como la configuración para construir cada elemento, lo que da como resultado la creación del árbol de elementos. El elemento es el widget que se monta (representa) en la pantalla. El proceso anterior da como resultado la creación del árbol de renderizado, que es un sistema de diseño y pintura de bajo nivel. Usará widgets y no necesitará interactuar directamente con el árbol de representación. Aprendió que los widgets sin estado tienen la configuración para crear elementos sin estado. Aprendió que los widgets con estado tienen la configuración para crear elementos con estado y las solicitudes de elementos con estado del widget para crear un objeto de estado.

Aprendió a instalar Flutter SDK, Xcode para compilar para iOS y Android Studio para compilar para dispositivos Android. Cuando Android Studio está instalado en una Mac, maneja la compilación para dispositivos iOS (a través de Xcode) y Android. Aprendió a instalar los complementos Flutter y Dart para ayudar a los flujos de trabajo de los desarrolladores, como la finalización de código, el resultado de sintaxis, la ejecución, la depuración, la recarga en caliente y el análisis de código.

En el próximo capítulo, aprenderá cómo crear su primera aplicación y usar la recarga en caliente para ver los cambios de inmediato. También aprenderá cómo usar temas para diseñar la aplicación, cuándo usar widgets sin estado o con estado, y cómo usar paquetes externos para agregar funciones como GPS y gráficos rápidamente.

LO QUE APRENDISTE EN ESTE CAPÍTULO

TEMA	CONCEPTOS CLAVE
Aleteo	Flutter es un marco de interfaz de usuario portátil para crear aplicaciones móviles modernas, nativas y reactivas para iOS y Android desde una única base de código. Flutter también ha ampliado el desarrollo para escritorio, web y dispositivos integrados.
Esquiar	Flutter utiliza el motor de renderizado Skia 2D que funciona con diferentes plataformas de hardware y software.
Dardo	Dart es el lenguaje utilizado para desarrollar aplicaciones de Flutter. Dart se compila con anticipación (AOT) en código nativo para un rendimiento rápido. Dart se compila justo a tiempo (JIT) para mostrar rápidamente los cambios de código a través de la recarga en caliente con estado de Flutter.
Interfaz de usuario declarativa (UI)	Flutter es declarativo y crea la interfaz de usuario para reflejar el estado de la aplicación. Cuando el estado cambia, la interfaz de usuario se vuelve a dibujar. Flutter usa Dart para crear la interfaz de usuario.
Widget	Los widgets son los componentes básicos de una aplicación Flutter, y cada widget es una declaración inmutable de la interfaz de usuario (UI). Los widgets son la configuración para crear un elemento.
Elemento	Los elementos son widgets que se montan (representan) en la pantalla. Los elementos son creados por la configuración del widget.
RenderObjeto	El RenderObject es un objeto en el árbol de renderizado que calcula e implementa los protocolos básicos de diseño y pintura.
Eventos del ciclo de vida de los widgets	Cada widget sin estado o con estado tiene un método de compilación con un BuildContext que maneja la ubicación del widget en el árbol de widgets. Los objetos BuildContext son objetos Element , lo que significa una instanciación del Widget en una ubicación en el árbol.
Widget sin estado	El StatelessWidget está construido en base a su propia configuración y no cambia dinámicamente. El widget sin estado se declara con una clase.
Widget con estado	StatefulWidget se crea en función de su propia configuración, pero puede cambiar dinámicamente . El widget con estado se declara con dos clases, la clase StatefulWidget y la clase State .
Árbol de widgets	El árbol de widgets se crea al componer (anidar) widgets; esto se conoce como composición. Se crean tres árboles: el árbol de widgets, el árbol de elementos y el árbol de representación.
árbol de elementos	El árbol de elementos representa cada elemento montado (renderizado) en la pantalla.

TEMA	CONCEPTOS CLAVE
árbol de renderizado	El árbol de renderizado es un sistema de diseño y pintura de bajo nivel que se hereda del RenderObject. El RenderObject implementa los protocolos básicos de diseño y pintura. Usará widgets y no necesitará interactuar directamente con el árbol de representación.
Flutter SDK y dardo	El kit de desarrollo de software móvil se ha expandido a dispositivos integrados, web y de escritorio.
Herramientas de desarrollo Xcode y Android Studio para crear aplicaciones móviles iOS y Android.	
Complemento de aleteo	Este complemento ayuda con los flujos de trabajo de los desarrolladores, como la ejecución, la depuración y la recarga en caliente.
Complemento de dardo	Este complemento ayuda con el análisis de código, como la validación del código de instancia y la finalización del código.

2

Creación de una aplicación Hello World

LO QUE APRENDERÁS EN ESTE CAPÍTULO

Cómo crear una nueva aplicación móvil Flutter

Cómo actualizar la aplicación en ejecución con recarga en caliente

Cómo diseñar la aplicación con temas

Cuándo usar un widget sin estado o con estado

Cómo agregar paquetes externos

Una excelente manera de aprender un nuevo lenguaje de desarrollo es escribir una aplicación básica. En programación, Hello World es el programa más básico para escribir. Simplemente muestra las palabras Hello World en la pantalla. En este capítulo, aprenderá los pasos principales para desarrollar este programa básico como una aplicación de Flutter. No te preocupes por entender el código todavía; Lo explicaré paso a paso más adelante en este libro.

Escribir este ejemplo mínimo lo ayuda a aprender la estructura básica de una aplicación Flutter, cómo ejecutar la aplicación en el simulador de iOS y el emulador de Android, y cómo realizar cambios en el código.

CONFIGURACIÓN DEL PROYECTO

La configuración inicial del proyecto para cada aplicación es la misma. Estoy usando Android Studio para crear las aplicaciones de muestra de este libro, pero puede elegir un editor diferente, como IntelliJ o Visual Studio Code. Una descripción general del proceso en Android Studio es la siguiente: crea un nuevo proyecto Flutter, selecciona la aplicación Flutter como tipo de proyecto (plantilla) e ingresa el nombre del proyecto. Luego, el kit de desarrollo de software (SDK) de Flutter crea el proyecto por usted, incluida la creación de un directorio de proyecto con el mismo nombre que el nombre del proyecto. Dentro del directorio del proyecto, la carpeta lib contiene el archivo main.dart con el código fuente (en otras palabras, project_name/lib/main.dart). También tendrá una carpeta de Android para la aplicación de Android, la carpeta de ios para la aplicación de iOS y una carpeta de prueba para las pruebas unitarias.

En este capítulo, la carpeta lib y el archivo main.dart son su enfoque principal. En el Capítulo 4, "Creación de una plantilla de proyecto de inicio", aprenderá cómo crear un proyecto de inicio de Flutter y cómo estructurar el código en archivos separados.

De manera predeterminada, la aplicación Flutter usa widgets de Componentes de materiales basados en Material Design. Material Design es un sistema de pautas de mejores prácticas para el diseño de interfaces de usuario. Los componentes de un proyecto de Flutter incluyen widgets visuales, de comportamiento y de movimiento. Los proyectos de Flutter también incluyen pruebas unitarias para widgets, que son archivos que contienen código individual para probar si la lógica funciona según lo diseñado.

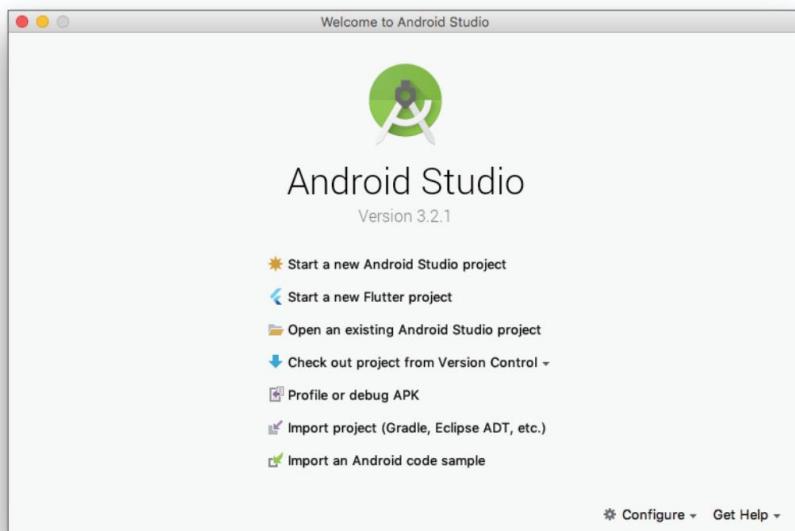
Las aplicaciones de Flutter se construyen con el lenguaje de programación Dart, y aprenderá los conceptos básicos de Dart en el Capítulo 3, "Aprender los conceptos básicos de Dart".

PRUÉBALO Creación de una nueva aplicación

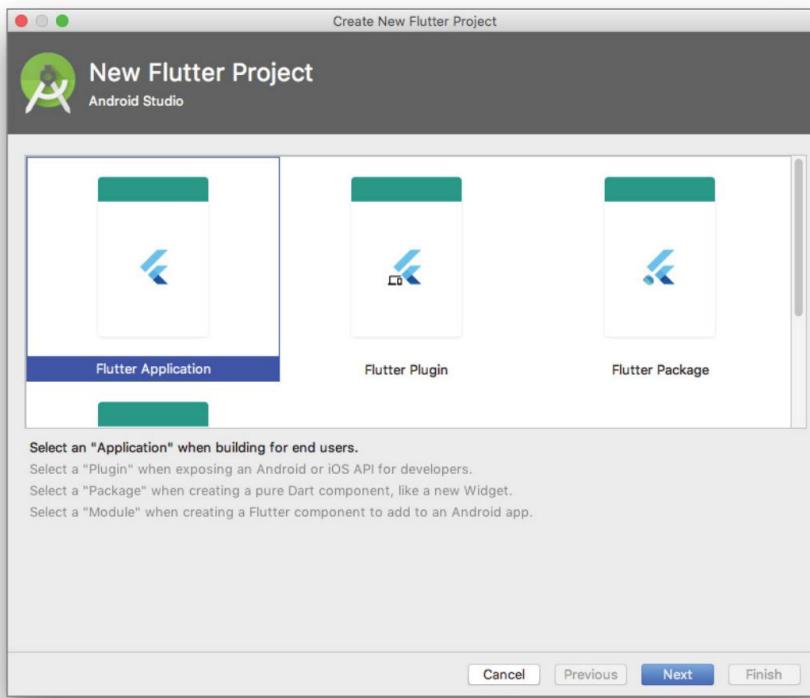
En este ejercicio, creará una nueva aplicación Flutter llamada ch2_my_counter. El nombre de la aplicación es el mismo que el nombre del proyecto. Esta aplicación utiliza la plantilla de proyecto predeterminada mínima de Flutter e incluye un botón de acción flotante (el botón +) que aparece en la parte inferior derecha de la pantalla del dispositivo. Cada vez que se toca este botón, un contador aumenta en uno.

La plantilla Flutter actual crea la aplicación básica que tiene un contador y un botón + (el botón de acción flotante). En futuros lanzamientos de Flutter, puede haber otras opciones de plantilla disponibles. ¿Por qué el equipo de Flutter decidió usar un contador como plantilla? Es inteligente porque muestra cómo tomar datos, manipularlos y mantener el estado (el valor del contador) con recarga en caliente.

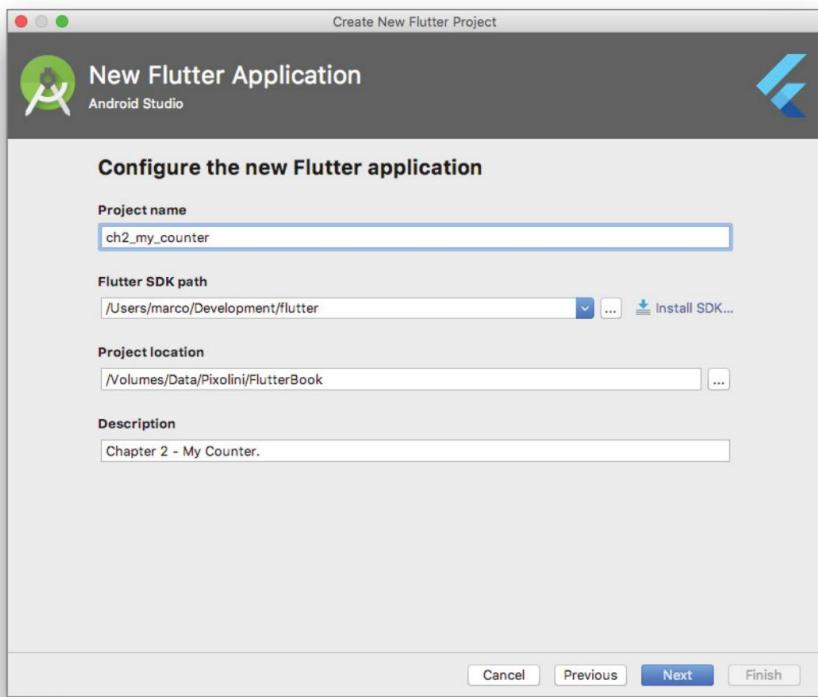
1. Inicie Android Studio.
2. Si tiene un proyecto abierto, haga clic en la barra de menú y seleccione Archivo → Nuevo → Nuevo proyecto Flutter. Si no proyecto está abierto, haga clic en Iniciar un nuevo proyecto de Flutter.



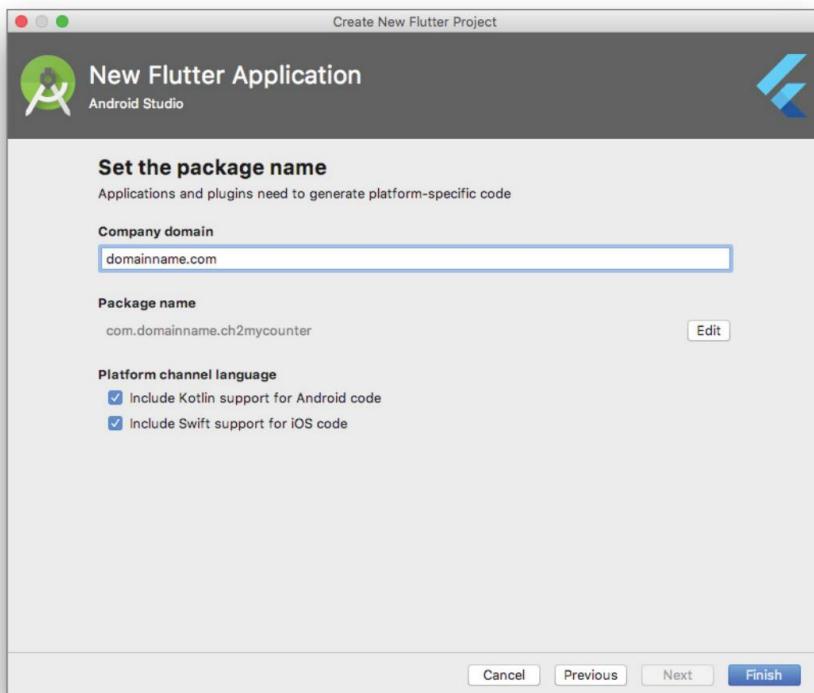
3. Seleccione la aplicación Flutter.



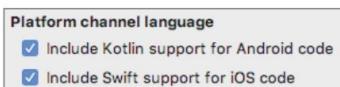
4. La convención de nomenclatura habitual es separar las palabras con un guión bajo, así que ingrese el proyecto nombre ch2_my_counter y haga clic en Siguiente. Tenga en cuenta que la ruta del SDK de Flutter es la carpeta de instalación que eligió en el Capítulo 1. Opcionalmente, puede cambiar la ubicación y la descripción del proyecto.



5. Introduzca el nombre de su empresa en el campo Dominio de la empresa y déle formato como nombrededominio.com. Usar cualquier nombre único si no tiene un nombre de empresa.



6. En la misma pantalla que el paso 5 para el idioma del canal de la plataforma, seleccione ambas opciones para Kotlin y rápido.



Estas opciones garantizarán que esté utilizando los últimos lenguajes de programación para Android (Kotlin) e iOS (Swift). El uso del canal de la plataforma, una forma de comunicación entre el código Dart de la aplicación y el código específico de la plataforma, le permite usar interfaces de programación de aplicaciones (API) específicas de la plataforma Android e iOS escribiendo código en cada idioma nativo, como escribir código nativo en Kotlin (Android) y Swift (iOS) para manejar la reproducción de audio mientras la aplicación está en modo de fondo. Echará un vistazo más de cerca a los canales de la plataforma, Kotlin y Swift en el Capítulo 12, "Escribir código nativo de la plataforma".

7. Haga clic en el botón Finalizar.

Cómo funciona

Se crea un proyecto de Flutter con el nombre de carpeta ch2_my_counter, igual que el nombre del proyecto. Utiliza la plantilla de proyecto estándar de Flutter, que muestra cómo actualizar un contador tocando el botón +.

De forma predeterminada, se utilizan los componentes de Material Design de Android. En un proyecto de Flutter, estos incluyen widgets visuales, de comportamiento y de movimiento. Para cada proyecto que cree, hay un directorio llamado lib, y el archivo main.dart se ejecutará primero cuando se ejecute la aplicación. El archivo main.dart contiene código Dart con la función main() que inicia la aplicación. El objetivo de crear esta aplicación es familiarizarse con la forma en que se crea y estructura una aplicación de Flutter.

El Capítulo 3 cubre los conceptos básicos de Dart, y el Capítulo 4 cubre el archivo main.dart con más detalle, particularmente cómo se estructura y se separa la lógica del código.

USO DE RECARGA EN CALIENTE

La recarga en caliente de Flutter lo ayuda a ver los cambios en el código y la interfaz de usuario de inmediato mientras conserva el estado de una aplicación que ejecuta la máquina virtual Dart. En otras palabras, cada vez que realiza cambios en el código, no necesita volver a cargar la aplicación, porque la página actual muestra los cambios de inmediato. Esta es una característica increíble que ahorra tiempo a cualquier desarrollador.

En Flutter, la clase State almacena datos mutables (modificables). Por ejemplo, la aplicación comienza con el valor del contador en cero, pero cada vez que toca el botón +, el valor del contador aumenta en uno.

Cuando el botón + se toca tres veces, el valor del contador muestra un valor de tres. El valor del contador es mutable, por lo que puede cambiar con el tiempo. Al usar la recarga en caliente, puede realizar cambios en la lógica del código y el estado (valor) del contador de la aplicación no se restablece a cero, sino que conserva el valor actual de tres.

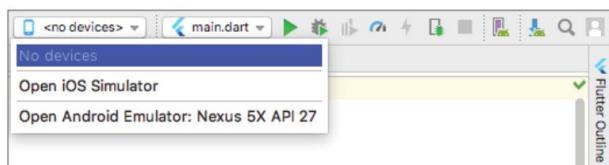
PRUÉBELO Ejecutando la aplicación

Para ver cómo funciona la recarga en caliente, iniciará el emulador/simulador, realizará cambios en el título de la página, los guardará y verá que los cambios se realizan de inmediato.

Desde el Capítulo 1, el simulador de iOS se creó automáticamente cuando instaló Xcode y creó manualmente el emulador de Android. El simulador de iOS solo está disponible si se ejecuta Android Studio en una computadora Mac porque requiere la instalación de Xcode de Apple. Se supone que tiene instalados tanto el simulador de iOS como el emulador de Android. Si no, usa el emulador de Android.

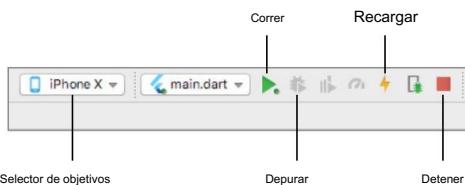
1. Desde Android Studio, haz clic en el botón de selección de dispositivos Flutter a la derecha de la barra de herramientas.

Una lista desplegable muestra el simulador de iOS y el emulador de Android disponibles.

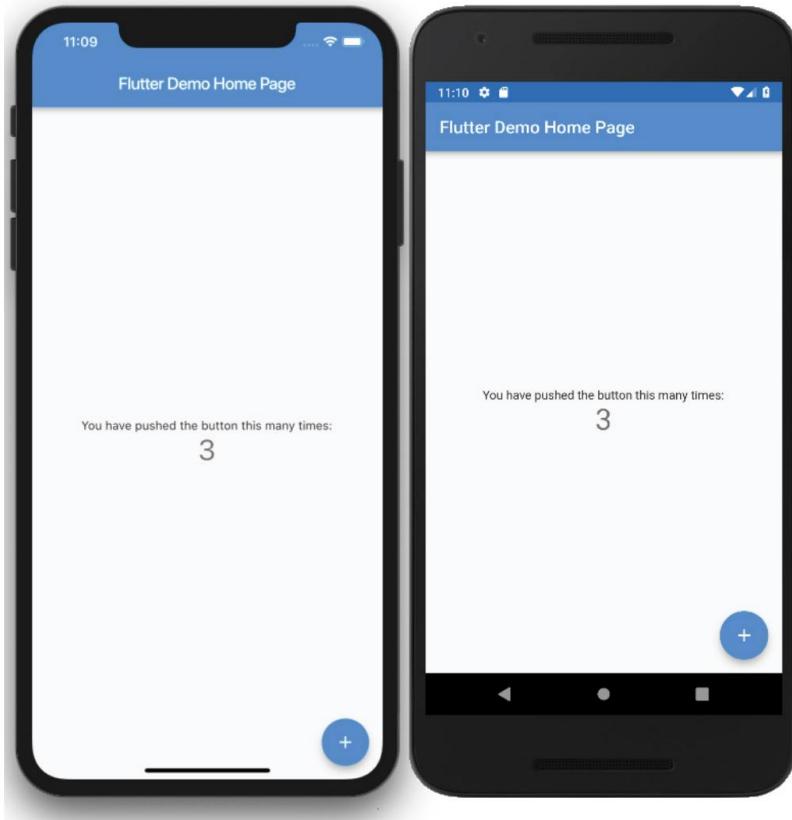


2. Seleccione el simulador de iOS o el emulador de Android.

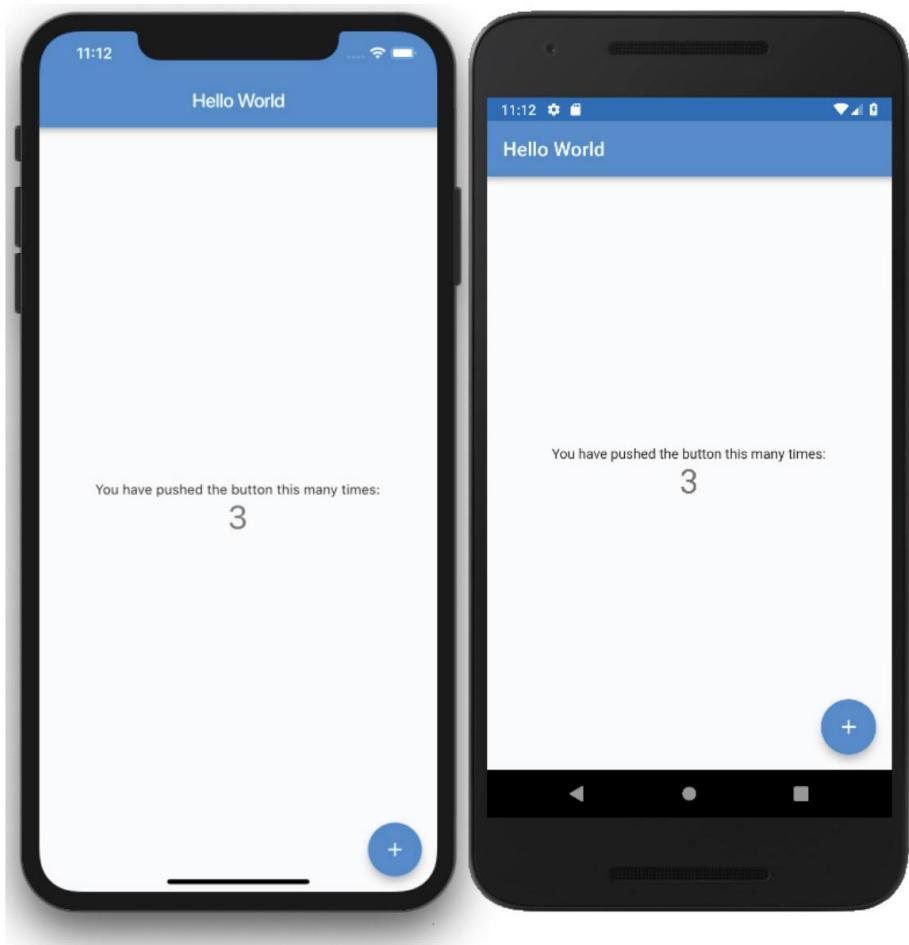
3. Haga clic en el ícono Ejecutar en la barra de herramientas.



4. Debería ver la aplicación ch2_my_counter en el simulador. Siga el paso 2 para ejecutar la aplicación tanto en el simulador de iOS (si está disponible) como en el emulador de Android. Al ejecutar la aplicación tanto en iOS como en Android, es evidente que se ve igual pero hereda las características de cada sistema operativo móvil. Tenga en cuenta que en el simulador de iOS el título de la aplicación está centrado, pero en el emulador de Android, el título está a la izquierda.
5. Haga clic en el botón de acción flotante + en la parte inferior derecha y verá que el contador aumenta cada vez. vez que haga clic en él.



6. En el archivo main.dart , cambie MyHomePage (título: 'Flutter Demo Home Page') a MyHomePage(título: 'Hello World') y guárdelo presionando (en Windows). Inmediatamente verás que el título barra de aplicaciones cambia y el estado del contador sigue siendo el mismo, sin volver a cero. Este cambio instantáneo se denomina recarga en caliente y lo usará con frecuencia para mejorar su productividad.



Cómo funciona

La recarga en caliente es una función increíble que permite ahorrar tiempo al ver los resultados de los cambios en el código fuente de forma inmediata mientras se mantiene el estado actual. Mientras se ejecuta la máquina virtual de Dart, la recarga activa inyecta el código fuente actualizado y el marco Flutter reconstruye el árbol de widgets. (El árbol de widgets se tratará en detalle en el Capítulo 5, “Comprensión del árbol de widgets”).

USO DE TEMAS PARA DISEÑAR TU APLICACIÓN

Los widgets de temas son una excelente manera de diseñar y definir colores globales y estilos de fuente para su aplicación. Hay dos formas de usar los widgets de temas: para diseñar la apariencia globalmente o para diseñar solo una parte de la aplicación. Por ejemplo, puede usar temas para diseñar el brillo del color (texto claro sobre un fondo oscuro o viceversa); los colores primarios y de acento; el color del lienzo; y el color de las barras de aplicaciones, tarjetas, divisores, opciones seleccionadas y no seleccionadas, botones, sugerencias, errores, texto, íconos, etc. La belleza de Flutter es que la mayoría de los elementos son widgets y casi todo es personalizable. De hecho, personalizar la clase ThemeData le permite cambiar el color y la tipografía de los widgets. (Obtendrá más información sobre los widgets en detalle en el Capítulo 5, "Comprensión del árbol de widgets" y el Capítulo 6, "Uso de widgets comunes").

Uso de un tema de aplicación global

Tomemos la nueva aplicación ch2_my_counter y modifiquemos el color primario. El color actual es azul, así que cambiémoslo a verde claro. Agregue una nueva línea debajo de primarySwatch y agregue código para cambiar el color de fondo (canvasColor) a lightGreen.

```
PrimarySwatch: Colors.blue, //  
Cámbielo a  
PrimarySwatch: Colors.lightGreen,  
canvasColor: Colors.lightGreen.shade100,
```

Guardar pulsando (). Se invoca recarga en caliente, por lo que la barra de la aplicación y el lienzo ahora tienen un tono verde claro.

Para mostrar un poco de genialidad de Flutter, agregue una propiedad de plataforma de TargetPlatform.iOS después de la propiedad canvasColor y ejecute la aplicación desde el emulador de Android. De repente, las características de iOS se ejecutan en Android. El título de la barra de la aplicación no se alinea a la izquierda sino que se cambia al centro, que es el estilo personalizado de iOS (Figura 2.1).

```
PrimarySwatch: Colors.blue, //  
Cámbielo a  
PrimarySwatch: Colors.lightGreen,  
canvasColor: Colors.lightGreen.shade100, plataforma:  
TargetPlatform.iOS
```

Esto se puede hacer a la inversa usando TargetPlatform. Específicamente, para mostrar características de Android en iOS, cambie la propiedad de la plataforma a TargetPlatform.android y ejecute la aplicación desde el simulador de iOS. El título de la barra de la aplicación no está alineado en el centro, sino que ha cambiado para estar alineado a la izquierda, que es el estilo habitual de Android (Figura 2.2). Una vez que se implemente la navegación con múltiples páginas, esto será aún más evidente. En iOS, al navegar a una nueva página, generalmente desliza la siguiente página desde el lado derecho de la pantalla hacia la izquierda. En Android, al navegar a una página nueva, generalmente desliza la siguiente página de abajo hacia arriba. TargetPlatform tiene tres opciones: Android, Fuchsia (el sistema operativo en desarrollo de Google) e iOS.

```
PrimarySwatch: Colors.blue, //  
Cámbielo a  
PrimarySwatch: Colors.lightGreen,  
canvasColor: Colors.lightGreen.shade100, plataforma:  
TargetPlatform.android
```



FIGURA 2.1: características de iOS ejecutándose en Android

Este es otro ejemplo del uso de un tema de aplicación global: si tiene un banner rojo de depuración en la parte superior derecha del emulador, puede desactivarlo con el siguiente código. Google tenía la intención de informar a los desarrolladores que el rendimiento de la aplicación no está en modo de lanzamiento. Flutter crea una versión de depuración de la aplicación y el rendimiento es más lento. Usando el modo de lanzamiento (solo dispositivo), crea una aplicación con optimización de velocidad. Agregue la propiedad `debugShowCheckedModeBanner` y establezca el valor en `false`, y se eliminará el banner rojo de depuración. Desactivar el banner rojo de depuración es solo por motivos estéticos; todavía está ejecutando una versión de depuración de la aplicación.

```
devolver nueva MaterialApp(  
    debugShowCheckedModeBanner: falso, título:  
    'Mi contador', tema: new  
    ThemeData(  
        ...
```



FIGURA 2.2: características de Android ejecutándose en iOS

Uso de un tema para parte de una aplicación Para anular

el tema de toda la aplicación, puede envolver widgets en un widget de tema . Este método anulará por completo la instancia de ThemeData de la aplicación sin heredar ningún estilo.

En la sección anterior, cambió primarySwatch y canvasColor a lightGreen, lo que afectó a todos los widgets de la aplicación. ¿Qué sucede si desea que solo un widget en una página tenga un esquema de color diferente y que el resto de los widgets usen el tema de aplicación global predeterminado? Anula el tema predeterminado con un widget de tema que usa la propiedad de datos para personalizar ThemeData (como el color de la tarjeta, el color primario y el color de lienzo) y el widget secundario usa la propiedad de datos para personalizar los colores.

```
cuerpo: Centro(  
    niño: tema(  
        // Tema único con ThemeData - Sobrescribir
```

```
datos:  
    ThemeData( cardColor: Colors.deepOrange, ),  
  
child: Card( child:  
    Text('Unique ThemeData'), ), ), ),
```

Recomiendo ampliar el tema principal de la aplicación, cambiar solo las propiedades necesarias y heredar el resto. Use el método `copyWith` para crear una copia del tema principal de la aplicación y reemplace solo las propiedades que necesita cambiar. Desglosándolo, `Theme.of(context).copyWith()` amplía el tema principal y puede anular las propiedades necesarias dentro de `copyWith(cardColor: Colors. deepOrange)`.

```
cuerpo: Centro(  
niño: tema(  
    // copyWith Theme - Heredar datos (Extendidos):  
    Theme.of(context).copyWith(cardColor: Colors.deepOrange), child: Card( child: Text('copyWith  
    Theme'), ), ), ),
```

El siguiente código de muestra muestra cómo cambiar el color predeterminado de la tarjeta a naranja intenso con el tema sobrescrito (`ThemeData()`) y extendido (`Theme.of().copyWith()`) para producir el mismo resultado. Los dos widgets de tema están envueltos dentro de un widget de columna para alinearlos verticalmente. En este punto, no se preocupe por el widget Columna , ya que se trata en el Capítulo 6.

```
body:  
Column( children:  
  
<Widget>[ Theme( // Tema único con ThemeData - Sobrescribir datos:  
    ThemeData( cardColor:  
        Colors.deepOrange, ), child: Card( child:  
  
        Text('Unique  
        ThemeData'), ), ), Theme( // copyWith Theme -
```

Heredar
datos (Extendidos): `Theme.of(context).copyWith(cardColor:
Colors.deepOrange), child: Card(child: Text('copyWith Theme'),),],),`

COMPRENSIÓN DE LOS WIDGETS SIN ESTADO Y CON ESTADO

Los widgets de Flutter son los componentes básicos para diseñar la interfaz de usuario (UI). Los widgets se construyen utilizando un marco moderno de estilo de reacción. La interfaz de usuario se crea anidando widgets en un árbol de widgets.

El marco de estilo de reacción de Flutter significa que observa cuándo cambia el estado de un widget y luego lo compara con el estado anterior para determinar la menor cantidad de cambios que se deben realizar. Flutter administra la relación entre el estado y la interfaz de usuario y reconstruye solo esos widgets cuando cambia el estado.

En esta sección, comparará widgets sin estado y con estado y aprenderá cómo implementar cada clase y, lo que es más importante, cuál usar según el requisito. En capítulos posteriores, creará aplicaciones para cada escenario. La clase adecuada se amplía (convirtiéndola en una subclase) utilizando la palabra clave extends seguida de StatelessWidget o StatefulWidget.

`StatelessWidget` se usa cuando los datos no cambian y se basa en la información inicial. Es un widget sin estado y los valores son finales. Algunos ejemplos son Texto, Botón, Icono e Imagen.

```
Las instrucciones de clase extienden StatelessWidget {  
    @anular  
    Compilación del widget (contexto BuildContext) {  
        return Text('Al usar un StatelessWidget...');  
    }  
}
```

`StatefulWidget` se usa cuando cambian los datos. Es un widget con estado que puede cambiar con el tiempo y requiere dos clases. Para que los cambios se propaguen a la interfaz de usuario, hacer una llamada al método `setState()` es necesario.

Clase `StatefulWidget` : crea una instancia de la clase `State` .

Clase de estado : esto es para datos que se pueden leer sincrónicamente cuando se crea el widget y pueden cambiar con el tiempo.

`setState()` : desde dentro de la clase `State` , realiza una llamada al método `setState()` para actualice los datos modificados, diciéndole al marco que el widget debe volver a dibujarse porque el estado ha cambiado. Para todas las variables que necesitan cambios, modifique los valores en `setState()` { `myValue += _ 50.0;`} . Cualquier valor de variable modificada fuera del método `setState()` no actualizar la interfaz de usuario. Por lo tanto, es mejor ubicar los cálculos que no necesitan cambios de estado fuera del método `setState()` .

Considere el ejemplo de una página que muestra su oferta máxima en un producto. Cada vez que se presiona el botón Aumentar oferta, su oferta aumenta en \$50. Comienza creando una clase `MaximumBid` que amplía la clase `StatefulWidget` . Cree una clase `_MaximumBidState` que amplíe el estado de la clase `MaximumBid` .

En la clase `_MaximumBidState` , declara una variable denominada `_maxBid` . El método `_increaseMyMaxBid()` llama al método `setState()` , que incrementa el valor de `_maxBid` en \$50. La interfaz de usuario consiste

de un widget de texto que muestra el valor 'Mi oferta máxima: \$_maxBid' y un FlatButton con una propiedad onPressed que llama al método _increaseMyMaxBid(). El método _increaseMyMaxBid() ejecuta el método setState() , que agrega \$50 a la variable _maxBid y la cantidad del widget de texto se vuelve a dibujar.

```
class MaximumBid extiende StatefulWidget { @override

    _MaximumBidState createState() => _MaximumBidState(); }

class _MaximumBidState extiende State<MaximumBid> { double _maxBid
    = 0.0;

    void _increaseMyMaxBid() { setState(() {
        // Agregar $50 a
        mi oferta actual _maxBid += 50.0; });

    }

    @anular
    Widget build(BuildContext context) { return
        Column( children:
            <Widget>[ Text('My Maximum
                Bid: $_maxBid'), FlatButton.icon( onPressed: () 
                    => _increaseMyMaxBid(),
                    icon: Icon(Icons.add_circle) , etiqueta: Texto('Aumentar
                    oferta'),
                    ),
            ), ], );
    }
}
```

USO DE PAQUETES EXTERNOS

A veces no vale la pena construir un widget desde cero. Flutter admite paquetes de terceros para los ecosistemas Flutter y Dart. Los paquetes contienen la lógica del código fuente y se comparten fácilmente. Hay dos tipos de paquetes, Dart y plugin.

Los paquetes de Dart están escritos en Dart y pueden contener dependencias específicas de Flutter.

Los paquetes de complementos están escritos en Dart (con el código de Dart exponiendo la API), pero se combinan con implementaciones de código específicas de la plataforma para Android (Java o Kotlin) y/o iOS (Objective-C o Swift). La mayoría de los paquetes de complementos tienen como objetivo admitir tanto Android como iOS.

Supongamos que está buscando agregar funcionalidad a su aplicación, como mostrar algunos gráficos, acceder a las ubicaciones de GPS de un dispositivo, reproducir audio de fondo o acceder a una base de datos como Firebase. Hay paquetes para todo eso.

Búsqueda de paquetes En la aplicación,

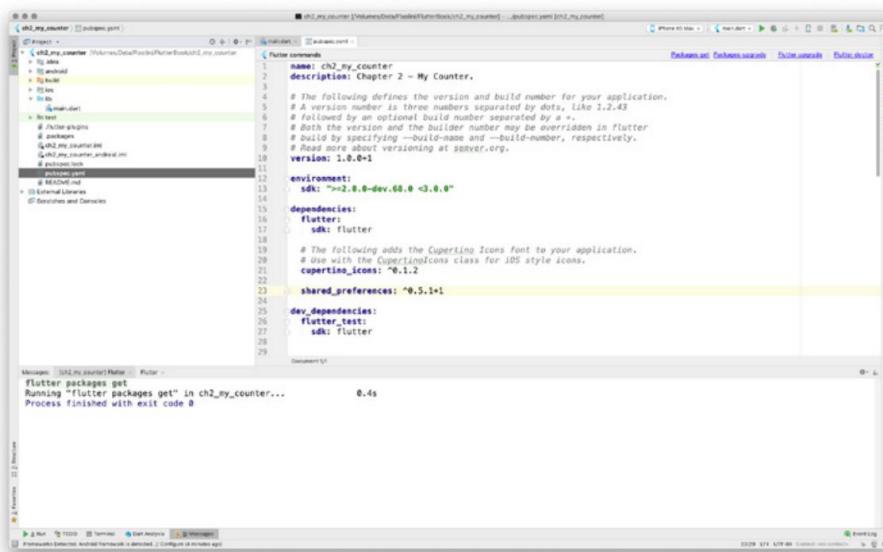
supongamos que necesita almacenar las preferencias del usuario tanto en iOS como en Android y desea encontrar un paquete que lo haga por usted.

1. Inicie su navegador web y vaya a <https://pub.dartlang.org/flutter>. Los paquetes se publican en esta ubicación a menudo por otros desarrolladores y Google.
2. Haga clic en la barra de búsqueda Buscar paquetes de Flutter. Ingrese las preferencias compartidas y los resultados se ordenarán por relevancia.
3. Haga clic en el vínculo del paquete shared_preferences . (El enlace directo es https://pub.dartlang.org/packages/shared_preferences.)
4. Los detalles sobre cómo instalar y usar el paquete shared_preferences están disponibles en esta ubicación. El equipo de Flutter crea este paquete en particular. Haga clic en la pestaña Instalación para obtener instrucciones detalladas. Cada paquete tiene instrucciones sobre cómo instalarlo y usarlo. La instalación de la mayoría de los paquetes es similar, pero difieren en cómo usar e implementar el código. Las instrucciones se encuentran en la página de inicio de cada paquete.

PRUÉBALO Instalación de paquetes

Ha aprendido a encontrar paquetes de terceros. A continuación, aprenderá a implementar el paquete externo shared_preferences en su aplicación.

1. Abra la aplicación ch2_my_counter con Android Studio.
2. Abra el archivo pubspec.yaml haciendo doble clic.
3. En la sección dependencias:, agregue shared_preferences: ^0.5.1+1. (Su versión podría ser más alta.)
4. Guarde el archivo y el paquete se instalará. Si no ve que el proceso se ejecuta automáticamente, puede invocarlo manualmente ingresando los paquetes flutter get en la ventana de su Terminal en Android Studio. Una vez terminado, el mensaje dirá Proceso terminado con código de salida 0.



5. Importe el paquete en el archivo main.dart después de la línea de importación material.dart , ubicada en la parte superior del archivo. Guarde los cambios.

```
import 'package:flutter/material.dart';
import 'package:shared_preferences/shared_preferences.dart';
```

Cómo funciona

Al agregar las dependencias shared_preference en el archivo pubspec.yaml , las dependencias se descargan en el proyecto local. Utilice la declaración de importación para que el paquete shared_preference esté disponible.

Uso de paquetes

Cada paquete tiene su forma única de ser implementado. Siempre es bueno leer la documentación.

Para el paquete shared_preferences , debe agregar algunas líneas para implementarlo. Recuerde que el punto principal aquí no es cómo usar este paquete, sino cómo agregar paquetes externos a su aplicación en general.

PRUÉBALO Implementación e inicialización de un paquete

En la clase _MyHomePageState , agregue una función llamada _updateSharedPreferences().

```
class _MyHomePageState extends State<MyHomePage> {
  // ...
  Future _updateSharedPreferences() async {
    SharedPreferences prefs = await SharedPreferences.getInstance();
    int contador =
        (prefs.getInt('contador') ?? 0) + 1;
    prefs.setInt('contador', contador);
  }
}
```

```
print('Presionó $contador de veces.');
await prefs.setInt('contador', contador);

} // ...
}
```

Cómo funciona

Este paquete guarda las preferencias de los usuarios tanto en iOS como en Android con unas pocas líneas de código Dart, que es extremadamente poderoso. No es necesario escribir código nativo para iOS o Android. Este es el poder de usar paquetes, pero tenga cuidado de no exagerar porque confía en los autores de los paquetes para mantenerlos actualizados.

RESUMEN

En este capítulo, aprendió cómo crear su primera aplicación y usar la recarga en caliente para ver los cambios instantáneamente. También vio cómo usar temas para diseñar aplicaciones, cuándo usar widgets sin estado y con estado, y cómo agregar paquetes externos para evitar reinventar la rueda. Ahora tiene una comprensión general de las ideas principales detrás del desarrollo de la aplicación Flutter. No se preocupe por entender el código real todavía. Aprenderás todo sobre esto a lo largo del libro.

En el próximo capítulo, aprenderá los conceptos básicos del lenguaje Dart que se usa para crear aplicaciones de Flutter.

LO QUE APRENDISTE EN ESTE CAPÍTULO

TEMA	CONCEPTOS CLAVE
Creando un aplicación de aleteo	Ahora puede codificar una aplicación básica y diseñar widgets.
Usando recarga caliente	La recarga en caliente muestra los cambios de código inmediatamente en una aplicación en ejecución mientras mantiene el estado.
Aplicar un tema	Los temas establecen el estilo y los colores en una aplicación.
Uso de paquetes externos	Los paquetes externos buscan e instalan paquetes de terceros para agregar funcionalidades como GPS, gráficos y más.

3

Aprendiendo los conceptos básicos de los dardos

LO QUE APRENDERÁS EN ESTE CAPÍTULO

Por qué usas Dart

Cómo comentar el código

Cómo usar la función main() de nivel superior

Cómo hacer referencia a variables como números, cadenas, booleanos, listas, mapas y runas

Cómo funcionan las declaraciones de flujo comunes (como if, for, while y el operador ternario), los bucles y el cambio y el caso

Cómo se usan las funciones para agrupar la lógica reutilizable

Cómo usar la declaración de importación para paquetes, bibliotecas o clases externas

Cómo crear clases

Cómo usar la programación asíncrona para evitar bloquear la interfaz de usuario

Dart es la base para aprender a desarrollar proyectos Flutter. En este capítulo, comenzará a comprender la estructura básica de Dart. En capítulos futuros, creará aplicaciones que implementen estos conceptos. Todo el código de muestra está en la carpeta ch3_dart_basics . (En el código de muestra, no se preocupe por cómo está diseñado; solo eche un vistazo para que pueda ver cómo está escrito el código Dart. Toque el botón de acción flotante ubicado en la parte inferior derecha para ver los resultados del registro).

¿POR QUÉ USAR DARDO?

Antes de que pueda comenzar a desarrollar aplicaciones de Flutter, debe comprender el lenguaje de programación utilizado, es decir, Dart. Google creó Dart y lo usa internamente con algunos de sus grandes productos, como Google AdWords. Disponible públicamente en 2011, Dart se usa para construir

Aplicaciones móviles, web y de servidor. Dart es productivo, rápido, portátil, accesible y, sobre todo, reactivo.

Dart es un lenguaje de programación orientado a objetos (OOP), tiene un estilo basado en clases y utiliza una sintaxis de estilo C. Si está familiarizado con los lenguajes C#, C++, Swift, Kotlin y Java/JavaScript, podrá comenzar a desarrollar en Dart rápidamente. Pero no se preocupe, incluso si no está familiarizado con estos otros idiomas, Dart es un idioma fácil de aprender y puede comenzar con relativa rapidez.

¿Cuáles son algunos de los beneficios de usar Dart?

Dart se compila con anticipación (AOT) en código nativo, lo que hace que su aplicación Flutter sea más rápida. En otras palabras, no hay intermediario para interpretar un idioma a otro, y no hay puentes. La compilación AOT se usa al compilar su aplicación para el modo de lanzamiento (como en Apple App Store y Google Play).

Dart también se compila justo a tiempo (JIT), lo que agiliza la visualización de los cambios en el código, como a través de la función de recarga en caliente con estado de Flutter. La compilación JIT se usa al depurar su aplicación ejecutándola en el simulador/emulador.

Dado que Flutter usa Dart, todo el código de la interfaz de usuario (UI) de muestra en este libro está escrito en Dart, lo que elimina la necesidad de usar lenguajes separados (marcado, diseñador visual) para crear la UI.

La representación de Flutter se ejecuta a 60 cuadros por segundo (fps) y 120 fps (para dispositivos compatibles con 120 Hz). Cuantos más fps, más fluida es la aplicación.

CÓDIGO DE COMENTARIO

En cualquier aplicación, los comentarios ayudan a la legibilidad del código, siempre que no se exageren. Los comentarios se pueden utilizar para describir la lógica y las dependencias de la aplicación.

Hay tres tipos de comentarios: comentarios de una sola línea, de varias líneas y de documentación. Los comentarios de una sola línea se usan comúnmente para agregar una breve descripción. Los comentarios de varias líneas son más adecuados para descripciones largas que abarcan varias líneas. Los comentarios de documentación se utilizan para documentar completamente una parte de la lógica del código, por lo general brindan explicaciones detalladas y código de muestra en los comentarios.

Los comentarios de una sola línea comienzan con // y el compilador de Dart ignora todo hasta el final de la línea.

```
// Recuperar de la base de datos la lista filtrada por empresa _listOrders.get(...)
```

Los comentarios de varias líneas comienzan con /* y terminan con */. El compilador de Dart ignora todo lo que se encuentra entre las barras.

```
/*
 * Permitir a los usuarios filtrar por múltiples opciones
 _listOrders.get(filterBy: _userFilter... *)
```

Los comentarios de la documentación comienzan con ///, y el compilador de Dart ignora todo hasta el final de la línea a menos que esté entre corchetes. Usando corchetes, puede referirse a clases, métodos, campos, nivel superior

variables, funciones y parámetros. En el siguiente ejemplo, la documentación generada, [FilterBy], se convierte en un enlace a la documentación de la API para la clase. Puede utilizar la herramienta de generación de documentación del SDK (dartdoc) para analizar el código Dart y generar documentación HTML.

```
// Múltiples opciones de filtro //

/// Diferente [FilterBy] enum FilterBy {

    COMPAÑÍA,
    CIUDAD,
    ESTADO
}
```

EJECUTAR EL PUNTO DE ENTRADA PRINCIPAL()

Cada aplicación debe tener una función main() de nivel superior , que es el punto de entrada a la aplicación. La función main() es donde comienza la ejecución de la aplicación y devuelve un vacío con un parámetro List<String> opcional para los argumentos. Cada función puede devolver un valor y, para la función main(), el tipo de devolución de datos es un vacío (vacío, no contiene nada), lo que significa que no devuelve ningún valor.

En el siguiente código, verá tres formas diferentes de usar la función main() , pero en todos los proyectos de ejemplo de este libro, usará el primer ejemplo: la sintaxis de flecha void main() => runApp(MyApp());. Las tres formas de llamar a la función main() son aceptables, pero prefiero usar la sintaxis de flecha ya que mantiene el código en una línea para una mejor legibilidad. Sin embargo, la razón principal para usar la sintaxis de flecha es que en todos los proyectos de ejemplo no es necesario llamar a varias declaraciones.

La sintaxis de la flecha => runApp(MyApp()) es la misma que { runApp(MyApp()); }.

```
// sintaxis de flecha void
main() => runApp(MyApp());

// o void
main()
{ runApp(MyApp());
}

// o con una lista de parámetros de cadenas void
main(List<Strings> filtros) { print('filtros: $filtros');

}
```

VARIABLES DE REFERENCIA

En la sección anterior, aprendió que main() es la entrada de nivel superior a una aplicación y, antes de comenzar a escribir código, es importante conocer las variables de Dart. Las variables almacenan referencias a un valor. Algunos de los tipos de variables integradas son números, cadenas, booleanos, listas, mapas y runas. Puede usar var para declarar (aprenderá a declarar variables en la siguiente sección) una variable sin especificar el tipo. Dart infiere el tipo de variable automáticamente. Aunque no hay nada de malo en usar var, como preferencia personal, generalmente me abstengo de usarlo a menos que sea necesario. Declarando el

El tipo de variable mejora la legibilidad del código y es más fácil saber qué tipo de valor se espera. En lugar de usar var, use el tipo de variable esperado: doble, Cadena, etc. (Los tipos de variables se tratan en la sección "Declaración de variables").

Una variable no inicializada tiene un valor nulo. Cuando se declara una variable sin darle un valor inicial, se llama no inicializada. Por ejemplo, una variable de tipo String se declara como String título de libro; y no está inicializado porque el valor del título del libro es nulo (sin valor). Sin embargo, si lo declara con un valor inicial de String bookTitle = 'Beginning Flutter', el valor bookTitle es igual a 'Beginning Flutter'.

Use final o const cuando no se pretende que la variable cambie el valor inicial. Use const para las variables que deben ser constantes en tiempo de compilación, lo que significa que el valor se conoce en el momento de la compilación.

DECLARACIÓN DE VARIABLES

Ahora sabes que las variables almacenan referencias a un valor. A continuación, aprenderá diferentes opciones para declarar variables.

En Dart, todas las variables se declaran públicas (disponibles para todos) de forma predeterminada, pero al comenzar el nombre de la variable con un guión bajo (_), puede declararlas como privadas. Al declarar una variable privada, está diciendo que no se puede acceder a ella desde clases/funciones externas; en otras palabras, solo se puede usar desde dentro de la clase/función de declaración. (Aprenderá sobre clases y funciones en las secciones "Funciones" y "Clases" más adelante en este capítulo). Tenga en cuenta que algunos tipos de variables de Dart incorporados están en minúsculas como double y algunos en mayúsculas como String .

¿Qué pasa si el valor de una variable no necesita cambiar? Comience la declaración de la variable con final o const. Use final cuando el valor se asigna en tiempo de ejecución (el usuario puede cambiarlo). Use const cuando el valor se conoce en tiempo de compilación (en código) y no cambiará en tiempo de ejecución.

```
// Declarado sin especificar el tipo - Infiere tipo var filter = 'company';
```

```
// Declarado por tipo
Filtro de cadena = 'empresa';
```

```
// La variable no inicializada tiene un valor inicial de nulo
filtro de cadena;
```

```
// El valor no cambiará filtro final
= 'empresa';
```

```
// o
filtro de cadena final = 'empresa';
```

```
// o
const String filter = 'empresa';
```

```
// o
const String filter = 'empresa' + filterOption;
```

```
// Variable pública (el nombre de la variable comienza sin guión bajo)
Cadena nombre de usuario = 'Sandy';

// Variable privada (el nombre de la variable comienza con un guión bajo)
Cadena _IDusuario = 'XW904';
```

Números

Declarar variables como números restringe los valores a números únicamente. Dart permite que los números sean int (enteros) o dobles. Use la declaración int si sus números no requieren precisión de punto decimal, como 10 o 40. Use la declaración double si sus números requieren precisión de punto decimal, como 50.25 o 135.7521. Tanto int como double permiten números positivos y negativos, y puede ingresar números extremadamente grandes y precisión decimal ya que ambos usan valores de 64 bits (memoria de computadora).

```
// Entero int
contador = 0; precio
doble = 0.0; precio = 125,00;
```

Instrumentos de cuerda

Declarar variables como String permite que los valores se ingresen como una secuencia de caracteres de texto. Para agregar una sola línea de caracteres, puede usar comillas simples o dobles como 'car' o "car". Para agregar caracteres de varias líneas, use comillas triples, como "coche". Las cadenas se pueden concatenar (combinar) usando el operador más (+) o usando comillas simples o dobles adyacentes.

```
// Cadenas
Cadena defaultMenu = 'principal';

// concatenación de cadenas
Cadena nombreCombinado = 'principal' +     ' ' + 'función';
String nombreCombinadoNoPlusSign = 'principal'           'función';

// Cadena multilinea
String multilineAddress =
    123 cualquier calle
    Código postal "";
```

Booleanos

Declarar variables como bool (booleano) permite ingresar un valor de verdadero o falso .

```
// Booleanos
bool isDone = false;
estáTerminado = verdadero;
```

Liza

Declarar variables como Lista (comparable a matrices) permite ingresar múltiples valores; una Lista es un grupo ordenado de objetos. En programación, una matriz es una colección iterable (a la que se accede secuencialmente) de objetos, con cada elemento accesible por la posición del índice o una clave. Para acceder a los elementos, la Lista utiliza

indexación basada en cero, donde el índice del primer elemento es 0 y el último elemento tiene la longitud de la lista (número de filas) menos 1 (ya que el primer índice es 0, no 1).

Una Lista puede ser de longitud fija o ampliable, según sus necesidades. De forma predeterminada, una lista se crea como ampliable mediante el uso de List() o []. Para crear una Lista de longitud fija , agregue la cantidad de filas requeridas utilizando este formato: Lista (25). El siguiente ejemplo utiliza la interpolación de cadenas para la declaración de impresión : print('filtro: \$filtro'). El signo \$ antes de la variable convierte el valor de la expresión en una cadena.

```
// Lista cultivable
Lista de contactos = Lista ();

// o
Listar contactos = [];
Listar contactos = ['Linda', 'Juan', 'María'];

// Lista de longitud fija
Lista de contactos = Lista (25);

// Listas - En Dart List hay una matriz List listOfFilters
= ['company', 'city', 'state']; listOfFilters.forEach((filtro) { print('filtro:
$filtro'); });
// Resultado de la declaración de impresión // filtro: empresa // filtro:
ciudad // filtro: estado
```

mapas

Los mapas son invaluables para asociar una lista de valores por una clave y un valor. El mapeo permite recuperar valores por su Key ID. La clave y el valor pueden ser cualquier tipo de objeto, como cadena, número, etc. Tenga en cuenta que la clave debe ser única, ya que la clave recupera el valor .

```
// Mapas: un objeto que asocia claves y valores.
// Clave: Valor - "ValorClave": 'Valor'
Mapa mapOfFilters = {'id1': 'compañía', 'id2': 'ciudad', 'id3': 'estado'};

// Cambiar el valor del tercer elemento con Clave de id3 mapOfFilters['id3']
= 'my filter';

print('Obtener filtro con id3: ${mapOfFilters['id3']}');
// Resultado de la declaración de impresión
// Obtener filtro con id3: mi filtro
```

runas

En Dart, declarar variables como Runas son los puntos de código UTF-32 de una Cadena. ¿Emojis, alguien?

Unicode define un valor numérico para cada letra, dígito y símbolo. Dart usa la secuencia de unidades de código UTF-16 para representar un valor Unicode de 32 bits de una cadena que requiere una sintaxis especial (uXXXX).

Un punto de código Unicode es \uXXXX, donde XXXX es un valor hexadecimal de cuatro dígitos. Las runas devuelven el

valor entero de Unicode; luego usa `String.fromCharCodes()` para asignar una nueva cadena para el charCode especificado.

```
// Emoji ángel sonriente Unicode es u+1f607
// Elimina el signo más y reemplázalo con corchetes
Runas myEmoji = Runas("\u{1f607}");
imprimir(miEmoji);
// Resultado de la declaración de
impresión // (128519)

imprimir(String.fromCharCodes(miEmoji)); //
Resultado de la declaración de
impresión //
```

USO DE OPERADORES

Un operador es un símbolo que se utiliza para realizar notaciones aritméticas, de igualdad, relacionales, de prueba de tipos, de asignación, lógicas, condicionales y en cascada. Las tablas 3.1 a 3.7 repasan algunos de los operadores comunes. Para el código de muestra, uso los valores directamente para simplificar los ejemplos en lugar de usar variables.

TABLA 3.1: Operadores aritméticos

OPERADOR	DESCRIPCIÓN	CÓDIGO DE MUESTRA
+	Agregar	$7 + 3 = 10$
-	Sustraer	$7 - 3 = 4$
*	Multiplicar	$7 * 3 = 21$
/	Dividir	$7 / 3 = 2,33$

TABLA 3.2: Operadores de igualdad y relacionales

OPERADOR	DESCRIPCIÓN	CÓDIGO DE MUESTRA
==	Igual	$7 == 3 = \text{falso}$
!=	No es igual	$7 != 3 = \text{verdadero}$
>	Más grande que	$7 > 3 = \text{verdadero}$
<	Menos que	$7 < 3 = \text{falso}$
>=	Mayor que o igual a	$7 >= 3 = \text{verdadero}$ $4 >= 4 = \text{verdadero}$
<=	Menos que o igual a	$7 <= 3 = \text{falso}$ $4 <= 4 = \text{verdadero}$

TABLA 3.3: Operadores de prueba tipo

OPERADOR	DESCRIPCIÓN	CÓDIGO DE MUESTRA
como	Typecast como prefijos de biblioteca de importación.	importar 'puntos de viaje'.dardo' como viaje;
es	Si el objeto contiene el tipo especificado, se evalúa como verdadero.	si (puntos es Lugares) = verdadero
¡es!	Si el objeto contiene el tipo especificado, se evalúa como falso (no se usa normalmente).	si (puntos es! Lugares) = falso

TABLA 3.4: Operadores de asignación

OPERADOR	DESCRIPCIÓN	CÓDIGO DE MUESTRA
=	Asigna valor	7 = 3 = 3
??=	Asigna valor solo si la variable a la que se asigna tiene un valor de nulo	Nulo ??= 3 = 3 7 ??= 3 = 7
+=	Se suma al valor actual	7 += 3 = 10
-=	Resta del valor actual	7 -= 3 = 4
*=	Multiplica del valor actual	7 *= 3 = 21
/=	Divide del valor actual	7 /= 3 = 2,33

TABLA 3.5: Operadores lógicos

OPERADOR	DESCRIPCIÓN	CÓDIGO DE MUESTRA
!	! es un 'no' lógico. Devuelve el valor opuesto de la variable/expresión.	si (!(7 > 3)) = falso
&&	&& es un 'y' lógico. Devuelve verdadero si los valores de la variable/expresión son todos verdaderos.	si ((7 > 3) && (3 < 7)) = verdadero si ((7 > 3) && (3 > 7)) = falso
!!	!! es un 'o' lógico. Devuelve verdadero si al menos un valor de la variable/expresión es verdadero.	si ((7 > 3) (3 > 7)) = verdadero si ((7 < 3) (3 > 7)) = falso

TABLA 3.6: Expresiones condicionales

OPERADOR	DESCRIPCIÓN	CÓDIGO DE MUESTRA
condición ? valor1: valor2	Si la condición se evalúa como verdadera, devuelve valor1. Si la condición se evalúa como falsa, devuelve valor2. El valor también se puede obtener llamando a métodos.	(7 > 3) ? verdadero : falso = verdadero (7 < 3) ? verdadero : falso = falso

TABLA 3.7: Notación en cascada (...)

OPERADOR	DESCRIPCIÓN	CÓDIGO DE MUESTRA
..	La notación en cascada está representada por puntos dobles (...) y le permite realizar una secuencia de operaciones en el mismo objeto.	Matrix4.identidad() ..escala(1.0, 1.0) ..traducir(30, 30);

USO DE DECLARACIONES DE FLUJO

Para controlar el flujo lógico del código Dart, eche un vistazo a las siguientes declaraciones de flujo:

if y else son las declaraciones de flujo más comunes; ellos deciden qué código ejecutar comparando múltiples escenarios.

El operador ternario es similar a las declaraciones if y else , pero se usa cuando solo se necesitan dos opciones.

los bucles for permiten iterar una lista de valores.

while y do-while son un par común. Use el ciclo while para evaluar la condición antes de ejecutar el ciclo y use do-while para evaluar la condición después del ciclo.

while y break son útiles si necesita dejar de evaluar la condición en el bucle.

continuar es para cuando necesita detener el ciclo actual y comenzar la siguiente iteración del ciclo.

switch y case son alternativas a las sentencias if y else , pero requieren un cláusula por defecto.

si y mas

La instrucción if compara una expresión y, si es verdadera, ejecuta la lógica del código. La expresión está envuelta por paréntesis de apertura y cierre, seguida de la lógica del código entre llaves. La declaración if también admite varias declaraciones else opcionales , que se utilizan para evaluar varios escenarios.

Hay dos tipos de sentencias else : else if y else. Puede usar varias declaraciones else if , pero solo puede tener una declaración else , que generalmente se usa como un escenario general.

En el siguiente ejemplo, la declaración if verifica si la tienda está abierta o cerrada y si los artículos están agotados o no coinciden. `isClosed`, `isOpen` y `isOutOfStock` son variables booleanas . La primera instrucción if verifica si la variable `isClosed` es verdadera y, en caso afirmativo, imprime en el registro 'La tienda está cerrada'. ¿ Cómo sabe que está comprobando si es verdadero o falso sin el operador de igualdad? Al verificar los valores booleanos , la instrucción if verifica de forma predeterminada si la variable es verdadera; este es el equivalente de `isClosed == true`. Para comprobar si una variable es falsa, puede utilizar el operador no igual (`!=`) como `isClosed != true` o el operador de igualdad (`==`) como `isClosed == false`. La declaración else if (`isOpen`) verifica si la variable `isOpen` es igual a verdadero, y es lo mismo para la variable else if (`isOutOfStock`) . La última sentencia else no tiene condición; es un escenario general si no se cumple ninguna de las otras condiciones.

```
// If y else if
(isClosed) { print('La
tienda está cerrada');

} else if (isOpen)
{ print('La tienda está abierta');

} else if (isOutOfStock) { print('El
artículo está agotado');

} else
{ print('Nada coincide');
}
```

Operador ternario El operador

ternario toma tres argumentos y generalmente se usa cuando solo se necesitan dos acciones.

El operador ternario verifica el primer argumento para la comparación, el segundo es la acción si el argumento es verdadero y el tercero es la acción si el argumento es falso (ver Tabla 3.8).

TABLA 3.8: operador ternario

COMPARACIÓN		VERDADERO		FALSO
está cerrado	?	preguntarParaAbrir()	:	preguntarParaCerrar()

Esto le resultará familiar por las expresiones condicionales de la sección "Operadores" porque se usa a menudo para tomar decisiones de flujo de código.

```
// ¿La forma más corta de la declaración if y else
isClosed? preguntarParaAbrir() : preguntarParaCerrar();
```

para bucles

El bucle for estándar le permite iterar una lista de valores. Los valores se obtienen restringiendo el número de bucles por una longitud definida. Un ejemplo es recorrer los tres valores principales, lo que significa que especifica la cantidad de veces que se ejecutará el ciclo. Usar una lista de valores también le permite usar el tipo de iteración for-in . La clase de iteración debe ser del tipo Iterable (una colección

de valores), y la clase List se ajusta a este tipo. A diferencia del bucle for estándar , el bucle for-in itera a través de cada objeto en la Lista, exponiendo los valores de las propiedades de cada objeto.

Echemos un vistazo a dos ejemplos que muestran cómo usar el bucle for estándar y los bucles for-in .

En el primer ejemplo, usará el bucle for estándar e iterará a través de una lista de valores de cadena con la variable listOffilters . El bucle for estándar toma tres parámetros.

El primer parámetro inicializa la variable i como una variable int contando cada bucle ejecutado.

Dado que la Lista utiliza una indexación basada en cero, la variable i se inicializa con 0 y no con 1.

El segundo parámetro controla cuántas veces recorrer la Lista comparando el número actual de bucles (i) con el número total de bucles (listOffilters.length) para ejecutar. Dado que la Lista usa indexación basada en cero, el valor de la variable i debe ser menor que el número de filas en la Lista.

El tercer parámetro aumenta el número de bucles ejecutados aumentando la variable i con cada bucle.

Dentro del ciclo, la declaración de impresión se usa para mostrar cada valor del listOffilters Lista.

```
// Estándar para lista de
bucle listOffilters = ['compañía', 'ciudad', 'estado']; for (int i = 0; i
< listaDeFilters.longitud; i++) { print('listaDeFilters: $
{listaDeFilters[i]}');

}
// Resultado de la declaración de
impresión // listOffilters:
empresa // listOffilters:
ciudad // listOffilters: estado
```

En el siguiente ejemplo, usará el bucle for-in e iterará a través de una lista de valores int con la variable listofNumbers . El bucle for-in toma un parámetro que expone las propiedades del objeto (listOf Numbers) . Declara la variable de número int para acceder a las propiedades de la lista listofNumbers . Dentro del ciclo, la declaración de impresión se usa para mostrar cada valor de listofNumbers usando el valor de la variable numérica .

```
// o bucle for-in
List listofNumbers = [10, 20, 30]; for (int
número en listaDeNúmeros) { print('número:
$número');

}
// Resultado de la declaración de
impresión //
número: 10 //
número: 20 // número: 30
```

mientras y hacer mientras

Tanto el bucle while como el do-while evalúan una condición y continúan en bucle mientras la condición devuelva un valor verdadero. El ciclo while evalúa la condición antes de que se ejecute el ciclo.

El bucle do-while evalúa la condición después de que el bucle se ejecuta al menos una vez. Veamos dos ejemplos que muestran cómo usar los bucles while y do-while .

En ambos ejemplos, se llama al método `askToOpen()` en el ciclo, ejecutando la lógica que establece la variable `isClosed` como un valor bool de verdadero o falso. Use el ciclo `while` si ya tiene suficiente información para evaluar la condición (`isClosed`) antes de que se ejecute el ciclo. Utilice `do-while` si necesita ejecutar el ciclo primero antes de tener suficiente información para evaluar la condición (`isClosed`).

En el primer ejemplo, usará el ciclo `while` e iterará siempre que la variable `isClosed` devuelva un valor verdadero. En este caso, el ciclo continúa ejecutándose mientras la variable `isClosed` sea verdadera y continúe el ciclo. Una vez que la variable `isClosed` devuelve falso, el `while` deja de ejecutar el siguiente bucle.

```
// Mientras - evalúa la condición antes del bucle while (isClosed)
{ askToOpen(); }

}
```

En el segundo ejemplo, usará el bucle `do-while` e iterará siempre que la variable `isClosed` devuelva un valor verdadero, como en el primer ejemplo. El ciclo se ejecuta primero al menos una vez; luego se evalúa la condición y, mientras devuelva verdadero, continúa en bucle. Una vez que la variable `isClosed` devuelve falso, `do-while` deja de ejecutar el siguiente bucle.

```
// Do While - evalúa la condición después del ciclo do

{ askToOpen(); }
while (está cerrado);
```

mientras y romper

El uso de la instrucción `break` le permite detener el ciclo al evaluar una condición dentro del ciclo `while`.

En este ejemplo, la instrucción `if` llama al método `askToOpen()` dentro del ciclo , ejecutando la lógica que devuelve un valor bool de verdadero o falso. Siempre que el valor devuelto sea falso, el ciclo continúa con normalidad llamando al método `checkForNewOrder()` . Pero una vez que `askToOpen()` devuelve un valor de verdadero, se ejecuta la instrucción `break` , deteniendo el ciclo. No se llama al método `checkForNewOrder()` y toda la instrucción `while` deja de ejecutarse nuevamente.

```
// Break - para detener el
ciclo while (isClosed)
{ if (askToOpen()) break;
  comprobarParaNuevoPedido();
}
```

continuar

Al usar la instrucción `continuar` , puede detenerse en la ubicación del bucle actual y saltar al inicio de la siguiente iteración del bucle.

En este ejemplo, la sentencia `for` recorre una lista de números del 10 al 80. Dentro del bucle, la sentencia `if` comprueba si el número es menor que 30 y mayor que 50 y, si se cumple la condición, la sentencia `continue` detiene el bucle actual y comienza la siguiente iteración. Usando la declaración de impresión , verá que solo los números 30, 40 y 50 se imprimen en el registro.

```
// Continuar - saltar a la siguiente iteración del bucle
List listOfNumbers = [10, 20, 30, 40, 50, 60, 70, 80]; for (int número en
listOfNumbers) { if (número < 30 || número > 50)
    { continuar;

    } print('numero: $numero'); // Imprimirá el número 30, 40, 50
}
```

interruptor y caja

La instrucción switch compara constantes de tiempo de compilación, cadena o enteros usando == (igualdad). La sentencia switch es una alternativa a las sentencias if y else . La declaración de cambio evalúa una expresión y usa la cláusula de caso para hacer coincidir una condición y ejecuta el código dentro del caso coincidente. Cada cláusula de caso termina colocando una declaración de ruptura como la última línea. No se usa comúnmente, pero si tiene una cláusula de caso vacía (sin código) , la declaración de ruptura no es necesaria ya que Dart permite que falle. Si necesita un escenario general, puede usar la cláusula predeterminada para ejecutar código que no coincide con ninguna de las cláusulas de caso , ubicadas después de todas las cláusulas de caso . La cláusula predeterminada no requiere una declaración de interrupción . Asegúrese de que el último caso sea una cláusula predeterminada que ejecute la lógica si ninguna cláusula del caso anterior coincide.

En nuestro ejemplo, tenemos la variable café String inicializada con el valor 'espresso' . La declaración de cambio utiliza la expresión de variable de café donde cada cláusula de caso debe coincidir con el valor de la variable de café . Cuando la cláusula case coincide con el valor correcto, se ejecuta el código asociado con la cláusula. Si ninguna de las cláusulas de caso coincide con el valor de la variable café , se selecciona la cláusula predeterminada y se ejecuta el código asociado.

```
// cambiar y caso
Cadena café = 'espresso'; cambiar (café)
{
    caso 'con sabor':
        ordenar con sabor();
        romper;
    caso 'tostado oscuro':
        ordenarAsadoOscuro();
        romper
    estuche 'espresso':
        ordenarEspresso();
        romper;

    predeterminado: orderNotAvailable();
}
```

USO DE FUNCIONES

Las funciones se utilizan para agrupar la lógica reutilizable. Una función puede opcionalmente tomar parámetros y devolver valores. Me encanta esta función. Debido a que Dart es un lenguaje orientado a objetos, las funciones se pueden asignar a variables o pasar como argumentos a otras funciones. Si la función ejecuta una sola expresión, puede usar la sintaxis de flecha (=>) . Todas las funciones devuelven un valor por defecto, y si no hay declaración de retorno

se especifica, Dart agrega automáticamente al cuerpo de la función la instrucción `return null`, que se agrega implícitamente para usted.

Dado que todas las funciones devuelven un valor, cada función se inicia especificando el tipo de devolución esperado. Cuando llame a una función y no se necesite un valor de retorno, inicie la función con el tipo `void`, que no significa nada. No se requiere el uso del tipo `void`, pero se recomienda para la legibilidad. Pero cuando se espera que la función devuelva un valor, inicie la función con el tipo de datos que se devuelven (`bool`, `int`, `String`, `List`...) y use la declaración de retorno para pasar un valor.

Los siguientes ejemplos muestran diferentes formas de crear/lamar funciones y devolver diferentes tipos de valores. El primer ejemplo muestra que el `main()` de la aplicación es una función con `void` como tipo de devolución.

```
// Funciones - Nuestro main() es una función void main()
=> runApp(new MyApp());
```

El segundo ejemplo tiene un vacío como tipo de retorno, pero la función toma un `int` como parámetro, y cuando se ejecuta el código, la declaración de impresión muestra el valor en el terminal de registro. Dado que la función espera un parámetro, la llama pasando el valor como `orderEspresso(3)`.

```
// Función - pasar valor void
orderEspresso(int cuantasTazas) { print('Tazas #:
$cuantasTazas');

} ordenarCafé Expreso(3);
// Resultado de la declaración de impresión
// Tazas #: 3
```

El tercer ejemplo se basa en el segundo ejemplo de recibir un parámetro y devolver un valor `bool` como tipo de retorno. Justo después de la función, se inicializa una variable `bool isOrderDone` llamando a la función y pasando un valor de tres; luego, la declaración de impresión muestra el valor `bool` devuelto por la función.

```
// Función - pasar valor y devolver valor bool orderEspresso(int
cuantasTazas) { print('Tazas #: $cuantasTazas');
devolver verdadero;

} bool isOrderDone = orderEspresso(3); print('Pedido
Realizado: $isOrderDone');
// Resultado de la declaración de impresión
// Tazas #: 3
// Pedido hecho: verdadero
```

El cuarto ejemplo se basa en el tercer ejemplo al hacer que el parámetro de la función sea opcional al envolver la variable `[int howManyCups]` entre corchetes.

```
// Función: pasa el valor opcional y el valor de retorno // El valor opcional
está entre corchetes [] bool orderEspresso1([int howManyCups]) { print('Tazas
#: $howManyCups'); bool ordenado = falso; if (cuantas
tazas! = nulo) { ordenado = verdadero;

} devolución ordenada;
```

```
    } bool isOrderDone1 = orderEspresso1();
    print('Pedido Hecho1: $esPedidoHecho1');
    // Resultado de la declaración de impresión
    // Copas #: nulo
    // Pedido hecho: falso
```

PAQUETES DE IMPORTACIÓN

Para usar un paquete externo, una biblioteca o una clase externa, use la declaración de importación . Separar la lógica del código en diferentes archivos de clase le permite separar y agrupar el código en objetos manejables. La declaración de importación permite el acceso a paquetes y clases externos. Solo requiere un argumento, que especifica el identificador uniforme de recursos (URI) de la clase/biblioteca. Si la biblioteca es creada por un administrador de paquetes, entonces especifica el paquete: esquema antes del URI. Si importa una clase, especifique la ubicación y el nombre de la clase o el paquete: directiva.

```
// Importar el paquete de materiales import
'package:flutter/material.dart';

// Importar clase externa import
'charts.dart';

// Importar clase externa en una carpeta diferente import 'services/
charts_api.dart';

// Importar clase externa con paquete: directiva import
'package:project_name/services/charts_api.dart';
```

CLASES DE USO

Todas las clases descienden de Object, la clase base para todos los objetos Dart. Una clase tiene miembros (variables y métodos) y usa un constructor para crear un objeto. Si no se declara un constructor, se proporcionará automáticamente un constructor predeterminado. El constructor predeterminado que se le proporcionó no tiene argumentos.

¿Qué es un constructor y por qué es necesario? Un constructor tiene el mismo nombre que la clase, con parámetros opcionales. Los parámetros sirven como captadores de valores al inicializar una clase por primera vez. Dart usa azúcar sintáctico para facilitar el acceso a los valores usando la palabra clave this , refiriéndose al estado actual de la clase.

```
// Getter
this.tipo = tipo;

// Azúcar sintáctico
this.type;
```

Una clase básica con un constructor tendría este diseño simple:

```
clase fruta {
    tipo de cadena;

    // Constructor - Mismo nombre que la clase
    Fruit(this.type);
}
```

El ejemplo anterior usa un constructor con azúcar sintáctico, `Fruit(this.type)`, y el constructor se llama de esta manera: `Fruit = Fruit('apple');`. Para usar parámetros con nombre, encierre el parámetro entre corchetes, `Fruit({this.type})`, y llame al constructor de esta manera: `Fruit = Fruit(type: 'Apple');`. Imagine pasar tres o cuatro parámetros; Prefiero usar parámetros con nombre para mantener el código legible. Cada parámetro es opcional a menos que especifique con `@required` que es un parámetro obligatorio.

```
// Parámetro requerido  
Fruit({@required this.type});  
  
// Constructor - Con nombre de parámetro opcional en init Fruit({this.type});
```

Además de marcar un parámetro como `@requerido`, puede agregar la declaración de afirmación para mostrar un error si falta un valor. La declaración de afirmación arroja un error durante el modo de desarrollo (depuración) y no tiene ningún efecto en el código de producción (lanzamiento).

```
// Constructor - Parámetro requerido más clase de aserción Fruit  
{  
    tipo de cadena;  
  
    Fruit({@requerido this.type}) : assert(type != null); }  
  
// Llamar a la clase Fruit Fruit  
fruit = Fruit(type: 'Apple'); print('fruta.tipo: $  
{fruta.tipo}'); En una clase, los métodos son
```

funciones que proporcionan lógica a un objeto. Los métodos pueden devolver un valor o anularse (sin valor de retorno, vacío).

```
// Método en la clase  
calcularFruitCalories() {  
    // Lógica para calcular calorías  
}
```

Veamos un ejemplo de una clase sin constructor y dos ejemplos de una clase con un constructor y un constructor con nombre.

Primero, veamos cómo crear una clase que no defina un constructor y declare dos variables para contener el nombre y la experiencia del barista. Dado que el ejemplo no declara un constructor, se proporciona un constructor predeterminado sin ningún argumento. ¿Qué quiere decir esto? Dentro de la clase, es lo mismo que si hubiera escrito `BaristaNoConstructor()`, que es un constructor predeterminado sin argumentos. Crea una instancia de la clase declarando una variable `BaristaNoConstructor` `baristaNo Constructor` inicializada con `BaristaNoConstructor()`, que es el constructor predeterminado que se le proporciona. Al tomar la variable `baristaNoConstructor`, puede usar el operador de punto como `baristaNoConstructor.experience` y darle un valor de 10.

```
// Declarar clases  
  
// Class Default No Arguments Constructor class  
BaristaNoConstructor {  
    Nombre de  
    cadena; experiencia interna;  
}
```

```
// Class Default No Arguments Constructor BaristaNoConstructor  
baristaNoConstructor = BaristaNoConstructor(); baristaNoConstructor.experience = 10;  
print('baristaNoConstructor.experiencia: $  
{baristaNoConstructor.experiencia}'); // baristaNoConstructor.experiencia: 10
```

A continuación, veamos cómo crear una clase que defina un constructor con parámetros con nombre que contengan el nombre y la experiencia del barista. Este ejemplo muestra cómo agregar un método whatIsTheExperience() que devuelve el valor de la variable de experiencia de la clase . Creas una instancia de la clase declarando una variable barista BaristaWithConstructor inicializada con el constructor BaristaWithConstructor(name: 'Sandy', experience: 10) . Los beneficios son inmediatamente obvios cuando se crea una clase con un constructor. Puede inicializar las variables de cada clase pasando los valores a través del constructor.

Todavía puede usar el operador de punto para modificar cualquiera de las variables, como barista.experience.

```
// Constructor con nombre de clase  
class BaristaWithConstructor {  
    Nombre de  
    cadena; experiencia interna;  
  
    // Constructor - Parámetros con nombre usando {}  
    BaristaWithConstructor({este.nombre, esta.experiencia});  
  
    // Método - valor de retorno int  
    whatIsTheExperience() { experiencia de  
        retorno;  
    }  
}
```

```
// Constructor con nombre de clase y valor de retorno  
BaristaWithConstructor barista = BaristaWithConstructor(nombre: 'Sandy', experiencia: 10); int experienciaPorPropiedad =  
barista.experiencia; int experienciaPorFuncion =  
barista.whatIsTheExperience(); print('experienciaPorPropiedad:  
$experienciaPorPropiedad'); print('experienciaPorFuncion:  
$experienciaPorFuncion'); // experiencia por propiedad: 10 // experiencia por  
función: 10
```

Los constructores con nombre le permiten implementar múltiples constructores para una clase y proporcionar intenciones claras de los datos inicializados.

Ahora veamos cómo crear una clase que defina un constructor con nombre que contenga el nombre y la experiencia del barista. En este ejemplo, se basará en el ejemplo anterior y agregará un segundo constructor, para ser precisos, un constructor con nombre. Lo declaras usando el nombre de la clase, el operador de punto y el nombre del constructor, como BaristaNamedConstructor.baristaDetails(name: 'Sandy', experience: 10), lo que te da un constructor con nombre usando parámetros con nombre. Todavía puede usar el operador de punto para modificar cualquiera de las variables, como barista.experience.

```
// Clase con constructor con nombre adicional class  
BaristaNamedConstructor {  
    Nombre de  
    cadena; experiencia interna;
```

```
// Constructor - Parámetros con nombre {}
BaristaNamedConstructor({este.nombre, esta.experiencia});

// Constructor con nombre - baristaDetails - Con parámetros con nombre
BaristaNamedConstructor.baristaDetails({este.nombre, esta.experiencia});
}

BaristaNamedConstructor barista = BaristaNamedConstructor.baristaDetails(nombre: 'Sandy',
experiencia: 10);
print('barista.nombre: ${barista.nombre} - barista.experiencia: ${barista.experiencia}'); // barista.nombre:
Sandy - barista.experiencia: 10
```

Herencia de clase

En programación, la herencia permite que los objetos comparten rasgos. Para heredar de otras clases, use la palabra clave `extends`. Utilice la palabra clave `super` para hacer referencia a la superclase (la clase principal). Los constructores no se heredan en la subclase.

En este ejemplo, tomará la clase `BaristaNamedConstructor` anterior y usará la herencia para crear una nueva clase que herede las características de la clase principal. Declare una nueva clase con el nombre `BaristaInheritance` usando la palabra clave `extends` y el nombre de la clase que está extendiendo, que aquí es `BaristaNamedConstructor`. El constructor de la clase heredada se ve un poco diferente a las declaraciones anteriores; al final del constructor, agrega dos puntos (`:`) y `super()`, en referencia a la superclase. Cuando se inicializa la clase `BaristaInheritance`, hereda las características de la clase principal, lo que significa que puede acceder a variables y métodos (funciones de clase) desde `BaristaNamedConstructor`.

```
// Clase de herencia
class BaristaInheritance extends BaristaNamedConstructor { int
yearsOnTheJob;

BaristaInheritance({this.yearsOnTheJob}) : super(); }

// Inicializar clase heredada
BaristaInheritance baristaInheritance = BaristaInheritance(yearsOnTheJob: 7); // Asignar variable
de clase principal baristaInheritance.name
= 'Sandy'; print('herenciabarista.yearsOnTheJob:
${herenciabarista.yearsOnTheJob}'); print('herenciabarista.nombre: ${herenciabarista.nombre}');
```

Mezclas de clase

Los mixins se usan para agregar funciones a una clase y le permiten reutilizar el código de la clase en diferentes clases. En otras palabras, los mixins le permiten acceder al código de clase entre clases no relacionadas. Para usar un mixin, agrega la palabra clave `with` seguida de uno o más nombres de mixin. Coloque la palabra clave `with` justo después de la declaración del nombre de la clase. La clase que implementa un mixin no declara un constructor. Por lo general, la clase mixin es una colección de métodos. En el Capítulo 7, "Agregar animación a una aplicación", creará dos aplicaciones de animación que usan mixins. Por ejemplo, el uso de `AnimationController` se basa en `TickerProviderStateMixin`.

En el siguiente ejemplo, la clase mixin BaristaMixinNoConstructor tiene un método llamado findBaristaFromLocation(String location) que devuelve una cadena. Este método llama al servicio localizarBarista() y devuelve el nombre del barista en una ubicación específica. Por lo general, tendría varios métodos en la clase que realizan una lógica de código diferente.

La clase BaristaWithMixin usa la clase mixin BaristaMixinNoConstructor a través de la palabra clave with . Las clases que usan un mixin pueden declarar constructores. En esta clase, tiene el método retrieveBaristaNameFromLocation() que llama al método de la clase mixin findBaristaFromLocation(this.location) para recuperar el nombre del barista de una ubicación.

Tenga en cuenta que llama a findBaristaFromLocation(this.location) sin especificar la clase a la que pertenece.

```
// Clase Mixin declarada sin constructor class BaristaMixinNoConstructor
{
    String findBaristaFromLocation(String ubicación) { // Llamar al servicio
        para encontrar al barista String baristaName =
            BaristaLocator().locateBarista(ubicación); return nombrebarista;
    }
}

// Clase usando una clase mixin
BaristaWithMixin with BaristaMixinNoConstructor {
    Ubicación de la cadena;

    // Constructor
    BaristaWithMixin({esta.ubicación});

    // El poder de mezclar tenemos acceso completo a BaristaNamedConstructor
    String recuperarBaristaNameFromLocation() {
        return findBaristaFromLocation(esta.ubicación);
    }
}

// Mixin
BaristaWithMixin baristaWithMixin = BaristaWithMixin(); String name =
baristaWithMixin.findBaristaFromLocation('Huston'); print('baristaWithMixin nombre: $nombre'); // baristaConMixin nombre: Sandy
```

IMPLEMENTACIÓN DE LA PROGRAMACIÓN ASINCRÓNICA

En una aplicación móvil, utilizará mucha programación asíncrona o asíncrona . Las funciones asíncronas realizan operaciones que consumen mucho tiempo sin esperar a que se complete la operación. En Dart, para no bloquear la interfaz de usuario, utiliza funciones que devuelven objetos Future o Stream .

Un objeto Future representa un valor que estará disponible en algún momento en el futuro. Por ejemplo, llamar a un servicio web para recuperar valores puede ser rápido o llevar mucho tiempo, y no desea que el usuario deje de usar la aplicación mientras se ejecuta el proceso. Al usar un objeto Future , a medida que la función recupera valores, devuelve el control a la interfaz de usuario y el usuario continúa usando la aplicación. Una vez que los valores

se recuperan, actualizará la interfaz de usuario con los nuevos datos. En el Capítulo 13, "Persistencia local: guardar datos", verá cómo usar objetos Stream , que permiten agregar o devolver datos en el futuro. Para lograr esto, Dart usa StreamController y Stream.

Las palabras clave `async` y `await` se usan juntas. Marque la función como asíncrona y coloque la palabra clave `await` antes de la función que devolverá datos en el futuro. Tenga en cuenta que las funciones marcadas como asíncronas deben tener un tipo de retorno assignable a `Futuro`.

En este ejemplo, la función `totalCookiesCount()` implementa un objeto `Futuro` que devuelve un valor `int` . Para implementar un objeto `Futuro` , inicia la función con `Future<int>` (int o cualquier tipo de datos válido), el nombre de la función y la palabra clave `async` . El código dentro de la función que devuelve un valor futuro está marcado con la palabra clave `await` . El método `lookupTotalCookiesCountDatabase()` representa una llamada a un servidor web para recuperar datos y está precedido por la palabra clave `await` . La palabra clave `await` permite que se realice la solicitud y, en lugar de esperar a que regresen los datos, continúa ejecutando el siguiente bloque de código. Una vez que se recuperan los datos, el código continúa para finalizar la función y devuelve el valor.

```
// Función asíncrona y espera con futuro: valor de retorno de un entero
Future<int> totalCookiesCount() asíncrono { int
    cookiesCount = esperar buscarTotalCookiesCountDatabase(); // Devuelve 33 return cookiesCount;

}

// Método asíncrono para llamar al servidor web
Future<int> lookupTotalCookiesCountDatabase() asíncrono {
    // En una aplicación del mundo real llamamos al servidor web para recuperar datos en vivo
    return 33;
}

// El usuario presionó el botón

totalCookiesCount() .then((count) { print("cookiesCount:
    ${count}"); });
print("Esto se imprimirá antes de cookiesCount");
// Esto se imprimirá antes de cookiesCount //
cookiesCount: 33
```

RESUMEN

Este capítulo cubrió los conceptos básicos del lenguaje Dart. Aprendió cómo comentar su código para una mejor legibilidad y cómo usar la función `main()` que inicia la aplicación. Aprendió a declarar variables para hacer referencia a diferentes tipos de valores, como números, cadenas, booleanos, listas y más. Usó la Lista para almacenar una matriz de filtros y aprendió a iterar a través de cada valor. Observó los símbolos de operadores comúnmente utilizados para realizar notaciones aritméticas, de igualdad, lógicas, condicionales y en cascada. La notación en cascada es un operador poderoso para realizar múltiples llamadas de secuencia en el mismo objeto, como escalar y traducir (posicionar) un objeto.

Para usar paquetes, bibliotecas y clases externos, usó la declaración de importación . Observó un ejemplo de cómo usar la programación asíncrona con un objeto Future . El ejemplo llamó a un servidor web simulado para buscar un recuento total de cookies en segundo plano mientras el usuario continuaba usando la aplicación sin interrupciones.

Finalmente, aprendió a usar clases para agrupar lógica de código que usaba variables para contener datos como el nombre y la experiencia de un barista. Las clases también definen funciones que ejecutan la lógica del código, como buscar la experiencia del barista.

En el próximo capítulo, creará una aplicación de inicio. Esta aplicación de inicio es la plantilla básica utilizada para iniciar cada nuevo proyecto.

LO QUE APRENDISTE EN ESTE CAPÍTULO

TEMA	CONCEPTOS CLAVE
Comentando el código	Hay comentarios de una sola línea, de varias líneas y de documentación.
Accediendo a main()	main() es la función de nivel superior.
Usando variables	Puede almacenar valores como números, cadenas, booleanos, listas, mapas y runas.
Usando operadores	Un operador es un símbolo que se utiliza para realizar notaciones aritméticas, de igualdad, relacionales, de prueba de tipos, de asignación, lógicas, condicionales y en cascada.
Uso de sentencias de flujo	Las instrucciones de flujo incluyen if y else, el operador ternario, bucles for , while y do-while, while y break, continue y switch y case.
Uso de funciones	Las funciones se utilizan para agrupar la lógica reutilizable.
Importación de paquetes	Puede usar la declaración de importación para importar paquetes, bibliotecas o clases externas.
usando clases	Puede crear clases para separar la lógica del código.
Implementando la programación asíncrona	Puede utilizar la programación asíncrona para no bloquear la interfaz de usuario.

4

Crear un iniciador Plantilla de proyecto

LO QUE APRENDERÁS EN ESTE CAPÍTULO

Cómo crear carpetas en el proyecto, agrupándolas por tipos de archivos

Cómo separar y estructurar widgets en diferentes archivos

En este capítulo, aprenderá cómo crear un proyecto inicial de Flutter y cómo estructurar los widgets. (Cubriré los widgets en profundidad en los próximos tres capítulos). En capítulos futuros, cada vez que inicie un nuevo ejemplo, me referiré a este capítulo, que contiene los pasos para crear un nuevo proyecto de inicio. Al igual que cuando se construye una casa, los cimientos son el factor más crítico, y lo mismo ocurre cuando se crean nuevas aplicaciones.

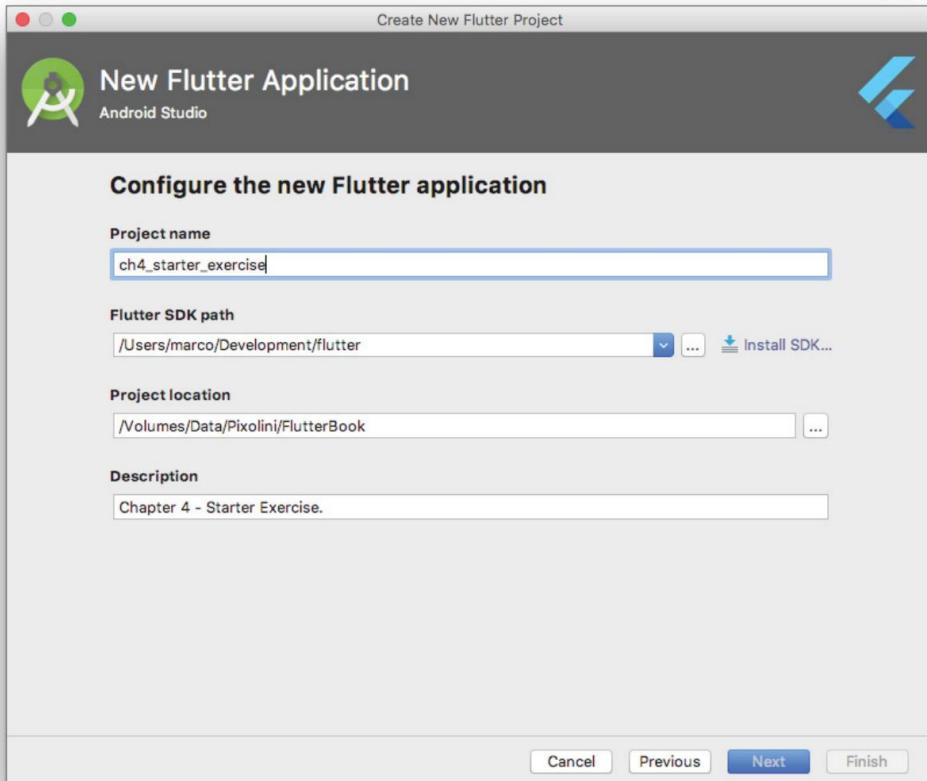
CREAR Y ORGANIZAR CARPETAS Y ARCHIVOS

Todas las aplicaciones de ejemplo creadas en este libro comienzan con los mismos pasos de este capítulo para crear un proyecto inicial, por lo que me referiré a este proceso con frecuencia. Para mantener el código del proyecto organizado, creará diferentes carpetas y archivos para agrupar lógica similar y estructurar los widgets.

PRUÉBALO Creación de la estructura de carpetas

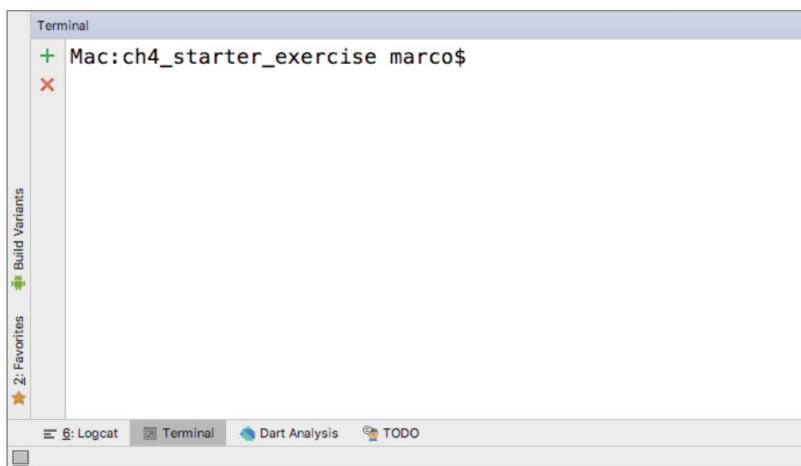
Cree un nuevo proyecto de Flutter siguiendo los pasos del Capítulo 2, "Creación de una aplicación Hello World".

1. En el paso 4 de la sección "Crear una nueva aplicación", ingrese ch4_starter_exercise para el proyecto nombre y haga clic en Siguiente. Esta aplicación es el ejercicio de muestra para estructurar proyectos futuros. Tenga en cuenta que la ruta del SDK de Flutter es la carpeta de instalación que eligió en el Capítulo 1. Opcionalmente, puede cambiar la ubicación y la descripción del proyecto.



Es hora de crear la estructura de carpetas para mantenerlo organizado. Esta estructura es mi preferencia personal y, según la complejidad del proyecto, es posible que necesite más o menos carpetas. Como mínimo, para cada nuevo proyecto, cree la carpeta de páginas . Contiene todas las páginas nuevas creadas para la aplicación, manteniéndolas separadas para mantenerlas.

2. Haga clic en el botón Terminal en la parte inferior de la ventana de Android Studio.

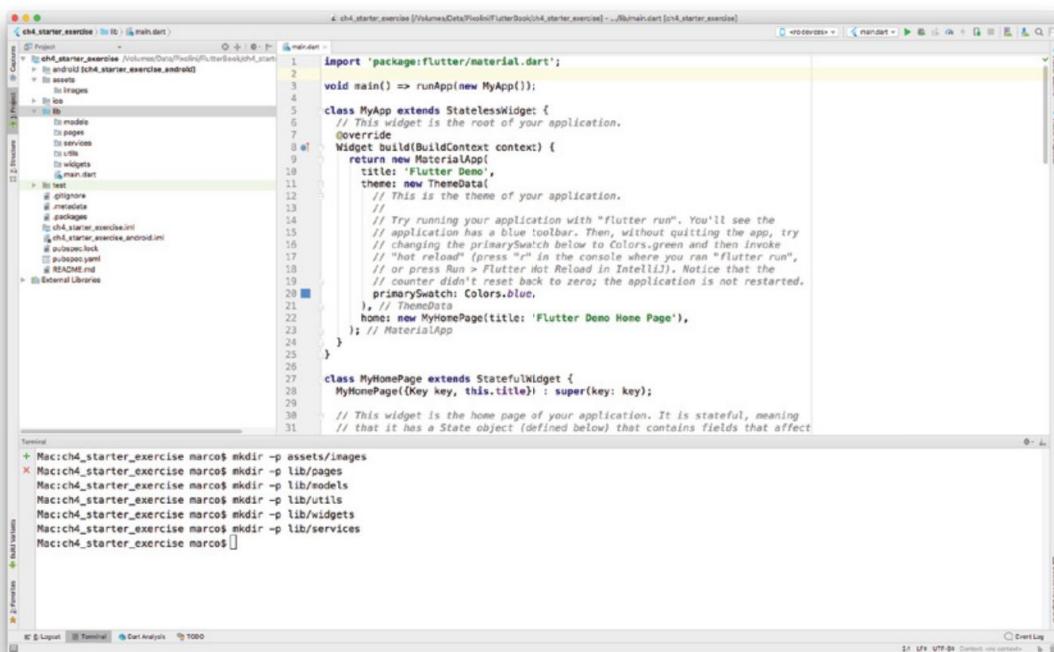


3. Para crear las estructuras de carpetas, ejecute el comando `mkdir -p carpeta/subcarpeta`. Este comando `mkdir` crea una carpeta y el parámetro `-p` crea una carpeta y una subcarpeta en una ejecución. El último parámetro que pasa es la estructura de carpetas/subcarpetas .
4. Ejecute cada comando `mkdir` en la ventana Terminal para crear cada estructura de carpetas. Por ejemplo, ejecute el comando `mkdir -p assets/images` para crear las carpetas de `assets/images` . Repita el comando `mkdir` para cada estructura de carpetas enumerada aquí. Para su comodidad, he enumerado los comandos tanto para Mac como para Windows.

```
// Desde la Terminal ingrese los siguientes comandos
Mac:starter_exercise marco$ mkdir -p recursos/ímágenes
Mac:ejercicio de inicio marco$ mkdir -p lib/pages
Mac:ejercicio de inicio marco$ mkdir -p lib/modelos
Mac:ejercicio de inicio marco$ mkdir -p lib/utils
Mac:ejercicio de inicio marco$ mkdir -p lib/widgets
Mac:ejercicio de inicio marco$ mkdir -p lib/servicios

// Desde el símbolo del sistema de Windows, ingrese los siguientes comandos
F:\Pixelini\Flutter\starter_exercise>mkdir activos\ímágenes
F:\Pixelini\Flutter\starter_exercise>mkdir lib\pages
F:\Pixelini\Flutter\starter_exercise>mkdir lib\modelos
F:\Pixelini\Flutter\starter_exercise>mkdir lib\utils
F:\Pixelini\Flutter\starter_exercise>mkdir lib\widgets
F:\Pixelini\Flutter\starter_exercise>mkdir lib\servicios
```

Echa un vistazo a las nuevas estructuras de carpetas. No todos los proyectos los usarán todos, pero es una excelente manera de mantenerse organizado. Las carpetas `assets` y `lib` se encuentran en la carpeta raíz del proyecto. La carpeta de activos contiene elementos como imágenes, archivos de datos y fuentes, y la carpeta `lib` contiene toda la lógica del código fuente, incluida la interfaz de usuario.



activos/ímágenes: la carpeta de activos contiene subcarpetas como imágenes, fuentes y archivos de configuración.

lib/pages: la carpeta de páginas contiene archivos de interfaz de usuario (UI) como inicios de sesión, listas de elementos, gráficos, y ajustes.

lib/models: la carpeta de modelos contiene clases para sus datos, como información del cliente y artículos de inventario.

lib/utils: la carpeta utils contiene clases auxiliares, como cálculos de fechas y conversión de datos.

lib/widgets: la carpeta de widgets contiene diferentes archivos Dart que separan los widgets para reutilizarlos a través de la aplicación.

lib/services: la carpeta de servicios contiene clases que ayudan a recuperar datos de servicios a través de Internet. Un gran ejemplo es cuando se usa Google Cloud Firestore, Cloud Storage, Realtime Data base, Authentication o Cloud Functions. Puede recuperar datos de cuentas de redes sociales, servidores de bases de datos, etc. En los Capítulos 14, 15 y 16, aprenderá cómo usar la administración de estado para autenticar usuarios, recuperar y sincronizar registros de bases de datos desde la nube usando Cloud Firestore.

Cómo funciona

Usando la terminal de Mac o el símbolo del sistema de Windows, ejecute el comando `mkdir` con el parámetro de nombre de carpeta. El comando `mkdir` crea la estructura de carpetas en la ubicación especificada.

WIDGETS DE ESTRUCTURACIÓN

Antes de comenzar a desarrollar una aplicación, es importante crear su estructura; como cuando se construye una casa, primero se crean los cimientos. (En el Capítulo 5, "Comprendión del árbol de widgets", explorará los widgets con más detalle). Estructurar los widgets de manera organizada mejora la legibilidad y el mantenimiento del código. Al crear un nuevo proyecto de Flutter, el kit de desarrollo de software (SDK) no crea automáticamente el archivo `home.dart` separado , que contiene la página de presentación principal cuando se inicia la aplicación. Por lo tanto, para tener separación de código, debe crear manualmente la carpeta de páginas y el archivo `home.dart` dentro de ella. El archivo `main.dart` contiene la función `main()` que inicia la aplicación y llama al widget de Inicio en el archivo `home.dart` .

PRUÉBELO Creación de archivos Dart y widgets

Una excelente manera de aprender cómo funciona Flutter es comenzar desde una pizarra en blanco. Elimina todo el contenido del `main.dart` archivo de dardos El archivo `main.dart` tiene tres secciones principales.

El paquete/archivo de importación

La función principal()

Una clase que extiende un widget `StatelessWidget` y devuelve la aplicación como un widget (como dije antes, casi todo es un widget)

Tenga en cuenta para el paquete de importación que utilizará Material Design de Google. Todos los ejemplos en el libro importan y usan Material Design. En el Capítulo 2, aprendió que Material Design es un sistema de pautas de mejores prácticas para el diseño de interfaces de usuario. Los componentes de Material Design en un proyecto de Flutter son widgets visuales, de comportamiento y de movimiento . Cupertino también se puede usar para adherirse al lenguaje de diseño de iOS de Apple que admite widgets de estilo iOS. Puede usar ambos estándares en diferentes partes de su aplicación. Desde el principio, Flutter es lo suficientemente inteligente como para mostrar las acciones nativas en ambos sistemas operativos sin que tengas que preocuparte por eso.

Por ejemplo, al importar la biblioteca `cupertino.dart` , puede mezclar algunos de los widgets de Cupertino con Material Design. El selector de fecha y hora funciona de manera diferente en Android e iOS, y puede especificar en el código qué widget mostrar según el sistema operativo. Sin embargo, deberá elegir por adelantado Material Design o Cupertino para la apariencia completa de la aplicación. ¿Por qué? Bueno, la base de su aplicación debe ser un widget de `MaterialApp` o un widget de `CupertinoApp` porque esto determina la disponibilidad de los widgets. En el paso 3 de este ejercicio, aprenderá a usar el widget `MaterialApp` .

Comencemos agregando el código al archivo `main.dart` y guardándolo.

1. Importe el paquete/archivo. La importación predeterminada es la biblioteca `material.dart` para permitir el uso de Diseño de materiales. (Para usar los widgets estilo Cupertino iOS, importa la biblioteca `cupertino.dart` en lugar de `material.dart` . Para las aplicaciones de este libro, usaré Material Design). Luego, importe la página `home.dart` ubicada en la carpeta de páginas .

```
importar 'paquete: flutter/material.dart'; import  
'paquete:ch4_starter_exercise/pages/home.dart';
```

2. Después de las dos instrucciones de importación , deje una línea en blanco e ingrese la función main() que se muestra a continuación. La función main() es el punto de entrada a la aplicación y llama a la clase MyApp .

```
void main() => runApp(MiAplicación());
```

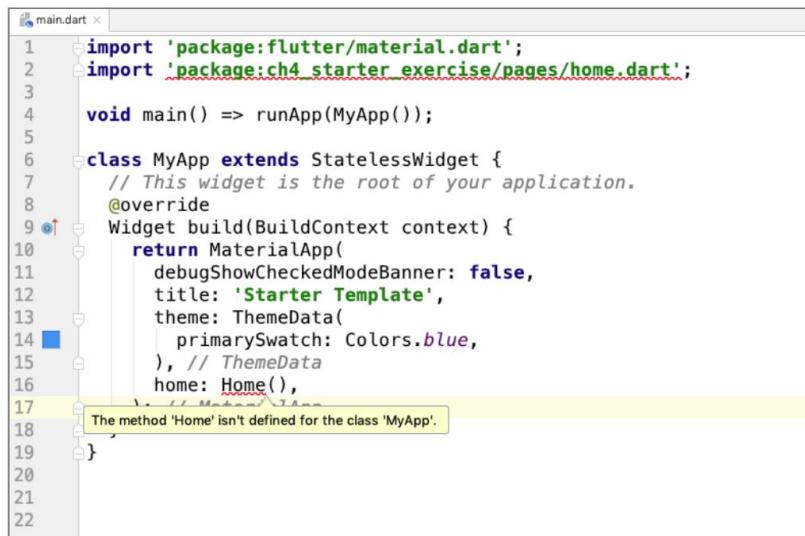
3. Escriba la clase MyApp que amplía StatelessWidget.

La clase MyApp devuelve un widget de MaterialApp que declara el título, el tema y las propiedades de inicio . Hay muchas otras propiedades de MaterialApp disponibles. Observe que la propiedad home llama a la clase Home() , que se crea más adelante en el archivo home.dart . class MyApp extiende

```
Widget { // Este widget es la raíz de su  
aplicación. @anular
```

```
Compilación del widget (contexto BuildContext) {  
    return  
        MaterialApp( debugShowCheckedModeBanner:  
        false, title: 'Starter Template', theme:  
  
        ThemeData( PrimarySwatch:  
  
            Colors.blue, ),  
        home: Home(), );  
  
}
```

Android Studio muestra una línea ondulada roja debajo de la declaración de importación pages/home.dart , así como el método Home() , que tiene este error: El método "Home" no está definido para la clase 'MyApp'. Al pasar el mouse sobre pages/home.dart y Home() , puede leer cada error.



Esto es normal ya que no ha creado el archivo home.dart que contiene la página de inicio. Este nombre puede ser el que quieras, pero siempre es bueno tener un nombre descriptivo para cada página.

4. Cree un nuevo archivo Dart en la carpeta de páginas . Haga clic derecho en la carpeta de páginas , seleccione Nuevo → Archivo Dart, ingrese home.dart y haga clic en el botón Aceptar para guardar.

5. Al igual que en el paso 1, importe el paquete/archivo material.dart . Como recordatorio, usará Material Design para todas las aplicaciones de ejemplo.

```
import 'paquete:flutter/material.dart';
```

6. Comience a escribir st y, wow, se abre la ayuda de autocompletado. A medida que escribe la abreviatura de un StatelessWidget , las plantillas en vivo de Android Studio completan automáticamente la estructura básica del widget de Flutter. Seleccione la abreviatura stful .



7. Ahora todo lo que necesita hacer es darle a la clase StatefulWidget su nombre: Inicio. Ya que es una clase, el La convención de nomenclatura es comenzar la palabra con una letra mayúscula.

```
// home.dart
import 'paquete:flutter/material.dart';

class Home extiende StatefulWidget {
    @anular
    _HomeState createState() => _HomeState();
}

class _HomeState extiende State<Home> {
    @anular
    Compilación del widget (contexto BuildContext) {
        devolver Contenedor();
    }
}
```

Está utilizando StatefulWidget para la clase Inicio porque en una aplicación del mundo real lo más probable es que se mantenga un estado para los datos. Un ejemplo de cuándo necesitaría un estado es un PopupMenuItem

widget en el widget AppBar que muestra una fecha seleccionada utilizada por varias páginas. Si la clase Inicio no necesita mantener el estado, utilice

StatelessWidget. class Home extiende
StatelessWidget { @override

```
    Comilación del widget (contexto BuildContext) {  
        devolver Contenedor();
```

```
}
```

8. Reemplace el widget Container() con un widget Scaffold . El widget Scaffold implementa el diseño visual básico de Material Design, lo que permite la simple adición de AppBar, BottomAppBar, FloatingActionButton, Drawer, Snackbar, BottomSheet y más. (Si se tratara de una CupertinoApp, podría usar CupertinoPageScaffold o

CupertinoTabScaffold). class _HomeState
extends State<Home> { @override

```
    Creación de widgets (contexto BuildContext) { return  
        Scaffold (
```

```
            appBar: AppBar(título:  
                Texto('Inicio'), ), cuerpo:
```

```
            Contenedor(), );
```

```
}
```

El siguiente es el código fuente completo de los archivos main.dart y home.dart :

```
// main.dart import  
'paquete:flutter/material.dart'; import  
'paquete:ch4_starter_exercise/pages/home.dart';  
  
void main() => runApp(MiAplicación());  
  
class MyApp extiende StatelessWidget { // Este widget  
    es la raíz de su aplicación.  
    @anular  
    Comilación del widget (contexto BuildContext) {  
        return  
            MaterialApp( debugShowCheckedModeBanner:  
                false, title: 'Starter Template', theme:  
                ThemeData( PrimarySwatch:  
                    Colors.blue, ), home: Home(), );  
  
    }  
}  
  
// home.dart  
import 'paquete:flutter/material.dart';  
  
class Home extiende StatefulWidget {  
    @anular
```

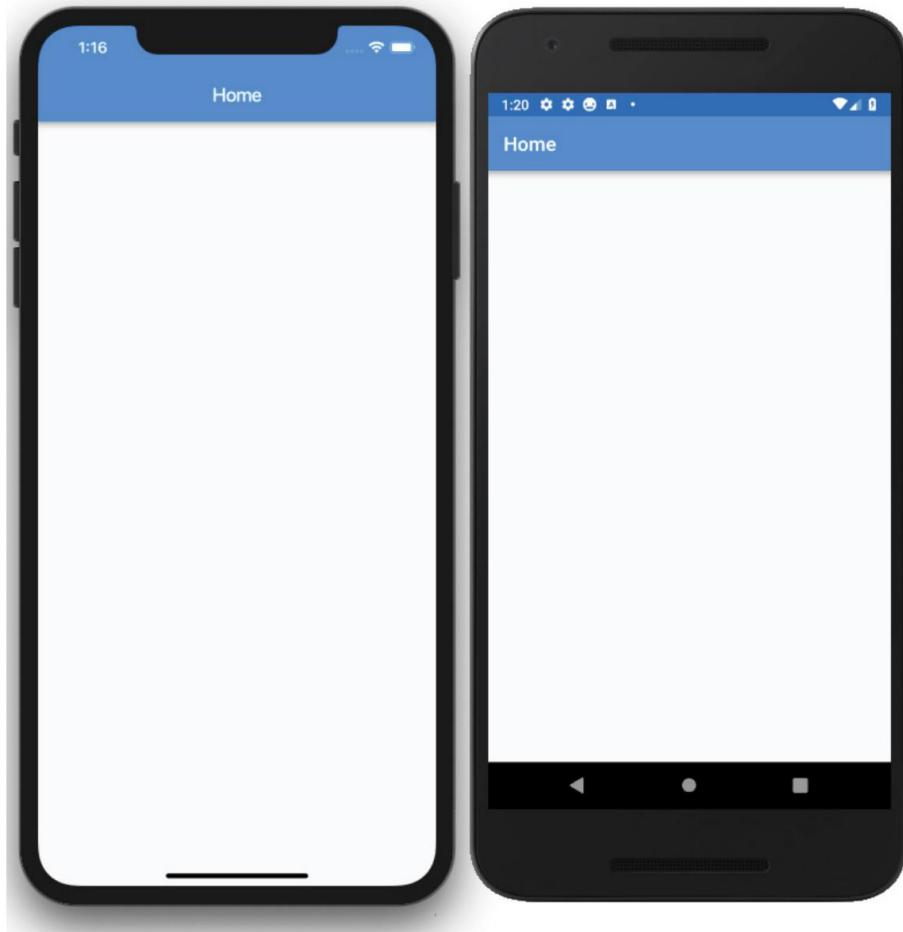
```
_HomeState createState() => _HomeState(); }

clase _HomeState extiende State<Home> { @override

    Creación de widgets (contexto BuildContext)
    { return Scaffold (
        appBar: AppBar(título:
            Texto('Inicio'), ), cuerpo:
        Contenedor(), );

    }
}
```

Continúe y ejecute el proyecto y vea cómo se ve su aplicación.



Tenga en cuenta que agregué lo siguiente a Scaffold y AppBar: Contenedor (puede ser un TabController, PageController, etc.) y FloatingActionButton.

Cómo funciona

Para mantener su código legible y mantenible, estructura los widgets apropiados en sus propias clases y archivos Dart. Usted estructura sus proyectos iniciales con el archivo main.dart que contiene la función main() que inicia la aplicación. La función main() llama al widget Inicio en el archivo home.dart . El widget Inicio es la página de presentación principal que se muestra cuando se inicia la aplicación. Por ejemplo, el widget Inicio puede contener un widget TabBar o BottomNavigationBar .

RESUMEN

En este capítulo, aprendió a crear el proyecto inicial que usará para todas las aplicaciones futuras de este libro. Creó carpetas con el comando mkdir y las nombró en consecuencia para agrupar la lógica. También creó dos archivos Dart: main.dart para la función main() que inicia la aplicación y home.dart para contener el código del widget Inicio .

En el próximo capítulo, analizamos el árbol de widgets. Flutter funciona anidando widgets, y descubrimos rápidamente que la legibilidad y la capacidad de mantenimiento se ven afectadas rápidamente. Echamos un vistazo a un ejemplo de cómo aplanar el árbol de widgets.

LO QUE APRENDISTE EN ESTE CAPÍTULO

TEMA	CONCEPTOS CLAVE
mkdir	Este es el comando para crear carpetas por nombre.
dardo principal	La función main() inicia la aplicación y devuelve MaterialApp (Android) o CupertinoApp (iOS).
casa.dardo	Contiene widgets que muestran el diseño de la primera página o página de inicio.

5

Comprender el árbol de widgets

LO QUE APRENDERÁS EN ESTE CAPÍTULO

Los fundamentos de los widgets

Cómo utilizar un árbol de widgets completo

Cómo usar un árbol de widgets poco profundo

El árbol de widgets es cómo creas tu interfaz de usuario; usted coloca los widgets unos dentro de otros para crear diseños simples y complejos. Dado que casi todo en el marco de Flutter es un widget, y a medida que comienza a anidarlos, el código puede volverse más difícil de seguir. Una buena práctica es tratar de mantener el árbol de widgets lo más superficial posible. Para comprender los efectos completos de un árbol profundo, observará un árbol de widgets completo y luego lo refactorizará en un árbol de widgets poco profundo, lo que hará que el código sea más manejable. Aprenderá tres formas de crear un árbol de widgets poco profundo mediante la refactorización: con una constante, con un método y con una clase de widget.

INTRODUCCIÓN A LOS WIDGETS

Antes de analizar el árbol de widgets, veamos la breve lista de widgets que utilizará para las aplicaciones de ejemplo de este capítulo. En este punto, no se preocupe por comprender la funcionalidad de cada widget; solo concéntrese en lo que sucede cuando anida widgets y cómo puede separarlos en secciones más pequeñas. En el Capítulo 6, "Uso de widgets comunes", profundizará en el uso de los widgets más comunes por funcionalidad.

Como mencioné en el Capítulo 4, "Creación de una plantilla de proyecto inicial", este libro utiliza Material Design para todos los ejemplos. Los siguientes son los widgets (utilizables solo con Material Design) que usará para crear los proyectos de árbol de widgets completos y superficiales para este capítulo:

Scaffold: implementa el diseño visual de Material Design, lo que permite el uso de Flutter
Widgets de componentes de materiales

AppBar: implementa la barra de herramientas en la parte superior de la pantalla

CircleAvatar: generalmente se usa para mostrar una foto de perfil de usuario redondeada, pero puede usarla para cualquier imagen

Divisor: dibuja una línea horizontal con relleno arriba y abajo

Si la aplicación que está creando usa Cupertino, puede usar los siguientes widgets en su lugar. Tenga en cuenta que con Cupertino puede usar dos andamios diferentes, un andamio de página o un andamio de pestañas.

CupertinoPageScaffold: implementa el diseño visual de iOS para una página. Funciona con CupertinoNavigationBar para proporcionar el uso de los widgets estilo iOS de Cupertino de Flutter.

CupertinoTabScaffold: implementa el diseño visual de iOS. Esto se utiliza para navegar entre varias páginas, con las pestañas en la parte inferior de la pantalla que le permiten usar los widgets estilo iOS de Cupertino de Flutter.

CupertinoNavigationBar: implementa la barra de herramientas de diseño visual de iOS en la parte superior de la pantalla.

La Tabla 5.1 resume una breve lista de los diferentes widgets para usar según la plataforma.

TABLA 5.1: Material Design vs. Cupertino Widgets

DISEÑO DE MATERIALES	CUPERTINO
Andamio	CupertinoPáginaAndamio CupertinoTabAndamio
barra de aplicaciones	CupertinoNavegaciónBar
CírculoAvatar	n / A
Divisor	n / A

Los siguientes widgets se pueden usar tanto con Material Design como con Cupertino:

SingleChildScrollView: agrega la capacidad de desplazamiento vertical u horizontal a un solo widget de niño.

Relleno: esto agrega relleno izquierdo, superior, derecho e inferior.

Columna: muestra una lista vertical de widgets secundarios.

Fila: muestra una lista horizontal de widgets secundarios.

Contenedor: este widget se puede usar como un marcador de posición vacío (invisible) o puede especificar altura, ancho, color, transformación (girar, mover, sesgar) y muchas más propiedades.

Expandido: esto expande y llena el espacio disponible para el widget secundario que pertenece a un Widget de columna o fila .

Texto: el widget Texto es una excelente manera de mostrar etiquetas en la pantalla. Se puede configurar para que sea de una sola línea o de varias líneas. Se puede aplicar un argumento de estilo opcional para cambiar el color, la fuente, el tamaño y muchas otras propiedades.

Stack—¡Qué widget tan poderoso! Stack le permite apilar widgets uno encima del otro y usar un widget posicionado (opcional) para alinear cada elemento secundario de Stack para el diseño necesario. Un gran ejemplo es un ícono de carrito de compras con un pequeño círculo rojo en la parte superior derecha para mostrar la cantidad de artículos a comprar.

Posicionado: el widget Posicionado funciona con el widget Apilado para controlar el posicionamiento y el tamaño de los elementos secundarios. Un widget posicionado le permite establecer la altura y el ancho. También puede especificar la distancia de ubicación de la posición desde los lados superior, inferior, izquierdo y derecho del widget de pila .

Ha aprendido acerca de cada widget que implementará en el resto de este capítulo. Ahora creará un árbol de widgets completo y luego aprenderá cómo refactorizarlo en un árbol de widgets poco profundo.

CONSTRUYENDO EL ÁRBOL COMPLETO DE WIDGET

Para mostrar cómo un árbol de widgets puede comenzar a expandirse rápidamente, utilizará una combinación de widgets Columna, Fila, Contenedor, CircleAvatar, Divisor, Relleno y Texto . Echará un vistazo más de cerca a estos widgets en el Capítulo 6. El código que escribirá es un ejemplo simple y podrá ver de inmediato cómo el árbol de widgets puede crecer rápidamente (Figura 5.1).

```

44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91

```

```

Row(
  mainAxisAlignment: MainAxisAlignment.start,
  mainAxisSize: MainAxisSize.max,
  children: <Widget>[
    Column(
      children: <Widget>[
        Container(
          color: Colors.yellow,
          height: 60.0,
          width: 60.0,
        ), // Container
        Padding(padding: EdgeInsets.all(16.0)),
        Container(
          color: Colors.amber,
          height: 40.0,
          width: 40.0,
        ), // Container
        Padding(padding: EdgeInsets.all(16.0)),
        Container(
          color: Colors.brown,
          height: 20.0,
          width: 20.0,
        ), // Container
        Divider(),
        Row(
          mainAxisAlignment: MainAxisAlignment.start,
          mainAxisSize: MainAxisSize.max,
          children: <Widget>[
            CircleAvatar(
              backgroundColor: Colors.lightGreen,
              radius: 100.0,
              child: Stack(
                children: <Widget>[
                  Container(
                    height: 100.0,
                    width: 100.0,
                    color: Colors.yellow,
                  ), // Container
                  Container(
                    height: 60.0,
                    width: 60.0,
                    color: Colors.amber,
                  ), // Container
                  Container(
                    height: 40.0,
                    width: 40.0,
                    color: Colors.brown,
                  ), // Container
                ],
              ),
            ),
          ],
        ),
      ],
    ),
  ],
),

```

FIGURA 5.1: Vista completa del árbol de widgets

PRUÉBALO Creación del árbol de widgets completo

Cree un nuevo proyecto Flutter llamado ch5_widget_tree. Puede seguir las instrucciones del Capítulo 4.

Para este proyecto, solo necesita crear la carpeta de páginas . Puede ver el árbol completo de widgets al final de los pasos.

1. Abra el archivo home.dart .

2. Agregue a la propiedad del cuerpo de Scaffold un widget SafeArea con la propiedad secundaria establecida en un Vista de desplazamiento de un solo niño. Agregue un widget de relleno como elemento secundario de SingleChildScrollView. Establezca la propiedad de relleno en EdgeInsets.all(16.0).

```
cuerpo: SafeArea(  
    hijo: SingleChildScrollView(  
        niño: relleno (  
            relleno: EdgeInsets.all(16.0), ), ), ),
```

3. Agregue a la propiedad secundaria de relleno un widget de columna con la propiedad secundaria establecida en una fila.

```
cuerpo: SafeArea(  
    hijo: SingleChildScrollView(  
        niño: relleno (  
            relleno: EdgeInsets.all(16.0), child:  
                Column( children:  
                    <Widget>[ Row( children:  
  
                        <Widget>[ ], ), ], ), ), ), ),
```

4. Agregue a los widgets secundarios de Fila en este orden: Contenedor, Relleno, Expandido, Relleno,

Contenedor y Acolchado. No ha terminado de agregar widgets; en el siguiente paso, agregará un widget de fila con varios widgets anidados.

Row(children:

```
<Widget>[ Container( color:  
    Colors.yellow,  
    height: 40.0,  
  
    width: 40.0, ), Padding(padding: EdgeInsets.all(16.0),),  
  
    Expanded( child:  
        Container( color:  
            Colors.amber , altura: 40.0,
```

```
ancho: 40.0, ), ),
```

```
Padding(relleno: EdgeInsets.all(16.0),), Container( color:  
Colors.brown,  
height: 40.0, width:  
40.0, ), ], ),
```

5. Agregue un widget de relleno para crear un espacio antes del siguiente widget de fila .

```
Relleno(relleno: EdgeInsets.all(16.0),)
```

6. Agregue un widget de Fila con la propiedad de niños establecida en una Columna. Agregar a la Columna niños un Contenedor, Relleno, Contenedor , Relleno, Contenedor, Divisor, Fila, Divisor y Texto. Todavía no ha terminado de agregar widgets y, en el siguiente paso, agregará otro widget de fila con varios widgets anidados.

Row(children:

```
<Widget>[ Column( crossAxisAlignment: CrossAxisAlignment.start,  
mainAxisSize: MainAxisSize.max, children:  
<Widget>[ Container( color:  
Colors.yellow,  
height: 60.0, width: 60.0, ),  
Padding(padding:
```

```
EdgeInsets.all(16.0),), Container( color: Colors.amber,  
height: 40.0,  
width: 40.0, ),  
Padding(padding:
```

```
EdgeInsets.all(16.0),), Container( color: Colors.brown,  
height: 20.0,  
ancho: 20.0, ), Divider(),  
Row( children:  
<Widget>[ // En  
el  
siguiente  
paso  
agregaremos más  
widgets ], ), Divider(), Text('Final de la
```

```
línea'), ], ), ], ),
```

7. Modifique el último widget de Fila (del paso 6) y establezca la propiedad de niños en un CircleAvatar con un niño como Pila.

Agregue a la propiedad Stack children tres widgets de contenedor .

8. Despu s del widget Stack (del paso 7), agregue un widget Divider y luego un widget Text con una cadena de 'Fin de L nea'.

Divisor(),
Texto('Fin de la Línea'),

Ha agregado muchos widgets anidados para crear un diseño complejo. El código completo se muestra a continuación. En una aplicación del mundo real, esto es común. Inmediatamente comenzará a ver cómo el árbol de widgets puede crecer, haciendo que el código sea ilegible e inmanejable. Para mantener el ejemplo enfocado en qué tan rápido puede crecer el árbol de widgets, aquí usó widgets básicos. En una aplicación de nivel de producción, tendría incluso más widgets, como campos de texto, para permitir que el usuario ingrese texto.

```
importar 'paquete: flutter/material.dart';

class Home extiende StatefulWidget {
  @anular
  _HomeState createState() => _HomeState();
}

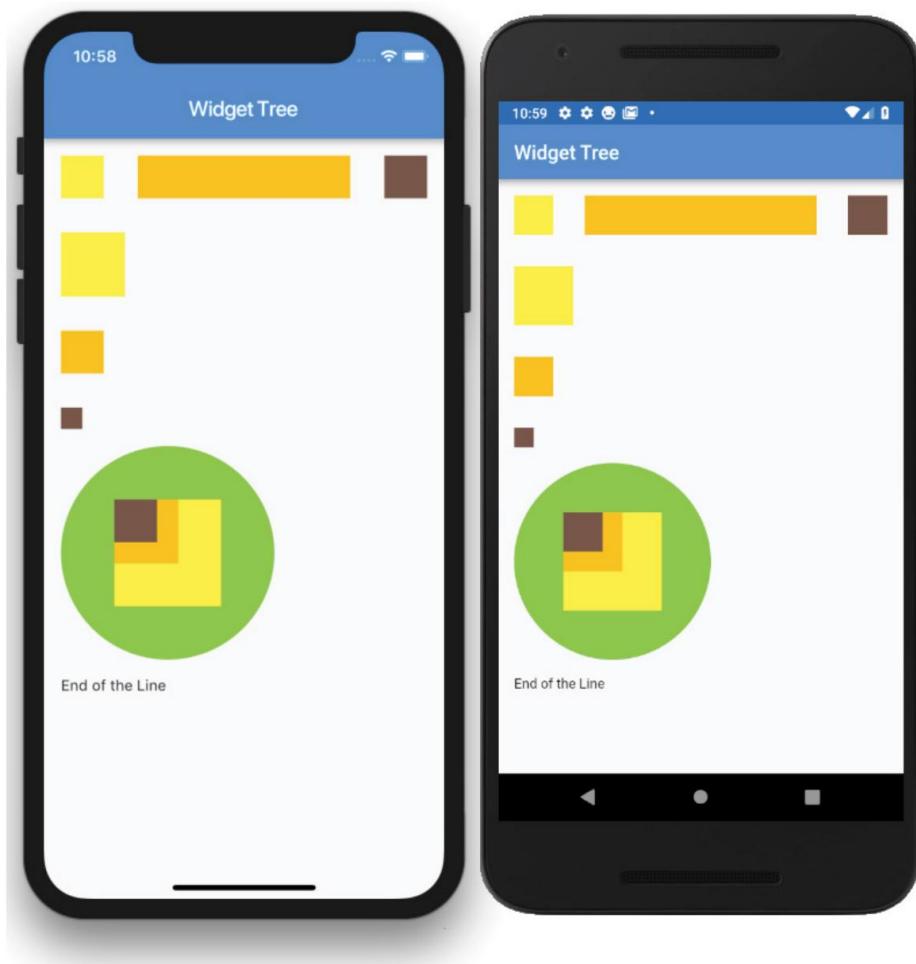
class _HomeState extiende State<Home> {
  @anular

  Compilación del widget (contexto BuildContext) {
```



```
Relleno(relleno: EdgeInsets.all(16.0),), Container( color:  
Colors.brown,  
height: 20.0, width: 20.0, ),  
Divider(),  
Row( children:  
  
<Widget>[ CircleAvatar(  
  
backgroundColor: Colors.lightGreen, radio:  
100.0, child:  
Stack( children:  
<Widget>[ Container( height:  
100.0, width:  
100.0, color:  
Colors.yellow, ),  
Container( height: 60.0,  
width: 60.0,  
color:  
Colors.amber, ),  
Container( altura: 40.0,  
ancho: 40.0,  
color:  
Colors.brown, ), ], ), ], ), );  
  
Divider(),  
Text('Fin de la  
  
línea'), ], ), ], ), ), ), ), );  
});
```

La siguiente imagen muestra el diseño de página resultante del árbol de widgets.



Cómo funciona

Para crear un diseño de página, anida widgets para crear una interfaz de usuario personalizada. El resultado de agregar widgets juntos se llama el árbol de widgets. A medida que aumenta la cantidad de widgets, el árbol de widgets comienza a expandirse rápidamente y hace que el código sea difícil de leer y administrar.

CONSTRUYENDO UN ÁRBOL DE ARTÍCULOS POCO PROFUNDO

Para que el código de ejemplo sea más legible y fácil de mantener, refactorizará las secciones principales del código en entidades separadas. Tiene varias opciones de refactorización y las técnicas más comunes son constantes, métodos y clases de widgets.

Refactorización con una constante

La refactorización con una constante inicializa el widget en una variable final . Este enfoque le permite separar los widgets en secciones, lo que mejora la legibilidad del código. Cuando los widgets se inicializan con una constante, se basan en el objeto BuildContext del widget principal.

¿Qué quiere decir esto? Cada vez que se vuelve a dibujar el widget principal, todas las constantes también volverán a dibujar sus widgets, por lo que no puede optimizar el rendimiento. En la siguiente sección, analizará en detalle la refactorización con un método en lugar de una constante. Los beneficios de hacer que el árbol de widgets sea más bajo son similares con ambas técnicas.

El siguiente código de ejemplo muestra cómo usar una constante para inicializar la variable contenedora como final con el widget Contenedor . Inserta la variable contenedora en el árbol de widgets donde sea necesario.

```
contenedor final = Contenedor( color:  
    Colores.amarillo, alto: 40.0,  
    ancho: 40.0, );
```

Refactorización con un método

La refactorización con un método devuelve el widget llamando al nombre del método. El método puede devolver un valor por un widget general (Widget) o un widget específico (Contenedor, Fila y otros).

Los widgets inicializados por un método se basan en el objeto BuildContext del widget principal. Podría haber efectos secundarios no deseados si este tipo de métodos están anidados y llaman a otros métodos/funciones anidados. Dado que cada situación es diferente, no asuma que usar métodos no es una buena opción. Este enfoque le permite separar los widgets en secciones, lo que mejora la legibilidad del código. Sin embargo, como cuando se refactoriza con una constante, cada vez que se vuelve a dibujar el widget principal, todos los métodos también volverán a dibujar sus widgets. Eso significa que el árbol de widgets no se puede optimizar para el rendimiento.

El siguiente código de ejemplo muestra cómo usar un método para devolver un widget de contenedor . Este primer método devuelve el widget de contenedor como un widget general y el segundo método devuelve el widget de contenedor como un widget de contenedor . Ambos enfoques son aceptables. Inserta el nombre del método _buildContainer() en el árbol de widgets donde sea necesario.

```
// Devolver por widget general Widget de  
nombre _buildContainer() { return  
    Container( color:  
        Colors.yellow, height: 40.0,  
        width: 40.0, );  
  
}  
  
// O Retornar por Widget específico como Contenedor en este caso Contenedor  
_buildContainer() { return  
    Contenedor( color:  
        Colors.yellow, height: 40.0,
```

```
    ancho: 40.0, );
```

```
}
```

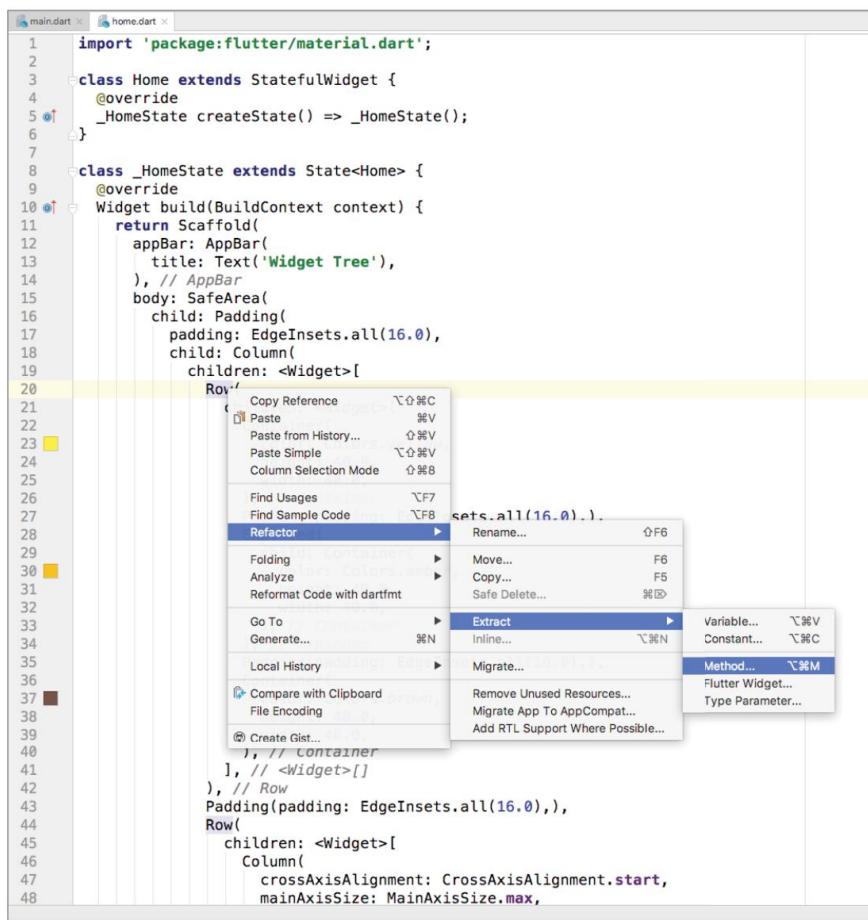
Veamos un ejemplo que refactoriza usando métodos. Este enfoque mejora la legibilidad del código al separar las partes principales del árbol de widgets en métodos separados. Se podría tomar el mismo enfoque refactorizando con una constante.

¿Cuál es el beneficio de usar el enfoque del método? El beneficio es la legibilidad del código pura y simple, pero pierde los beneficios de la reconstrucción del subárbol de Flutter: el rendimiento.

PRUÉBALO Refactorización con un método para crear un árbol de widgets poco profundo

Para refactorizar los widgets, use el patrón de método para aplanar el árbol de widgets.

1. Abra el archivo home.dart .
2. Coloque el cursor en el widget de la primera fila y haga clic con el botón derecho.
3. Seleccione Refactorizar → Extraer → Método y haga clic en Método.



4. En el cuadro de diálogo Extraer método, ingrese `_buildHorizontalRow` para el nombre del método. Observe la guión bajo antes de construir; esto le permite a Dart saber que es un método privado. Observe que todo el widget Fila y los elementos secundarios están resaltados para facilitar la visualización del código afectado.

```
15     body: SafeArea(
16       child: Padding(
17         padding: EdgeInsets.all(16.0),
18         child: Column(
19           children: <Widget>[
20             Row(
21               children: [
22                 Container(
23                   color: Colors.amber,
24                   height: 40.0,
25                   width: 40.0,
26                 ), // Container
27                 Padding(
28                   padding: EdgeInsets.all(16.0),
29                   child: Container(
30                     color: Colors.brown,
31                     height: 40.0,
32                     width: 40.0,
33                   ), // Container
34                 ), // Expanded
35                 Padding(padding: EdgeInsets.all(16.0)),
36                 Container(
37                   color: Colors.brown,
38                   height: 40.0,
39                   width: 40.0,
40                 ), // Container
41             ], // <Widget>[]
42           ],
43           padding: EdgeInsets.all(16.0),
44           Row(
45             children: <Widget>[
46               Column(
47                 mainAxisAlignment: MainAxisAlignment.start,
48                 mainAxisSize: MainAxisSize.max,
```

5. El widget Fila se reemplaza con el método `_buildHorizontalRow()`. Desplácese hasta la parte inferior de la código, y el método y los widgets están muy bien refactorizados.

```
Fila _construirFilaHorizontal()
{
  devolver Fila(
    children:

      <Widget>[ Container( color:
        Colors.yellow,
        height: 40.0, width: 40.0,
      ),
      Padding(padding: EdgeInsets.all(16.0),), Expanded( child:
        Container( color:
          Colors.amber, height: 40.0,
          width: 40.0, ),
      ),
      Relleno(relleno: EdgeInsets.all(16.0),), Contenedor( color:
        Colors.brown,
```

```
alto: 40.0, ancho:  
40.0, ], );  
  
}
```

6. Continúe y refactorice las otras filas y los widgets Fila y Pila .

El código fuente completo de home.dart se muestra a continuación. Observe cómo se aplana el árbol de widgets, lo que facilita la lectura. Decidir cuán superficial hacer el árbol de widgets depende de cada circunstancia y de su preferencia personal. Por ejemplo, digamos que está trabajando en su código y comienza a notar que se está desplazando mucho vertical u horizontalmente para hacer cambios. Esta es una buena indicación de que puede refactorizar partes del código en secciones separadas.

```
// home.dart import  
'paquete:flutter/material.dart';  
  
class Home extiende StatefulWidget {  
  @anular  
  _HomeState createState() => _HomeState();  
}  
  
class _HomeState extiende State<Home> {  
  @anular  
  Creación de widgets (contexto BuildContext) { return  
    Scaffold (  
      appBar: AppBar(título:  
        Texto('Árbol de widgets'), ), cuerpo:  
        SafeArea(  
          hijo: SingleChildScrollView(  
            niñ o: relleno (   
              padding: EdgeInsets.all(16.0), child:  
              Column( children:  
                <Widget>[ _buildHorizontalRow(),  
                  Padding(padding:  
                    EdgeInsets.all(16.0), ), _buildRowAndColumn(), ],  
                  ),  
                  ),  
                  ),  
                  ),  
                );  
  }  
  
  Row _buildHorizontalRow() { return  
    Row( children:  
      <Widget>[ Container( color:  
        Colors.yellow,  
        height: 40.0, width: 40.0, ),
```

```
Padding(padding: EdgeInsets.all(16.0),), Expanded( child:  
Container( color:  
    Colors.amber, height: 40.0,  
    width: 40.0, ), ),  
Padding(padding:  
  
    EdgeInsets.all(16.0),), Container( color: Colors.brown,  
    altura: 40.0,  
    ancho: 40.0, ], );  
  
}  
  
Row _buildRowAndColumn() { return  
    Row( children:  
  
        <Widget>[ Column( mainAxisAlignment: MainAxisAlignment.start,  
            mainAxisSize: MainAxisSize.max, children:  
                <Widget>[ Container( color:  
                    Colors.yellow,  
                    height: 60.0, width: 60.0, ),  
                    Relleno(relleno:  
  
                        EdgeInsets.all(16.0),), Contenedor( color: Colors.amber,  
                        altura: 40.0,  
                        ancho: 40.0, ),  
                        Relleno(relleno:  
  
                            EdgeInsets.all(16.0),), Contenedor( color: Colors.brown,  
                            alto: 20.0,  
                            ancho: 20.0, ), Divider(),  
  
                            _buildRowAndStack(),  
                            Divider(),  
                            Text('Final de la línea'), ], ), ], );  
  
}  
  
Row _buildRowAndStack() { return  
    Row( children:  
        <Widget>[
```

```
CírculoAvatar(  
    backgroundColor: Colors.lightGreen, radio: 100.0, child:  
    Stack( children:  
  
        <Widget>[ Container( height:  
            100.0, width:  
            100.0, color:  
            Colors.yellow, ),  
        Container( height: 60.0, width:  
  
            60.0, color:  
            Colors.amber, ),  
        Container( altura:  
            40.0, ancho: 40.0, color:  
  
            Colors.brown, ), ], ), ], );  
  
}  
}
```

Cómo funciona

La creación de un árbol de widgets poco profundo significa que cada widget se separa en su propio método por funcionalidad. Tenga en cuenta que la forma en que separa los widgets será diferente según la funcionalidad necesaria. Separar los widgets por método mejora la legibilidad del código, pero pierde los beneficios de rendimiento de la reconstrucción del subárbol de Flutter. Todos los widgets del método se basan en el BuildContext del parent, lo que significa que cada vez que se vuelve a dibujar el parent, también se vuelve a dibujar el método.

En este ejemplo, creó el método `_buildHorizontalRow()` para crear el widget de fila horizontal con widgets secundarios. El método `_buildRowAndColumn()` es un excelente ejemplo de aplazarlo aún más llamando al método `_buildRowAndStack()` para uno de los widgets secundarios de `Column`. La separación de `_buildRowAndStack()` se realiza para mantener plano el árbol de widgets porque el método `_buildRowAndStack()` crea un widget con varios widgets secundarios .

Refactorización con una clase de widget

La refactorización con una clase de widget le permite crear el widget creando una subclase de la clase `StatelessWidget` . Puede crear widgets reutilizables dentro del archivo Dart actual o separado e iniciarlos en cualquier parte de la aplicación. Tenga en cuenta que el constructor comienza con una palabra clave `const` , que le permite almacenar en caché y reutilizar el widget. Al llamar al constructor para iniciar el widget, use la `const`

palabra clave. Al llamar con la palabra clave const , el widget no se reconstruye cuando otros widgets cambian su estado en el árbol. Si omite la palabra clave const , se llamará al widget cada vez que se redibuje el widget principal.

La clase de widget se basa en su propio BuildContext, no en el padre como los enfoques de constante y método.

BuildContext es responsable de manejar la ubicación de un widget en el árbol de widgets.

En el Capítulo 7, "Agregar animación a una aplicación", creará un ejemplo que refactoriza y separa widgets con varios StatefulWidgets en lugar de la clase StatelessWidget .

¿Qué quiere decir esto? Cada vez que se vuelve a dibujar el widget principal, todas las clases de widget no se volverán a dibujar. Se construyen solo una vez, lo cual es excelente para optimizar el rendimiento.

El siguiente código de muestra muestra cómo usar una clase de widget para devolver un widget de contenedor . Inserta el widget const ContainerLeft() en el árbol de widgets donde sea necesario. Tenga en cuenta el uso de la palabra clave const para aprovechar el almacenamiento en caché.

```
clase ContainerLeft extiende StatelessWidget { const
  ContainerLeft({
    Clave
    clave, }) : super(clave: clave);

  @anular
  Widget build (contexto BuildContext) { return
    Container (color:
      Colors.yellow, height: 40.0,
      width: 40.0, );

  }

// Llame para inicializar el widget y anote la palabra clave const const
ContainerLeft(),
```

Veamos un ejemplo que refactoriza usando clases de widgets (un widget de Flutter). Este enfoque mejora la legibilidad y el rendimiento del código al separar las partes principales del árbol de widgets en clases de widgets separadas.

¿Cuál es el beneficio de usar las clases de widgets? Es rendimiento puro y simple durante las actualizaciones de pantalla. Al llamar a una clase de widget, debe usar la declaración const ; de lo contrario, se reconstruirá cada vez, sin almacenamiento en caché. Un ejemplo de refactorización con una clase de widget es cuando tiene un diseño de interfaz de usuario en el que solo los widgets específicos cambian de estado y otros permanecen igual.

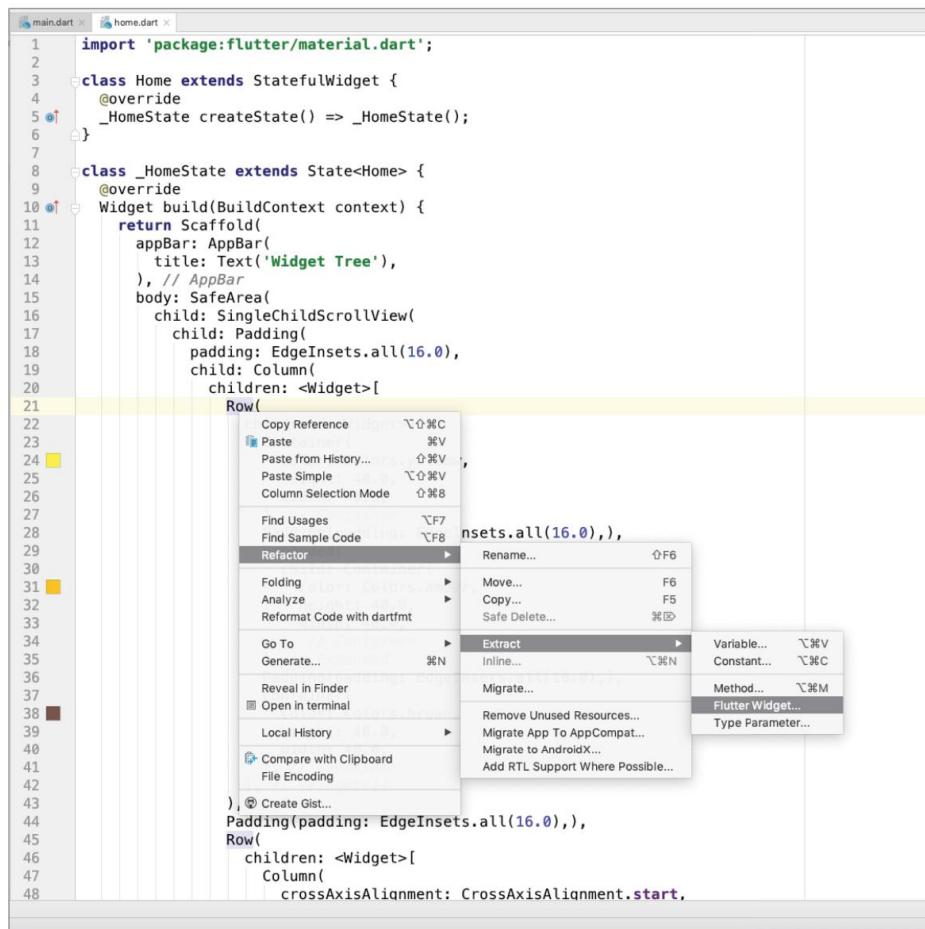
PRUÉBALO Refactorización con una clase de widget para crear un árbol de widgets poco profundo

Para refactorizar los widgets, use el patrón de clase de widget para aplinar el árbol de widgets.

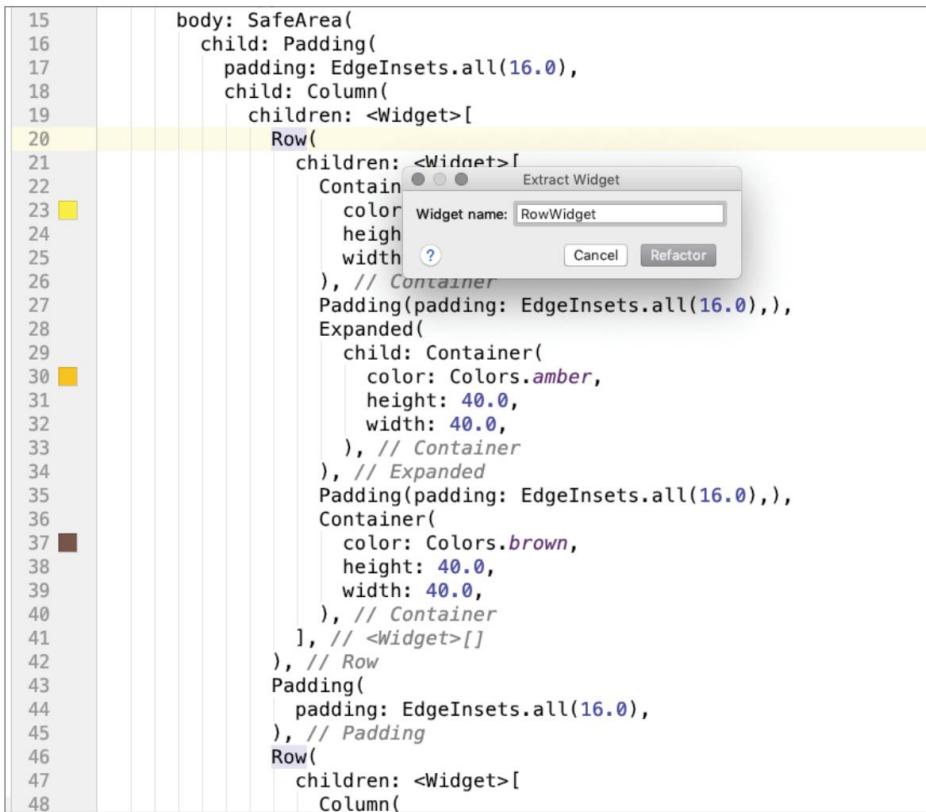
Cree un nuevo proyecto Flutter llamado ch5_widget_tree_performance. Puede seguir las instrucciones del Capítulo 4. Para este proyecto, solo necesita crear la carpeta de páginas . Para mantener este ejemplo simple,

creará las clases de widgets en el archivo home.dart , pero en el Capítulo 7 aprenderá cómo separarlas en archivos separados.

1. Abra el archivo home.dart . Copie el árbol de widgets completo original en home.dart (de la sección "Creación del árbol de widgets completo" de este capítulo) al archivo home.dart de este proyecto .
2. Coloque el cursor en el widget de la primera fila y haga clic con el botón derecho.
3. Seleccione Refactorizar → Extraer → Flutter Widget.



4. En el cuadro de diálogo Extraer widget, ingrese RowWidget para el nombre del widget.



5. El widget Row se reemplaza con la clase de widget RowWidget() . Dado que el widget Fila no cambie el estado, agregue la palabra clave const antes de llamar a la clase RowWidget() . Desplácese hasta la parte inferior del código y los widgets se refactorizarán muy bien en la clase RowWidget (StatelessWidgetWidget) .

```
class RowWidget extiende StatelessWidget { const
  RowWidget({  
    Clave  
    clave, }) : super(clave: clave);  
  
  @anular  
  Compilación del widget (contexto BuildContext) {  
    imprimir('WidgetFila');  
  
  return  
    Row( children:  
  
      <Widget>[ Container( color:  
        Colors.yellow,  
        height: 40.0,  
  
        width:  
          40.0, ), Padding( padding:  
            EdgeInsets.all(16.0), ).
```

```
        Expandido( hijo:  
            Contenedor( color:  
                Colors.amber, altura: 40.0,  
                ancho: 40.0, ), ),  
            Padding( padding:  
  
                EdgeInsets.all(16.0), ), Contenedor( color:  
  
                    Colors.brown,  
                    altura: 40.0, ancho : 40.0, ], );  
  
    }  
}
```

6. Continúa y refactorice las otras filas (clase RowAndColumnWidget) y los widgets Row y Stack (clase RowAndStackWidget).

El código fuente completo de home.dart se muestra a continuación. Observe cómo se aplana el árbol de widgets, lo que facilita la lectura. Decidir cuán superficial hacer el árbol de widgets depende de cada circunstancia y de su preferencia personal.

```
// home.dart import  
'paquete:flutter/material.dart';  
  
class Home extiende StatelessWidget {  
    @anular  
    _HomeState createState() => _HomeState();  
}  
  
class _HomeState extiende State<Home> {  
    @anular  
    Creación de widgets (contexto BuildContext) { return  
        Scaffold (  
            appBar: AppBar(título:  
                Texto('Árbol de widgets'), ), cuerpo:  
  
                SafeArea(  
                    hijo: SingleChildScrollView(  
                        niño: relleno (   
                            relleno: EdgeInsets.all(16.0), child:  
                            Column( children:  
                                <Widget>[ const RowWidget(),  
                                Padding( padding:  
  
                                    EdgeInsets.all(16.0), ), const RowAndColumnWidget(), ],  
  
                            ),  
                        ),  
                    ),  
                ),  
            ),  
        );  
    }  
}
```

```
        ), ), ), );  
    }  
  
    class RowWidget extiende StatelessWidget { const RowWidget({  
  
        Clave  
        clave, }) : super(clave: clave);  
  
        @anular  
        Compilación del widget (contexto BuildContext) {  
            return  
                Row( children:  
  
                    <Widget>[ Container( color:  
                        Colors.yellow,  
                        height: 40.0,  
  
                        width:  
                        40.0, ), Padding( padding: EdgeInsets.all(16.0), ),  
  
                    Expanded( child:  
                        Container( color: Colors.  
                            ámbar, alto: 40.0,  
                            ancho: 40.0, ), ),  
  
                    Padding( relleno: EdgeInsets.all(16.0), ),  
  
                    Container( color:  
                        Colors.brown, alto: 40.0,  
                        ancho: 40.0, ], );  
  
                }  
  
                class RowAndColumnWidget extiende StatelessWidget {  
                    const RowAndColumnWidget({  
                        Clave  
                        clave, }) : super(clave: clave);  
  
                        @anular  
                        Compilación del widget (contexto BuildContext) {  
                            return  
                                Row( children:  
  
                                    <Widget>[ Column( crossAxisAlignment: CrossAxisAlignment.start,
```

```
mainAxisSize: MainAxisSize.max, children:  
<Widget>[ Container( color:  
    Colors.yellow,  
    height: 60.0, width: 60.0, ),  
    Padding( padding:  
  
        EdgeInsets.all(16.0), ), Container( color:  
  
    Colors.amber ,  
    alto: 40.0, ancho: 40.0, ),  
    Padding( relleno:  
  
        EdgeInsets.all(16.0), ), Container( color:  
  
    Colors.brown,  
    alto: 20.0, ancho: 20.0, ),  
    Divider(), const  
  
    RowAndStackWidget(), Divider(),  
    Text('Fin de la  
línea. Fecha: ${DateTime.now()}'), ], ), ], );  
}  
  
clase RowAndStackWidget extiende StatelessWidget {  
    const RowAndStackWidget({  
        Clave  
    clave, }) : super(clave: clave);  
  
    @anular  
    Compilación del widget (contexto BuildContext) {  
        return Fila( niños:  
            <Widget>[  
                CírculoAvatar(  
                    backgroundColor: Colors.lightGreen, radio: 100.0,  
                    child: Stack( children:  
  
                        <Widget>[ Container( height:  
                            100.0, width:  
                            100.0, color:  
                            Colors.yellow, ),  
                            Container( height: 60.0,
```

```
        ancho: 60.0,  
        color: Colores.ámbar, ),  
  
        Contenedor( alto:  
            40.0, ancho:  
            40.0, color:  
  
            Colores.marrón, ), ], ), ], );  
  
    }}
```

Cómo funciona

La creación de un árbol de widgets poco profundo significa que cada widget se separa en su propia clase de widget por funcionalidad. Tenga en cuenta que la forma en que separa los widgets será diferente según la funcionalidad necesaria.

En este ejemplo, creó la clase de widget RowWidget() para construir el widget Row horizontal con widgets secundarios. La clase de widget RowAndColumnWidget() es un excelente ejemplo de aplazarlo aún más llamando a la clase de widget RowAndStackWidget() para uno de los widgets secundarios de Column . Se separa la calificación agregando el RowAndStackWidget() adicional para mantener plano el árbol de widgets porque la clase RowAndStackWidget() crea un widget con varios elementos secundarios.

En el código fuente del proyecto, agregué para su conveniencia un botón que aumenta el valor de un contador, y cada clase de widget usa una declaración de impresión para mostrar cada vez que se llama a cada uno cuando cambia el estado del contador.

El siguiente es el archivo de registro que se muestra cada vez que se llama a un widget. Cuando se toca el botón, el widget CounterTextWidget se vuelve a dibujar para mostrar el nuevo valor del contador, pero observe que los widgets RowWidget, RowAndColumnWidget y RowAndStackWidget se llaman solo una vez y no se vuelven a dibujar cuando cambia el estado. Al utilizar la técnica de clase de widget, solo se llaman los widgets que necesitan volver a dibujarse, lo que mejora el rendimiento general.

```
// Aplicación cargada por  
primera vez flutter:  
RowWidget flutter:  
RowAndColumnWidget flutter:  
RowAndStackWidget flutter: CounterTextWidget 0  
  
// Se llama al botón de aumento de valor y observa que los widgets de fila no se vuelven a dibujar flutter:  
CounterTextWidget 1 flutter:  
CounterTextWidget 2 flutter:  
CounterTextWidget 3
```

RESUMEN

En este capítulo, aprendió que el árbol de widgets es el resultado de widgets anidados. A medida que aumenta la cantidad de widgets, el árbol de widgets se expande rápidamente y disminuye la legibilidad y la capacidad de administración del código.

Llamo a esto el árbol completo de widgets. Para mejorar la legibilidad y la capacidad de administración del código, puede separar los widgets en su propia clase de widgets, creando un árbol de widgets menos profundo. En cada aplicación, debe esforzarse por mantener el árbol de widgets poco profundo.

Al refactorizar con una clase de widget, puede aprovechar la reconstrucción del subárbol de Flutter, lo que mejora el rendimiento.

En el próximo capítulo, verá el uso de widgets básicos. Aprenderá a implementar diferentes tipos de botones, imágenes, iconos, decoradores, formularios con validación y orientación de campos de texto.

LO QUE APRENDISTE EN ESTE CAPÍTULO

TEMA	CONCEPTOS CLAVE
Widgets de anidamiento	Aprendió sobre los widgets disponibles para Material Design y Cupertino y cómo anidar widgets para componer el diseño de la interfaz de usuario. Los widgets básicos que cubrimos para Material Design fueron Scaffold, AppBar, CircleAvatar, Divider, SingleChildScrollView, Padding, Column, Row, Container, Expanded, Text, Stack y Positioned.
	Los widgets básicos que cubrimos para Cupertino fueron CupertinoPageScaffold, CupertinoTabScaffold y CupertinoNavigationBar.
Creación de un árbol de widgets completo	Un árbol de widgets completo es el resultado de anidar widgets para crear la interfaz de usuario de la página. Cuantos más widgets se agreguen, más difícil será leer y administrar el código.
Crear un árbol de widgets poco profundo	Un árbol de widgets poco profundo es el resultado de separar los widgets en secciones manejables para realizar cada tarea. Los widgets se pueden separar mediante una variable constante, un método o una clase de widget. El objetivo es mantener el árbol de widgets poco profundo para mejorar la legibilidad y la capacidad de administración del código. Para mejorar el rendimiento, puede refactorizar utilizando la clase de widget que aprovecha la reconstrucción del subárbol de Flutter.

PARTE II

Aleteo Intermedio: Descarnar

Fuera de una aplicación

CAPÍTULO 6: Uso de widgets comunes

CAPÍTULO 7: Agregar animación a una aplicación

CAPÍTULO 8: Creación de la navegación de una aplicación

CAPÍTULO 9: Creación de listas de desplazamiento y efectos

CAPÍTULO 10: Planos de construcción

CAPÍTULO 11: Aplicación de la interactividad

CAPÍTULO 12: Plataforma de escritura: código nativo

6

Uso de widgets comunes

LO QUE APRENDERÁS EN ESTE CAPÍTULO

Cómo usar widgets básicos como Scaffold, AppBar, SafeArea, Container, Text, RichText, Columna y Fila, así como diferentes tipos de botones

Cómo anidar los widgets de Columna y Fila para crear diferentes diseños de interfaz de usuario

Formas de incluir imágenes, íconos y decoradores

Cómo usar widgets de campo de texto para recuperar, validar y manipular datos

Cómo comprobar la orientación de tu aplicación

En este capítulo, aprenderá a usar los widgets más comunes. Los llamo nuestros bloques de construcción básicos para crear hermosas UI y UX. Aprenderá a cargar imágenes localmente o a través de la Web a través de un localizador uniforme de recursos (URL), usar los íconos de componentes de materiales enriquecidos incluidos y aplicar decoradores para mejorar la apariencia de los widgets o usarlos como guías de entrada para los campos de entrada. . También explorará cómo aprovechar el widget de formulario para validar los widgets de entrada de campos de texto como un grupo, no solo individualmente. Además, para tener en cuenta la variedad de tamaños de dispositivos, verá cómo usar el widget MediaQuery u OrientationBuilder es una excelente manera de detectar la orientación, porque es extremadamente importante usar los widgets de orientación y diseño del dispositivo en función de la posición vertical u horizontal. Por ejemplo, si el dispositivo está en modo vertical, puede mostrar una fila de tres imágenes, pero cuando el dispositivo está en modo horizontal, puede mostrar una fila de cinco imágenes, ya que el ancho es un área mayor que en el modo vertical.

USO DE WIDGETS BÁSICOS

Al crear una aplicación móvil, normalmente implementará ciertos widgets para la estructura base. Es necesario familiarizarse con ellos.

Scaffold Como aprendió en el Capítulo 4, "Creación de una plantilla de proyecto inicial", el widget Scaffold implementa el diseño visual básico de Material Design, lo que le permite agregar fácilmente varios widgets como AppBar, BottomAppBar, FloatingActionButton, Drawer, Snack Bar, BottomSheet, y más.

AppBar El widget de AppBar generalmente contiene el título estándar, la barra de herramientas, el encabezado y las propiedades de acciones (junto con los botones), así como muchas opciones de personalización.

título La propiedad del título se implementa normalmente con un widget de texto . Puede personalizarlo con otros widgets, como un widget DropdownButton . **principal** La propiedad

principal se muestra antes de la propiedad del título . Por lo general, este es un IconButton o Botón Atrás.

acciones La propiedad acciones se muestra a la derecha de la propiedad del título . Es una lista de widgets alineados en la parte superior derecha de un widget de AppBar , generalmente con un IconButton o Botón de menú emergente.

espacio flexible La propiedad espacio flexible se apila detrás de la barra de herramientas o el widget de barra de pestañas . La altura suele ser la misma que la altura del widget AppBar . Por lo general, se aplica una imagen de fondo a la propiedad flexibleSpace , pero se puede usar cualquier widget, como un ícono .

SafeArea El widget SafeArea es necesario para los dispositivos actuales, como el iPhone X o los dispositivos Android con una muesca (un corte parcial que oscurece la pantalla, generalmente ubicado en la parte superior del dispositivo). El widget SafeArea agrega automáticamente suficiente relleno al widget secundario para evitar intrusiones por parte del sistema operativo. Opcionalmente, puede pasar una cantidad mínima de relleno o un valor booleano para no aplicar el relleno en la parte superior, inferior, izquierda o derecha.

Contenedor El widget Contenedor es un widget de uso común que permite la personalización de su widget secundario . Puede agregar fácilmente propiedades como color, ancho, alto, relleno, margen, borde, restricción, alineación, transformación (como rotar o cambiar el tamaño del widget) y muchas otras. La propiedad secundaria es opcional y el widget Contenedor se puede usar como un marcador de posición vacío (invisible) para agregar espacio entre los widgets.

Texto El widget Texto se utiliza para mostrar una cadena de caracteres. El constructor de texto toma los argumentos cadena, estilo, líneas máximas, desbordamiento, alineación de texto y otros. Un constructor es cómo se pasan los argumentos para inicializar y personalizar el widget de texto .

RichText El widget de RichText es una excelente manera de mostrar texto utilizando varios estilos. El widget RichText toma TextSpans como elementos secundarios para diseñar diferentes partes de las cadenas.

Columna Un widget de columna muestra sus hijos verticalmente. Se necesita una propiedad secundaria que contenga una matriz de List<Widget>, lo que significa que puede agregar varios widgets. Los niños se alinean verticalmente sin ocupar toda la altura de la pantalla. Cada widget secundario se puede incrustar en un widget ampliado para llenar el espacio disponible. CrossAxisAlignmentAlignment, MainAxisAlignment y MainAxisSize se pueden usar para alinear y dimensionar cuánto espacio está ocupado en el eje principal.

Fila Un widget de fila muestra sus hijos horizontalmente. Toma una propiedad secundaria que contiene una matriz de List<Widget>. Las mismas propiedades que contiene la columna se aplican al widget de fila con la excepción de que la alineación es horizontal, no vertical.

Botones Hay una variedad de botones para elegir para diferentes situaciones tales como RaisedButton, FloatingActionButton, FlatButton, IconButton, PopupMenuButton y Barra de botones.

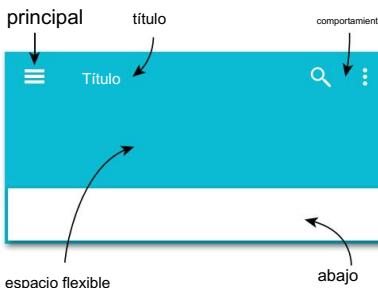
PRUÉBELO Adición de widgets de AppBar

Cree un nuevo proyecto de Flutter y asígnele el nombre ch6_basics; puede seguir las instrucciones en el Capítulo 4. Para este proyecto, necesita crear solo la carpeta de páginas . El objetivo de esta aplicación es brindar una idea de cómo usar los widgets básicos, no necesariamente para diseñar la interfaz de usuario más atractiva. En el Capítulo 10, "Creación de diseños", se centrará en la creación de diseños complejos y atractivos.

1. Abra el archivo main.dart . Cambie la propiedad primarySwatch de azul a verde claro.

muestra primaria: Colors.lightGreen,

2. Abra el archivo home.dart . Comience por personalizar las propiedades del widget AppBar .



3. Agregue a la barra de aplicaciones un IconButton principal. Si anula la propiedad principal , por lo general es una IconButton o Botón Atrás.

```
principal: IconButton(
  icon: Icon(Icons.menu),
  onPressed: () {}),
```

4. La propiedad del título suele ser un widget de texto , pero se puede personalizar con otros widgets, como un botón desplegable. Siguiendo las instrucciones del Capítulo 4, ya ha agregado el widget Texto a la propiedad del título ; si no, agregue el widget de Texto con un valor de 'Inicio'.

```
título: Texto('Inicio'),
```

5. La propiedad actions toma una lista de widgets; agregue dos widgets IconButton .

acciones:

```
<Widget>[ IconButton( icon:
  Icon(Icons.search),
  onPressed: () {}), IconButton(
  icon: Icon(Icons.more_vert), onPressed: () {}),
```

],

6. Debido a que está utilizando un ícono para la propiedad flexibleSpace , agreguemos un SafeArea y un ícono como un niño

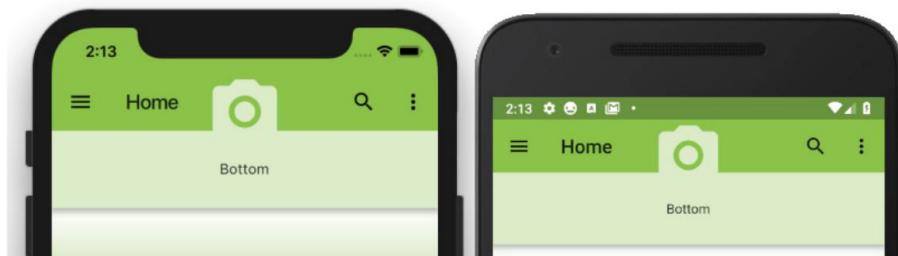
```
espacio flexible: área segura (
  niño: ícono (
```

Iconos.foto_cámara, tamaño:
75,0, color:
Colores.blanco70,),),

Sin área segura

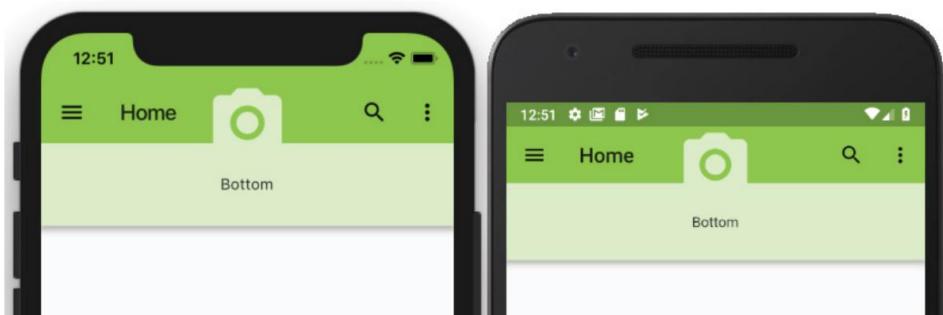


Con SafeArea



7. Agregue un tamaño preferido para la propiedad inferior con un contenedor para un elemento secundario.

```
abajo: PreferredSize( child:  
Container(  
color: Colors.lightGreen.shade100, altura: 75,0,  
ancho: doble.infinito,  
niño: Centro (niño: Texto ('Abajo'),  
,  
),  
tamaño preferido: tamaño.desde la altura (75,0), ),
```



Cómo funciona

Aprendió a personalizar el widget AppBar mediante el uso de widgets para establecer las propiedades de título, barra de herramientas, encabezado y acciones . Todas las propiedades que aprendió en este ejemplo están relacionadas con la personalización de AppBar.

En el Capítulo 9, "Creación de listas de desplazamiento y efectos", aprenderá a usar el widget SliverAppBar , que es una barra de aplicaciones incrustada en una astilla que usa CustomScrollView, lo que hace que cualquier aplicación cobre vida con personalizaciones precisas, como la animación de paralaje. Me encanta usar astillas porque agregan una capa adicional de personalización.

En la siguiente sección, aprenderá a personalizar la propiedad del cuerpo Scaffold anidando widgets para crear el contenido de la página.

Área segura

El widget SafeArea es imprescindible para los dispositivos actuales, como el iPhone X o los dispositivos Android con una muesca (un corte parcial que oscurece la pantalla, generalmente ubicado en la parte superior del dispositivo). El widget SafeArea agrega automáticamente suficiente relleno al widget secundario para evitar intrusiones por parte del sistema operativo. Opcionalmente, puede pasar el relleno mínimo o un valor booleano para no aplicar relleno en la parte superior, inferior, izquierda o derecha.

PRUÉBELO Agregando un SafeArea al cuerpo

Continúe modificando el archivo home.dart .

Agregue un widget de relleno a la propiedad del cuerpo con un SafeArea como elemento secundario. Debido a que este ejemplo incluye diferentes usos de widgets, agregue un SingleChildScrollView como elemento secundario de SafeArea. SingleChildScrollView permite al usuario desplazarse y ver widgets ocultos; de lo contrario, el usuario ve una barra amarilla y negra que indica que los widgets se están desbordando.

cuerpo:

Relleno(relleno: EdgeInsets.all(16.0),

```
hijo: SafeArea( hijo:  
    SingleChildScrollView(  
        niño: Columna( niños:  
            <Widget>[
```

],),),),),

Sugerencia de ejemplo de ajuste

de widget Hay una excelente manera de ajustar un widget actual como elemento secundario de otro widget. Coloque el cursor sobre el widget actual para envolver y luego presione Opción + Intro en su teclado. Aparece el dardo/asistencia rápida. Elija la opción Envolver con nuevo widget.

No agregue los siguientes pasos a su proyecto; este es un consejo sobre cómo envolver rápidamente un widget con otro.

1. Coloque el cursor en el widget para envolver.
 2. Pulse Opción+Intro (Alt+Intro en Windows). Aparece el dardo/asistencia rápida.



3. Seleccione una opción Ajustar con nuevo widget, como body: widget(child: Container()).
 4. Cambie el nombre del widget a SafeArea y observe que child: es automáticamente el widget Container() .

Asegúrese de agregar una coma después del widget Container() , como se muestra aquí. Colocar una coma después de cada propiedad garantiza el formato correcto de Flutter en varias líneas.

cuerpo: SafeArea(hijo: Contenedor(),).

Cómo funciona

Agregar el widget SafeArea ajusta automáticamente el relleno para dispositivos que tienen una muesca. Todos los widgets secundarios de SafeArea están restringidos al relleno correcto.

Envase

El widget Contenedor tiene una propiedad de widget secundaria opcional y se puede usar como un widget decorado con un borde personalizado, color, restricción, alineación, transformación (como rotar el widget) y más.

Este widget se puede utilizar como un marcador de posición vacío (invisible) y, si se omite un elemento secundario, se ajusta al tamaño de pantalla completo disponible.

PRUEBE Agregar un contenedor

Continúe modificando el archivo home.dart . Dado que desea mantener su código legible y manejable, creará clases de widgets para construir cada sección de widget de cuerpo de la lista de widgets de Columna .

1. Agregue a la propiedad del cuerpo un widget Padding con la propiedad secundaria configurada como un widget SafeArea . Agregue al elemento secundario SafeArea un SingleChildScrollView . Agregue una columna al elemento secundario SingleChildScrollView . Para los elementos secundarios Column , agregue la llamada a la clase de widget ContainerWithBoxDecorationWidget() , que creará a continuación. Asegúrese de que la clase de widget use la palabra clave const para aprovechar el almacenamiento en caché (rendimiento).

```
cuerpo: relleno
    (relleno: EdgeInsets.all (16.0), niño: SafeArea
    (niño:
        SingleChildScrollView (
            niño: Column( niños:
            <Widget>[
                const ContainerWithBoxDecorationWidget(), ], ), ), ), ),
```

2. Cree la clase de widget ContainerWithBoxDecorationWidget() después de que la clase Home se extienda Widget sin estado {...} . La clase de widget devolverá un Widget . Tenga en cuenta que cuando refactoriza creando clases de widgets, son del tipo StatelessWidget a menos que especifique usar un StatefulWidget .

```
class ContainerWithBoxDecorationWidget extiende StatelessWidget { const
    ContainerWithBoxDecorationWidget({
        Clave
        clave, }) : super(clave: clave);

    @anular
    Compilación del widget (contexto BuildContext)
    { columna de retorno
        (hijos: <Widget>[
            Envase(), ], );
```

}

3. Comience a agregar propiedades al Contenedor agregando una altura de 175,0 píxeles. Tenga en cuenta la coma después del número, que separa las propiedades y ayuda a mantener el formato del código Dart. Vaya a la línea siguiente para agregar la propiedad de decoración , que acepta una clase BoxDecoration . La clase BoxDecoration proporciona diferentes formas de dibujar un cuadro y, en este caso, está agregando una clase BorderRadius en la parte inferior izquierda e inferior derecha del contenedor.

```
Contenedor
    (altura: 100.0,
    decoración: BoxDecoration(), ),
```

4. El uso del constructor con nombre BorderRadius.only() le permite controlar los lados para dibujar esquinas redondeadas.

Deliberadamente hice que el radio inferior izquierdo sea mucho más grande que el inferior derecho para mostrar las formas personalizadas que puede crear.

```
BoxDecoration( borderRadius: BorderRadius.only(  
    bottomLeft: Radius.circular(100.0), bottomRight:  
    Radius.circular(10.0), ), ),
```

BoxDecoration también admite una propiedad de degradado . Está usando un LinearGradient, pero también podría haber usado un RadialGradient. LinearGradient muestra los colores del degradado de forma lineal y RadialGradient muestra los colores del degradado de forma circular. Las propiedades de inicio y fin le permiten elegir las posiciones de inicio y fin del degradado mediante la clase AlignmentGeometry .

AlignmentGeometry es una clase base para Alignment que permite una resolución consciente de la dirección. Tiene muchas direcciones para elegir, como Alignment.bottomLeft, Alignment.centerRight y más.

```
comenzar: Alignment.topCenter, finalizar:  
Alignment.bottomCenter,
```

La propiedad colors requiere una lista de tipos de color , List<Color>. La lista de colores se ingresa entre corchetes separados por comas.

```
colores:  
[ Colores.blanco,  
Colores.verde claro.sombra500, ],
```

Aquí está el código fuente completo de la propiedad de degradado:

```
degradado: LinearGradient(  
    comenzar: Alignment.topCenter, end:  
    Alignment.bottomCenter, colors:  
    [ Colors.white,  
  
    Colors.lightGreen.shade500,  
    ],  
,
```

5. La propiedad boxShadow es una excelente manera de personalizar una sombra y toma una lista de BoxShadows, llamada List<BoxShadow>. Para BoxShadow, establezca las propiedades de color, blurRadius y offset .

```
boxShadow:  
  
[ BoxShadow( color: Colors.white,  
    blurRadius: 10.0, offset:  
    Offset(0.0, 10.0),  
    ),  
,
```

La última parte del contenedor es agregar un widget de texto secundario envuelto por un widget de centro . El widget Center le permite centrar el widget secundario en la pantalla.

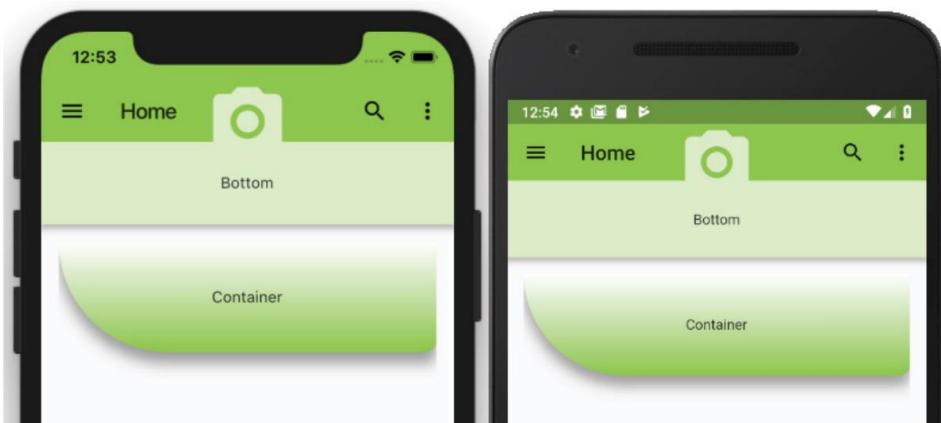
```
hijo: Centro( hijo:  
    Texto('Contenedor'), ),
```

6. Agregue un widget de centro como elemento secundario del contenedor y agregue al elemento secundario del widget de centro un elemento de texto con la cadena Contenedor. (En la siguiente sección, repasaré el widget de texto en detalle).

```
hijo: Centro( hijo:  
    Texto('Contenedor'), ),
```

Aquí está el código fuente completo de la clase de widget ContainerWithBoxDecorationWidget() :

```
class ContainerWithBoxDecorationWidget extends StatelessWidget { const  
    ContainerWithBoxDecorationWidget(  
        Clave  
        clave, ) : super(clave: clave);  
  
    @anular  
    Widget build(BuildContext context) { return  
        Column( children:  
            <Widget>[ Container( height:  
                100.0,  
                decoration:  
                    BoxDecoration( borderRadius:  
                        BorderRadius.only(  
                            bottomLeft: Radius.circular(100.0), bottomRight:  
                                Radius.circular(10.0), gradiente: LinearGradient(  
  
                            comenzar: Alignment.topCenter, end:  
                                Alignment.bottomCenter, colors:  
                                    [ Colors.white,  
  
                                    Colors.lightGreen.shade500, ],  
  
                            ),  
                            boxShadow:  
  
                                [ BoxShadow( color: Colors.grey,  
                                    blurRadius: 10.0, offset:  
                                        Offset(0.0, 10.0),  
                                ),  
                                ],  
                            ),  
                            hijo: Centro( hijo:  
                                RichText( texto:  
                                    Texto('Contenedor'), ),  
                            ),  
                            ],  
                        );  
        }  
    }
```



Cómo funciona

Los contenedores pueden ser widgets poderosos llenos de personalización. Mediante el uso de decoradores, degradados y sombras, puede crear hermosas interfaces de usuario. Me gusta pensar en los contenedores como elementos que mejoran una aplicación de la misma manera que un marco atractivo se suma a una pintura.

Texto

Ya ha utilizado el widget de texto en los ejemplos anteriores; es un widget fácil de usar pero también personalizable. El constructor de texto toma los argumentos cadena, estilo, líneas máximas, desbordamiento, alineación de texto y otros.

```
Text('Flutter World for Mobile', style:  
    TextStyle(fontSize:  
        24.0, color:  
            Colors.deepPurple, decoration:  
                TextDecoration.underline, decorationColor:  
                    Colors.deepPurpleAccent, decorationStyle:  
                        TextDecorationStyle.dotted, fontStyle:  
                            FontStyle.italic,  
                            fontWeight: FontWeight.bold, ),  
        maxLines: 4, overflow:  
            TextOverflow.ellipsis, textAlign:  
                TextAlign.justify, ),
```

Texto rico

El widget RichText es una excelente manera de mostrar texto usando múltiples estilos. El widget RichText toma TextSpan como elementos secundarios para diseñar diferentes partes de las cadenas (Figura 6.1).



FIGURA 6.1: RichText con TextSpan

PRUÉBALO Reemplazo de texto con un contenedor secundario RichText

En lugar de usar el widget de texto contenedor anterior para mostrar una propiedad de texto sin formato, puede usar un widget de texto enriquecido para mejorar y enfatizar las palabras en su cadena. Puede cambiar el color y el estilo de cada palabra.

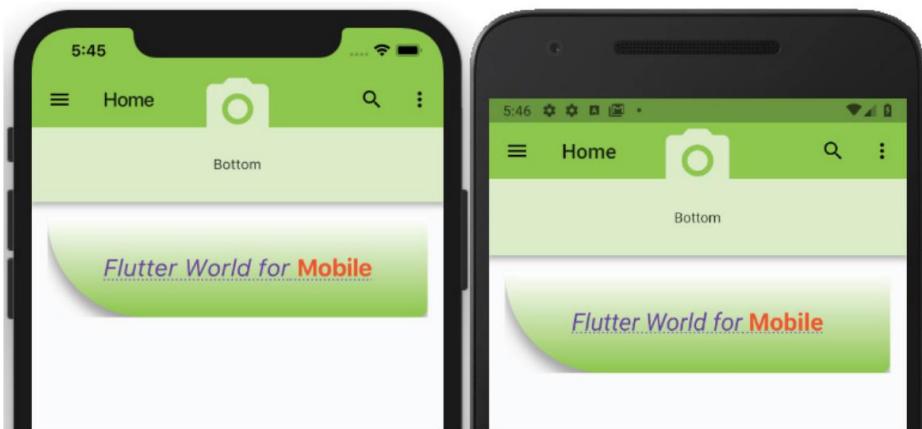
1. Busque el widget de texto secundario del contenedor y elimine Text('Container').

```
hijo: Centro( hijo:  
    Texto('Contenedor'), ),
```

2. Reemplace el widget de texto del contenedor secundario con un widget de texto enriquecido . La propiedad de texto RichText es un objeto TextSpan (clase) que se personaliza mediante el uso de TextStyle para la propiedad de estilo . El TextSpace tiene una lista de elementos secundarios de TextSpan donde se colocan diferentes objetos de TextSpan para formatear diferentes partes de RichText completo .

Al usar el widget RichText y combinar diferentes objetos TextSpan , crea un formato de texto enriquecido como con un procesador de textos.

```
hijo: Centro( hijo:  
    RichText(  
        texto: TextSpan( texto:  
            'Flutter World', estilo:  
                TextStyle( fontSize: 24.0,  
                    color: Colors.deepPurple,  
                    decoration: TextDecoration.underline,  
                    decorationColor: Colors.deepPurpleAccent, decorationStyle:  
                        TextDecorationStyle.dotted, fontStyle: FontStyle.italic,  
                        fontWeight : FontWeight.normal, ), children: <TextSpan>[ TextSpan( text:  
                ' for',  
                    ),  
                TextSpan( texto:  
                    'Móvil', estilo:  
                        TextStyle( color:  
                            Colors.deepOrange, fontStyle:  
                                FontStyle.normal, fontWeight:  
                                    FontWeight.bold),  
                            ),  
                        ],  
                    ),  
                ),  
            ),
```



Cómo funciona

RichText es un widget poderoso cuando se combina con el objeto (clase) TextSpan . Hay dos partes principales para diseñar, la propiedad de texto predeterminada y la lista de elementos secundarios de TextSpan. La propiedad de texto que utiliza TextSpan establece el estilo predeterminado para RichText. La lista de niños de TextSpan le permite usar múltiples objetos de TextSpan para formatear diferentes cadenas.

Columna

Un widget de Columna (Figuras 6.2 y 6.3) muestra sus hijos verticalmente. Toma una propiedad secundaria que contiene una matriz de List<Widget>. Los niños se alinean verticalmente sin ocupar toda la altura de la pantalla. Cada widget secundario se puede incrustar en un widget ampliado para llenar el espacio disponible. Puede usar CrossAxisAlignment, MainAxisAlignment y MainAxisSize para alinear y dimensionar cuánto espacio está ocupado en el eje principal.

Columna

```
(crossAxisAlignment: CrossAxisAlignment.center,
mainAxisAlignment: MainAxisAlignment.spaceEvenly, mainAxisSize:
MainAxisSize.max, children:
<Widget>[ Text('Column
1'), Divider(),
Text('Column
2'), Divider(),
Text('Columna
3'),
],
),
```

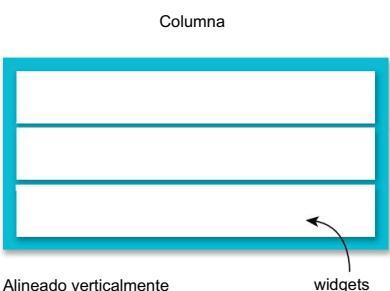


FIGURA 6.2: Widget de columna

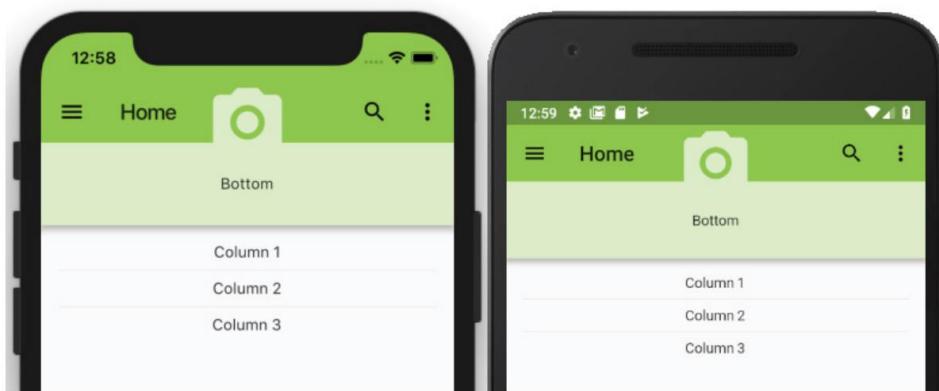


FIGURA 6.3: Widget de columna representado en la aplicación

Fila

Un widget de Fila (Figuras 6.4 y 6.5) muestra sus hijos horizontalmente. Toma una propiedad secundaria que contiene una matriz de List<Widget>. Las mismas propiedades que contiene la columna se aplican al widget de fila con la excepción de que la alineación es horizontal, no vertical.

```
Row( crossAxisAlignment: CrossAxisAlignment.start,
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      mainAxisSize: MainAxisSize.max, children:
      <Widget>[ Row( children:
          <Widget>[ Text('Row 1'),
          Padding(padding:
              EdgeInsets.all(16.0),), Text('Fila 2'), Relleno(relleno:
              EdgeInsets.all(16.0),),
          Text('Fila 3'), ], ), ], ),
```

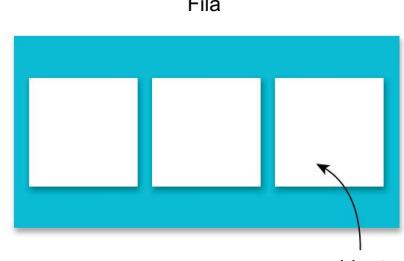


FIGURA 6.4: Widget de fila

Anidamiento de columnas y filas

Una excelente manera de crear diseños únicos es combinar widgets de columna y fila para necesidades individuales. Imagine tener una página de diario con Texto en una columna con una fila anidada que contiene una lista de imágenes (Figuras 6.6 y 6.7).

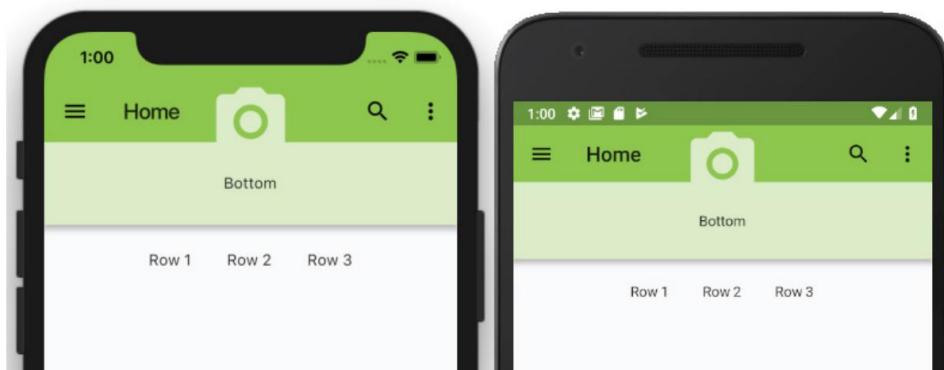


FIGURA 6.5: Widget de fila representado en la aplicación

Agregue un widget de Fila dentro del widget de Columna . Use mainAxis Alignment: MainAxisAlignment.spaceEvenly y agregue tres widgets de texto .

Columna

```
(crossAxisAlignment: CrossAxisAlignment.start,
mainAxisAlignment: MainAxisAlignment.
spaceEvenly,
mainAxisSize: MainAxisSize.max, children:
<Widget>[ Text('Columnas
y anidamiento de filas 1'), Text('Columnas y
anidamiento de filas 2'), Text('Anidamiento de
filas y columnas 3'), Padding(padding:
EdgeInsets.all(16.0),), Row( mainAxisAlignment:
```

```
MainAxisAlignment.spaceEvenly, children: <Widget>[ Text('Row Nesting
1'), Text('Anidamiento de
filas 2'), Text('Anidamiento de
filas 3'), ], ), ], ),
```

Columna con Fila

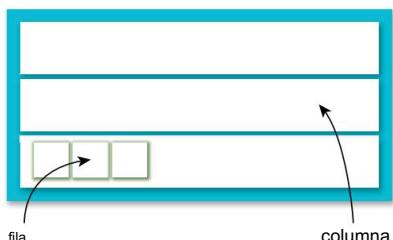


FIGURA 6.6: Anidamiento de columnas y filas

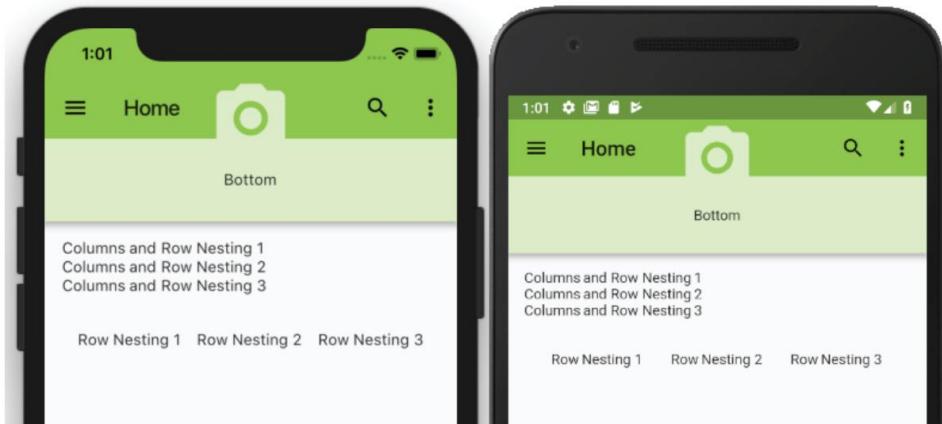


FIGURA 6.7: Widgets de columna y fila representados en la aplicación

PRUEBE Agregar columna, fila y anidar la fila y la columna juntas como clases de widgets

Agregará tres clases de widgets a la sección de propiedades del cuerpo de la lista Columna de widgets. Entre cada clase de widget, agregará un widget Divider() simple para dibujar líneas de separación entre secciones.

1. Agregue los nombres de las clases de widgets ColumnWidget(), RowWidget() y ColumnAndRowNestingWidget() a la lista de widgets secundarios de columna . El widget Columna se encuentra en la propiedad del cuerpo . Agregue un widget Divider() entre cada nombre de clase de widget. Asegúrese de que cada clase de widget use la palabra clave const .

```
cuerpo: relleno
    (relleno: EdgeInsets.all (16.0), niño: SafeArea
    (
        hijo: SingleChildScrollView(
            niño: Columna( niños:
                <Widget>[
                    const ContainerWithBoxDecorationWidget(),
                    Divisor(), const
                    ColumnWidget(),
                    Divisor(), const
                    RowWidget(),
                    Divider(), const
                    ColumnAndRowNestingWidget(), ],
                ),
            ),
        ),
    ),
```

2. Cree la clase de widget ColumnWidget() después de ContainerWithBoxDecorationWidget() clase de

```
widget. class ColumnWidget extiende StatelessWidget
{ const

    ColumnWidget({ Clave clave, }) : super(clave: clave);

    @anular
    Widget compilación (contexto BuildContext)
    { columna de
        retorno (crossAxisAlignment: CrossAxisAlignment.center,
        mainAxisAlignment: MainAxisAlignment.spaceEvenly, mainAxisSize:
        MainAxisSize.max, children:
        <Widget>[ Text('Column
        1'), Divider(),
        Text('Column
        2'), Divisor(),
        Texto('Columna
        3'), ], );

    }

}
```

3. Cree la clase de widget RowWidget() después de la clase de widget ColumnWidget() .

```
clase RowWidget extiende StatelessWidget
{ const

    RowWidget({ Key key, }) : super(key: key);

    @anular
    Compilación del widget (contexto BuildContext) {
        return
            Row( crossAxisAlignment: CrossAxisAlignment.start,
            mainAxisAlignment: MainAxisAlignment.spaceEvenly, mainAxisSize:
            MainAxisSize.max, children:
            <Widget>[ Row( children:

                <Widget>[ Text('Row 1'),
                Padding(padding:
                    EdgeInsets.all (16.0,),), Text('Row 2'), Padding(padding:
                    EdgeInsets.all(16.0,),),
                Text('Row 3'), ], ), ], );

    }

}
```

4. Cree la clase de widget ColumnAndRowNestingWidget() después de la clase de widget RowWidget(). class

```
ColumnAndRowNestingWidget extiende StatelessWidget {  
    const ColumnaYFilaAnidadoWidget({  
        Clave  
        clave, }) : super(clave: clave);  
  
    @anular  
    Widget compilación (contexto BuildContext)  
    { columna de  
        retorno (crossAxisAlignment: CrossAxisAlignment.start,  
        mainAxisAlignment: MainAxisAlignment.spaceEvenly,  
        mainAxisSize: MainAxisSize.max,  
        children:  
            <Widget>[ Text('Columnas y anidamiento de  
                filas 1'), Text('Columnas y anidamiento de  
                filas 2', Text('Columnas y anidamiento de  
                filas 3',), Padding(padding: EdgeInsets.all(16.0),),  
  
        Row( mainAxisAlignment: MainAxisAlignment.spaceEvenly,  
            children: <Widget>[ Text('  
                Anidamiento de filas 1'),  
                Text('Anidamiento de filas  
                2'), Text('Anidamiento de  
                filas 3'), ], ), ], );  
    }  
}
```

Cómo funciona

Columna y Fila son widgets útiles para diseñar vertical u horizontalmente. Anidar los widgets de Columna y Fila crea diseños flexibles necesarios para cada circunstancia. El anidamiento de widgets está en el centro del diseño de los diseños de la interfaz de usuario de Flutter.

Botones

Hay una variedad de botones para elegir, dependiendo de la situación, como FloatingActionButton Button, FlatButton, IconButton, RaisedButton, PopupMenuItem y ButtonBar.

FloatingActionButton El widget

FloatingActionButton generalmente se coloca en la parte inferior derecha o en el centro de la pantalla principal en la propiedad Scaffold floatingActionButton . Use el widget FloatingActionButtonLocation para acoplar (muesca) o flotar sobre la barra de navegación. Para acoplar un botón a la navegación

barra, use el widget `BottomAppBar`. De forma predeterminada, es un botón circular, pero se puede personalizar con la forma de un estadio mediante el constructor con nombre `FloatingActionButton.extended`. En el código de ejemplo, comenté el botón de la forma del estadio para que lo pruebes.

```
floatingActionButtonLocation: FloatingActionButtonLocation.endDocked,  
floatingActionButton: FloatingActionButton( onPressed:  
() {}, child:  
Icon(Icons.play_arrow),  
backgroundColor: Colors.lightGreen.shade100, ), //  
  
o //  
Esto crea una forma de estadio FloatingActionButton //  
floatingActionButton : FloatingActionButton.extended( // onPressed:  
(() {}, // icon:  
Icon(Icons.play_arrow), // label:  
Text('Play'), // ),  
  
bottomNavigationBar:  
  
BottomAppBar( hasNotch: true, color:  
Colors.lightGreen.shade100, child: Row( mainAxisAlignment:  
  
MainAxisAlignment.spaceEvenly,  
children:  
<Widget>[ Icon(Icons.pause),  
  
Icon(Icons.stop), Icon(Icons.access_time), Padding( padding: EdgeInsets.all (32.0), ), ], ), ),  
  
Icon(Icons.stop), Icon(Icons.access_time), Padding( padding: EdgeInsets.all (32.0), ), ], ), ),
```

La Figura 6.8 muestra el widget `FloatingActionButton` en la parte inferior derecha de la pantalla con la muesca habilitada.

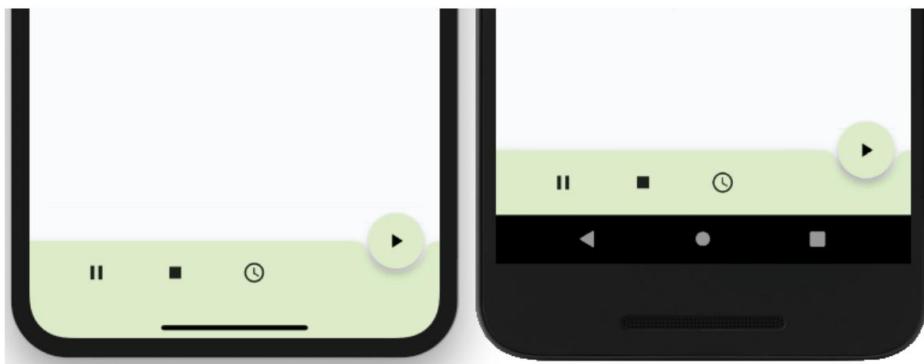


FIGURA 6.8: `FloatingActionButton` con muesca

FlatButton El

widget FlatButton es el botón más minimalista utilizado; muestra una etiqueta de texto sin bordes ni elevación (sombra). Dado que la etiqueta de texto es un widget, puede usar un widget de ícono en su lugar u otro widget para personalizar el botón. Color, HighlightColor, SplashColor, TextColor y otras propiedades se pueden personalizar.

```
// Predeterminado - botón izquierdo  
botón plano (  
    onPressed: () {}, hijo:  
    Text('Bandera'), ),
```

```
// Personalizar - botón derecho
```

```
FlatButton( onPressed: () {},  
    child: Icon(Icons.flag), color:  
    Colors.lightGreen, textColor:  
    Colors.white, ),
```

La Figura 6.9 muestra el widget FlatButton predeterminado a la izquierda y el widget FlatButton personalizado a la derecha.

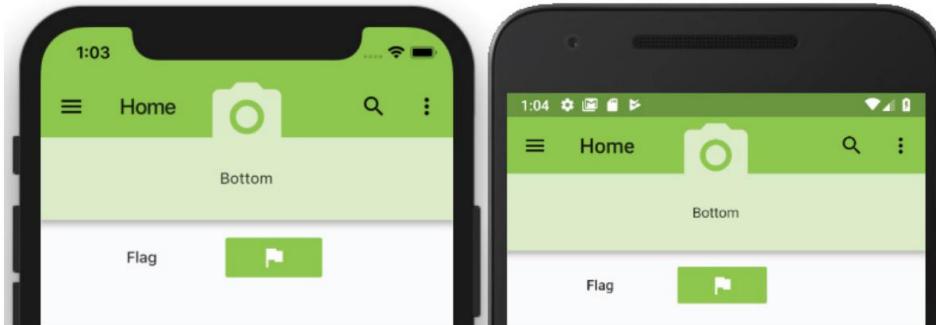


FIGURA 6.9: Botón Plano

Botón elevado

El widget RaisedButton agrega una dimensión y la elevación (sombra) aumenta cuando el usuario presiona el botón.

```
// Predeterminado - botón izquierdo
```

```
RaisedButton(onPressed: ()  
    {}, child: Text('Save'),  
,)
```

```
// Personalizar - botón derecho
```

```
RaisedButton( onPressed:  
() {}, child: Icon(Icons.save),  
color: Colors.lightGreen, ),
```

La figura 6.10 muestra el widget RaisedButton predeterminado a la izquierda y el widget RaisedButton personalizado a la derecha.

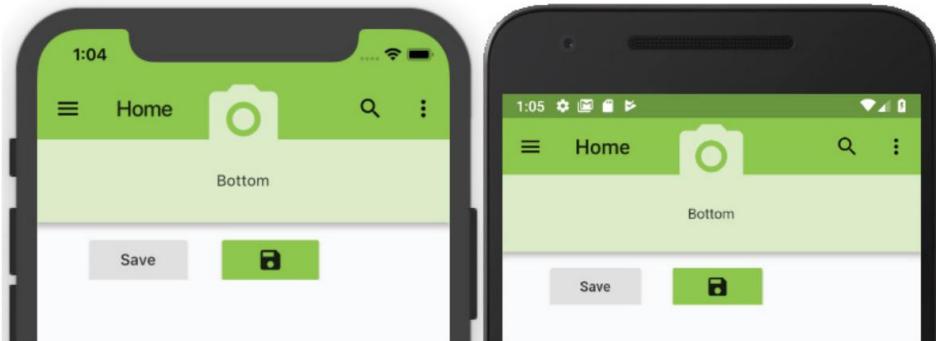


FIGURA 6.10: Botón elevado

IconoBotón

El widget IconButton utiliza un widget de ícono en un widget de componente de material que reacciona a los toques llenándose de color (tinta). La combinación crea un agradable efecto de toque, que informa al usuario de que se ha iniciado una acción.

```
// Por defecto - botón  
izquierdo  
IconButton( onPressed:  
() {}, icon: Icon(Icons.flight), ),
```

```
// Personalizar - botón derecho
```

```
IconButton( onPressed:  
() {}, icon: Icon(Icons.flight),  
iconSize: 42.0,  
color: Colors.white,  
tooltip: 'Flight', ),
```

La figura 6.11 muestra el widget IconButton predeterminado a la izquierda y el widget IconButton personalizado a la derecha.

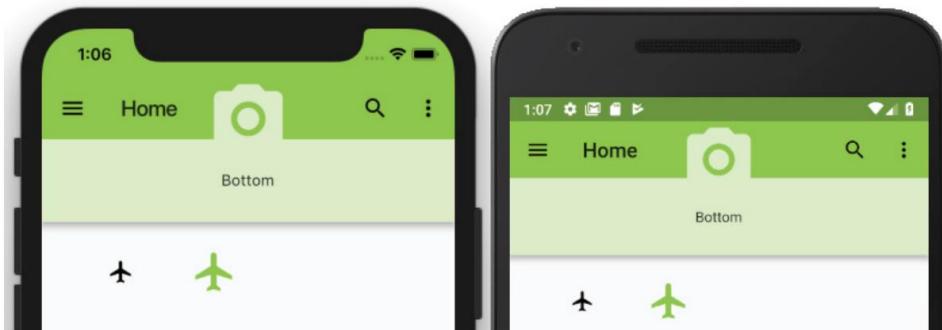


FIGURA 6.11: Botón Icono

BotónMenúEmergente

El widget PopupMenuItem muestra una lista de elementos de menú. Cuando se presiona un elemento del menú, el valor pasa a la propiedad `onSelected`. Un uso común de este widget es colocarlo en la parte superior derecha del widget AppBar para que el usuario seleccione diferentes opciones de menú. Otro ejemplo es colocar el widget Popup MenuItem en medio del widget AppBar que muestra una lista de filtros de búsqueda.

PRUÉBALO Creando el PopupMenuItem y la Clase y Lista de Elementos

Antes de agregar los widgets PopupMenuItem , creamos la clase y la lista necesarias para crear los elementos que se mostrarán. Por lo general, la clase TodoMenuItem (modelo) se crearía en un archivo Dart separado, pero para mantener el ejemplo enfocado, lo agregará al archivo home.dart . En los últimos tres capítulos de este libro, separará las clases en sus propios archivos.

1. Cree una clase TodoMenuItem . Cuando cree esta clase, asegúrese de que no esté dentro de otra clase.

Cree la clase y la lista al final del archivo después del último corchete de cierre, }. La clase TodoMenuItem contiene un título y un ícono.

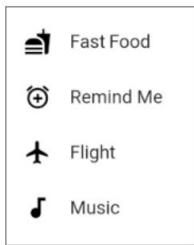
```
clase TodoMenuItem  
{ título de cadena final;  
ícono de ícono final;  
  
TodoMenuItem({this.title, this.icon}); }
```

2. Cree una Lista de TodoMenuItem. Esta `List<TodoMenuItem>` se llamará foodMenuList y contiene una Lista (matriz) de TodoMenuItem.

```
// Crear una lista de elementos de menú para PopupMenuItem  
List<TodoMenuItem> foodMenuList =  
[ TodoMenuItem(title: 'Fast Food', icon: Icon(Icons.fastfood)), TodoMenuItem(title:  
'Remind Me', icon: Icon(Icons.add_alarm)), TodoMenuItem(título: 'Vuelo', ícono:  
Icono(Iconos.vuelo)), TodoMenuItem(título: 'Música', ícono:  
Icono(Iconos.pista de audio)), ];
```

3. Cree un botón de menú emergente. Utilizará un itemBuilder para construir la Lista de TodoMenuItems. Si no configura un ícono para PopupMenuItem, se usa un ícono de menú predeterminado de manera predeterminada. onSelected recuperará el elemento seleccionado en la lista . Use itemBuilder para crear una lista de foodMenuList y asigne a TodoMenuItem. Se devuelve un PopupMenuItem para cada elemento de foodMenuList. Para el elemento secundario PopupMenuItem , utiliza un widget de fila para mostrar los widgets de icono y texto juntos.

```
PopupMenuButton<TodoMenuItem>(
    icon: Icon(Icons.view_list),
    onSelected: ((valueSelected) {
        print('valueSelected: $valueSelected.title');
    }), itemBuilder: (BuildContext context) {
        return foodMenuList.map((TodoMenuItem todoMenuItem) {
            return PopupMenuItem<TodoMenuItem>(
                value: todoMenuItem,
                child: Row(
                    children: [
                        <Widget>[ Icon(todoMenuItem.icon.icon),
                        Padding(padding: EdgeInsets.all(8.0),),
                        Text(todoMenuItem.title),
                    ],
                ),
            ).Listar();
        });
    },
);
```



4. Modifique la propiedad inferior de AppBar agregando el nombre de clase del widget:

PopupMenuWidget().

abajo: PopupMenuItemWidget(),

5. Cree la clase de widget PopupMenuItemWidget() después de la clase de widget ColumnAndRowNestingWidget() . Dado que la propiedad inferior espera un PreferredSizeWidget, usa la palabra clave implements PreferredSizeWidget en la declaración de la clase. La clase extiende StatelessWidget e implementa PreferredSizeWidget.

Después de la compilación del widget, implemente el captador @override de tamaño preferido ; este es un paso obligatorio porque el propósito de PreferredSizeWidget es proporcionar el tamaño del widget; en este ejemplo, establecerá la propiedad de altura . Sin este paso, no tendríamos un tamaño especificado.

```
@anular
// implementar tamaño preferido
Tamaño obtener tamaño preferido => Tamaño.desdeAltura(75.0);
```

La siguiente es la clase de widget PopupMenuItemWidget completa . Tenga en cuenta que la propiedad de altura del widget Contenedor utiliza la propiedad preferentSize.height que configuró en el getter PreferredSize Widget .

```
clase PopupMenuItemWidget extiende StatelessWidget implementa PreferredSizeWidget {
    const PopupMenuItemWidget({
        Clave
        clave, }) : super(clave: clave);

    @anular
    Widget compilación (contexto BuildContext) { return
        Container (
            color: Colors.lightGreen.shade100, altura:
            tamaño preferido.altura, ancho:
            doble.infinito, hijo: Centro( hijo:

                PopupMenuItem<TodoMenuItem>( icon:
                    Icon(Icons.view_list), onSelected:
                    ((valueSelected) { print(' valueSelected: $
                        {valueSelected.title}'); }), itemBuilder: (contexto BuildContext)

                { return foodMenuList.map((TodoMenuItem
                    todoMenuItem) {
                        return PopupMenuItem<TodoMenuItem>(valor:
                            todoMenuItem, hijo: Fila(
                                niños:
                                    <Widget>[ Icon(todoMenuItem.icon.icon),

                                Padding( padding: EdgeInsets.all(8.0), ,

                                    Text(todoMenuItem.title), ], ), ); }).Listar(); }, ), ), );
    }

    @override //
    implementar tamaño preferido
    Tamaño obtener tamaño preferido => Tamaño.desdeAltura(75.0);
}
```

Cómo funciona

El widget PopupMenuItem es un excelente widget para mostrar una lista de elementos, como opciones de menú. Para la lista de elementos, creó una clase TodoMenuItem para contener un título y un ícono. Tú creaste el

foodMenuList, que es una lista de cada TodoMenuItem. En este caso, los elementos de la lista están codificados de forma rígida, pero en una aplicación del mundo real, los valores se pueden leer desde un servicio web. En los Capítulos 14, 15 y 16, implementará Cloud Firestore para acceder a datos desde un servidor web.

Barra de botones

El widget ButtonBar (Figura 6.12) alinea los botones horizontalmente. En este ejemplo, el widget ButtonBar es un elemento secundario de un widget Container para darle un color de fondo.

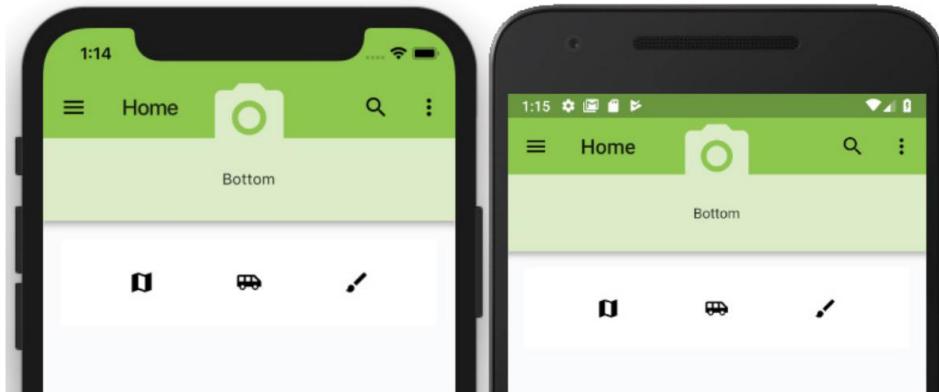


FIGURA 6.12: Barra de botones

```
Contenedor( color: Colors.white70,
hijo: ButtonBar( alineación:
    MainAxisAlignment.spaceEvenly, hijos: <Widget>[ IconButton(
        icon: Icon(Icons.map), onPressed:
            () {}, ), IconButton( icon:
                Icon(Icons.airport_shuttle), onPressed: () {}, ),
        IconButton( icon:
            Icon(Icons.brush),
            onPressed : () {}, ),
    ],
),
),
```

PRUÉBELO Adición de botones como clases de widgets

Ha examinado los widgets `FloatingActionButton`, `FlatButton`, `RaisedButton`, `IconButton`, `PopupMenu` `Button` y `ButtonBar`. Aquí creará dos clases de widgets para organizar el diseño de los botones.

1. Agregue los nombres de clase de widgets `ButtonsWidget()` y `ButtonBarWidget()` a la lista de widgets secundarios de `columna`. La columna se encuentra en la propiedad del cuerpo. Agregue un widget `Divider()` entre cada nombre de clase de widget. Asegúrese de que cada clase de widget use la palabra clave `const`.

```
cuerpo: relleno  
(relleno: EdgeInsets.all(16.0), niño:  
SafeArea (niño:  
    SingleChildScrollView (  
        niño: Columna( niños:  
            <Widget>[  
                const ContainerWithBoxDecorationWidget(),  
                Divisor(),  
                const ColumnWidget(),  
                Divisor(),  
                const RowWidget(),  
                Divider(),  
                const ColumnAndRowNestingWidget(),  
                Divisor(),  
                const BotonesWidget(),  
                Divisor(),  
                const ButtonBarWidget(), ], ), ), ), ),
```

2. Cree la clase de widget `ButtonsWidget()` después de la clase de widget `ColumnAndRowNestingWidget()`.

La clase devuelve una columna con tres widgets de fila para la lista de elementos secundarios de `Widget`. Cada lista secundaria de Fila de Widget contiene diferentes botones, como `FlatButton`, `RaisedButton` e `IconButton`.

```
class ButtonsWidget extiende StatelessWidget { const  
    ButtonsWidget({  
        Clave  
    clave, }) : super(clave: clave);  
  
    @anular  
    Compilación del widget (contexto BuildContext)  
    { columna de  
        retorno (hijos: <Widget>[  
  
            Fila( niños: <Widget>[  
                Relleno(relleno: EdgeInsets.all(16.0)), FlatButton(  
  
                    onPressed: () {},
```

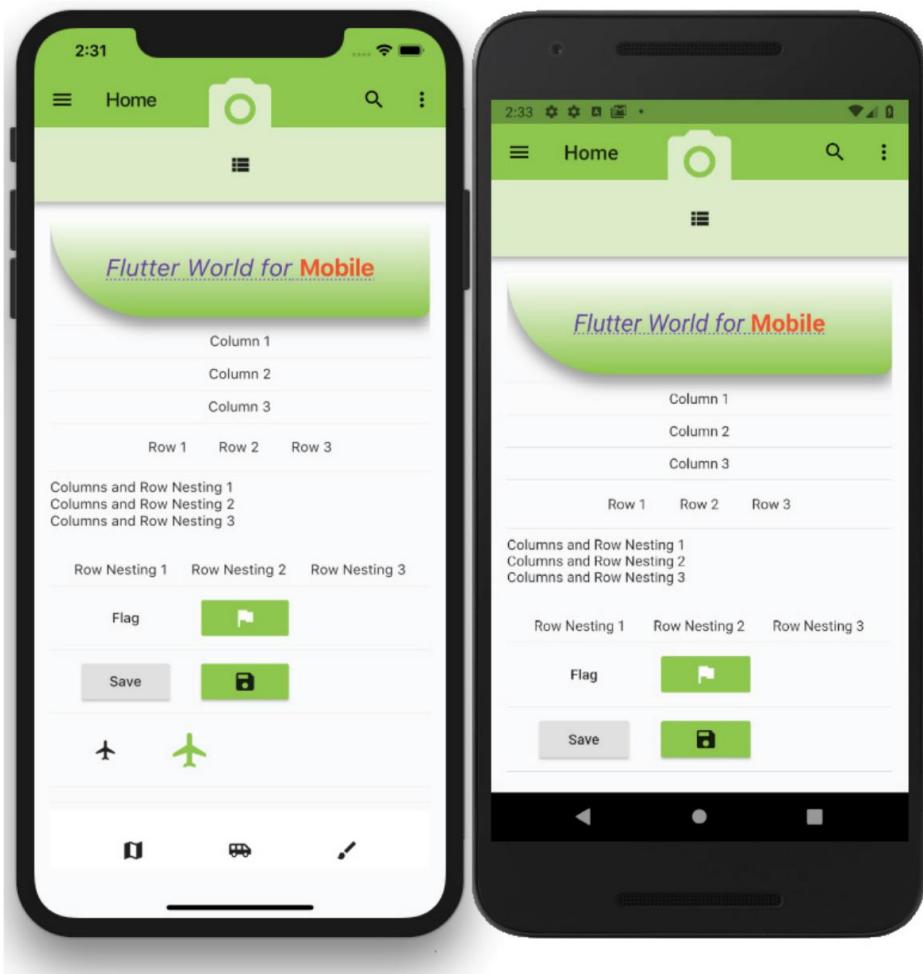
```
niño: Texto('Bandera'), ),  
Relleno(relleno: EdgeInsets.all(16.0)),  
  
FlatButton( onPressed:  
() {}, niño:  
Icono(Icons.bandera), color:  
Colors.lightGreen, textColor:  
  
  
  
  
Colores .white, ), ], ), Divider(), Row( children: <Widget>[  
Padding(padding: EdgeInsets.all(16.0)),  
  
RaisedButton( onPressed:  
() {}, child: Text('Save'), ),  
  
Padding(padding: EdgeInsets.all(16.0)),  
  
RaisedButton( onPressed:  
() {}, child: Icon(Icons.save),  
color: Colors.lightGreen, ), ], ),  
  
  
  
Divider(),  
  
Row( children: <Widget>[  
Relleno(relleno: EdgeInsets.all(16.0)),  
  
IconButton( icono:  
Icon(Icons.flight),  
  
onPressed: () {}, Relleno(relleno:  
EdgeInsets.all(16.0)),  
IconButton( icono:  
Icon( Icons.flight), iconSize:  
42.0, color:  
Colors.lightGreen,  
  
tooltip: 'Flight', onPressed: () {}, ], ),
```

```
        Divisor(), ], );  
  
    })  
  
3. Cree la clase de widget ButtonBarWidget() después de la clase de widget ButtonsWidget() . La clase devuelve un  
contenedor con una barra de botones como elemento secundario. La lista de elementos secundarios de  
ButtonBar de Widget contiene tres widgets IconButton .
```

```
class ButtonBarWidget extiende StatelessWidget { const  
    ButtonBarWidget({  
        Clave  
        clave, }) : super(clave: clave);  
  
    @anular  
    Compilación del widget (contexto BuildContext) { return  
        Container (color:  
            Colors.white70, child: ButtonBar  
            (alineación:  
                MainAxisAlignment.spaceEvenly, children:  
                <Widget>[ IconButton( icon:  
                    Icon(Icons.map),  
                    onPressed: () {}, ),  
                    IconButton( icon:  
  
                        Icon(Icons.airport_shuttle), onPressed: () {}, ),  
                        IconButton( icon:  
  
                            Icon(Icons.brush),  
                            HighlightColor: Colors.purple,  
                            onPressed: () {}, ], ), );  
  
    })
```

Cómo funciona

Los widgets FloatingActionButton, FlatButton, RaisedButton, IconButton, PopupMenuItem y ButtonBar se pueden configurar configurando el ícono de propiedades, iconSize, información sobre herramientas, color, texto y más.



USO DE IMÁGENES E ICONOS

Las imágenes pueden hacer que una aplicación se vea tremenda o fea dependiendo de la calidad de la obra de arte. Las imágenes, los iconos y otros recursos suelen estar integrados en una aplicación.

paquete de activos

La clase AssetBundle brinda acceso a recursos personalizados, como imágenes, fuentes, audio, archivos de datos y más. Antes de que una aplicación Flutter pueda usar un recurso, debe declararlo en el archivo pubspec.yaml .

```
// archivo pubspec.yaml para editar  
# Para agregar activos a su aplicación, agregue una sección de activos, como esta:
```

activos:

- activos/índices/logotipo.png
- activos/índices/trabajo.png
- activos/datos/seed.json

En lugar de declarar cada activo, que puede ser muy largo, puede declarar todos los activos en cada directorio. Asegúrese de terminar el nombre del directorio con una barra diagonal, /. A lo largo del libro, usaré este enfoque al agregar recursos a los proyectos.

```
// archivo pubspec.yaml para editar
# Para agregar activos a su aplicación, agregue una sección de activos, como esta: activos:
  —activos/índices/
  —activos/datos/
```

Imagen

El widget Imagen muestra una imagen de una fuente local o URL (web). Para cargar un widget de imagen , hay algunos constructores diferentes para usar.

`Image()`: recupera la imagen de una clase `ImageProvider`

`Image.asset()`—Recupera la imagen de una clase `AssetBundle` usando una clave

`Image.file()`: recupera la imagen de una clase de archivo

`Image.memory()`: recupera la imagen de una clase `Uint8List`

`Image.network()`: recupera la imagen de una ruta URL

Presione Ctrl+Barra espaciadora para invocar la finalización del código para las opciones disponibles (Figura 6.13).

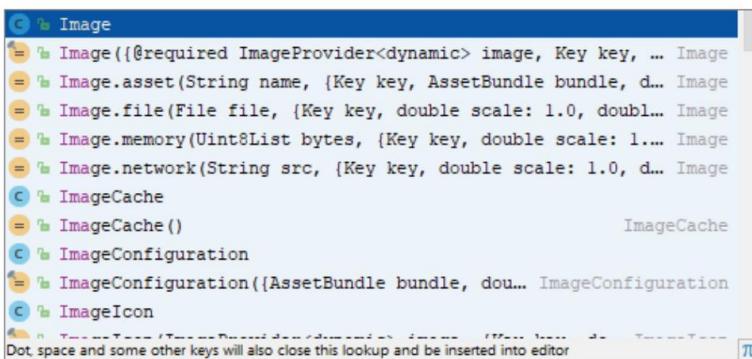


FIGURA 6.13: Finalización del código de imagen

Como nota al margen, el widget Imagen también admite GIF animados.

El siguiente ejemplo usa el constructor de imágenes predeterminado para inicializar la imagen y ajustar los argumentos.

El argumento de la imagen se establece mediante el constructor `AssetImage()` con la ubicación del paquete predeterminado del archivo logo.png . Puede usar el argumento de ajuste para dimensionar el widget Imagen con las opciones de Ajuste de cuadro , como contener, cubrir, llenar, altura de ajuste, ancho de ajuste o ninguno (Figura 6.14).

```
// Imagen: en el lado izquierdo Imagen
(imagen:
  AssetImage ("assets/images/logo.png"), ajuste: BoxFit.cover, ),

// Imagen de una URL - en el lado derecho
Image.network( 'https://
flutter.io/images/catalog-widget-placeholder.png' ).
```

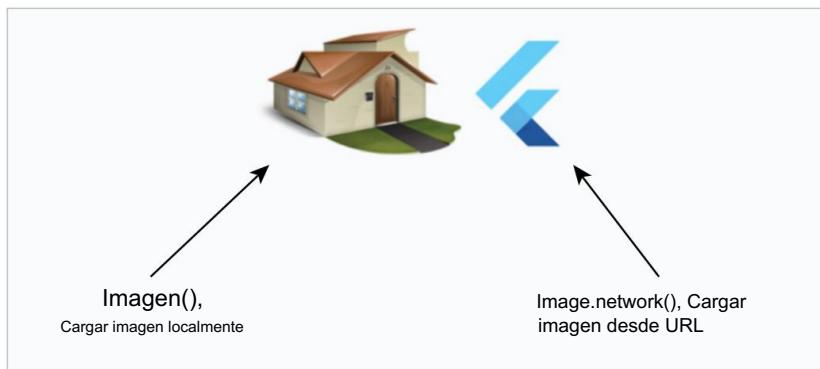


FIGURA 6.14: Imágenes cargadas localmente y desde red (web)

Si agrega color a la imagen, colorea la parte de la imagen y deja las transparencias como están, dando una apariencia de silueta (Figura 6.15).

```
// Imagen
  Imagen( imagen: AssetImage("assets/images/logo.png"), color:
    Colors.deepOrange, ajuste:
    BoxFit.cover,
),
```

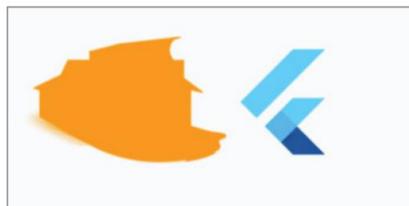


FIGURA 6.15: Imagen estilo silueta

Icono

El widget `Icon` se dibuja con un glifo de una fuente descrita en `IconData`. El archivo `icons.dart` de Flutter tiene la lista completa de íconos disponibles en la fuente `MaterialIcons`. Una excelente manera de agregar íconos personalizados es agregar fuentes al `AssetBundle` que contengan glifos. Un ejemplo es `Font Awesome`, que tiene una lista de íconos de alta calidad y un paquete Flutter. Por supuesto, hay muchos otros íconos de alta calidad disponibles de otras fuentes.

El widget `Icon` le permite cambiar el color, el tamaño y otras propiedades del widget `Icon` (Figura 6.16).

```
Icon( Icons.brush,
  color: Colors.lightBlue, size: 48.0,
),
```


3. Agregue el nombre de clase del widget ImagesAndIconWidget() a la lista de widgets secundarios de la columna . La columna se encuentra en la propiedad del cuerpo .

```
cuerpo: SafeArea(  
    hijo: SingleChildScrollView(  
        niño: relleno (  
            relleno: EdgeInsets.all(16.0), child:  
                Column( children:  
                    <Widget>[ const  
                        ImagesAndIconWidget(), ], ), ), ), ), ),
```

4. Agregue la clase de widget ImagesAndIconWidget() después de que la clase Home extienda StatelessWidget {...}. En la clase de widget, la clase AssetImage carga una imagen local . Usando la Imagen. constructor de red , una imagen se carga mediante una cadena de URL. La propiedad de ancho del widget Imagen usa MediaQuery.of(context).size.width / 3 para calcular el valor del ancho como un tercio del ancho del dispositivo.

```
clase ImagesAndIconWidget extiende StatelessWidget {  
    const ImagesAndIconWidget({  
        Clave  
        clave, }) : super(clave: clave);  
  
    @anular  
    Compilación del widget (contexto BuildContext) {  
        return  
            Row( mainAxisAlignment: MainAxisAlignment.spaceEvenly, children:  
                <Widget>[ Image( image:  
  
                    AssetImage("assets/images/logo.png"), //color: Colors.orange,  
                    fit: BoxFit.cover, width:  
                        MediaQuery.  
                        de(contexto).tamaño.ancho / 3, ), Imagen.red(  
  
                            'https://flutter.io/images/catalog-widget-placeholder.png', ancho:  
                            MediaQuery.of(context).size.width / 3, ), Icon( Icons.brush,  
  
                            color:  
                                Colors.lightBlue,  
                                size : 48.0, ), ], );  
  
    }}  
}
```

Cómo funciona

Al declarar sus activos en el archivo pubspec.yaml , la clase AssetImage puede acceder a ellos desde un AssetBundle. El widget de imagen a través de la propiedad de imagen carga una imagen local con la clase AssetBundle . Para cargar una imagen a través de una red (como la Web), utiliza el constructor Image.network pasando una cadena de URL. El widget Icon usa la biblioteca de fuentes MaterialIcons , que dibuja un glifo de la fuente descrita en la clase IconData .

USO DE DECORADORES

Los decoradores ayudan a transmitir un mensaje según la acción del usuario o personalizan la apariencia de un widget. Hay diferentes tipos de decoradores para cada tarea.

Decoración: la clase base para definir otras decoraciones.

BoxDecoration: proporciona muchas formas de dibujar un cuadro con borde, cuerpo y boxShadow.

InputDecoration: se utiliza en TextField y TextFormField para personalizar el borde, la etiqueta, el ícono y los estilos. Esta es una excelente manera de brindar comentarios al usuario sobre la entrada de datos, especificando una pista, un error, un ícono de alerta y más.

Una clase BoxDecoration (Figura 6.17) es una excelente manera de personalizar un widget de Contenedor para crear formas configurando las propiedades borderRadius, color, degradado y boxShadow .

```
// Contenedor
BoxDecoration(altura:
    100.0, ancho:
    100.0, decoración: BoxDecoration(
        borderRadius: BorderRadius.all(Radius.circular(20.0)),
        color: Colors.orange,
        boxShadow: [ BoxShadow( color:
            Colors.grey,
            blurRadius:
            10.0, offset: Offset(0.0, 10.0),
        ),
    ],
),
)
```

La clase InputDecoration (Figura 6.18) se utiliza con un campo de texto o TextFormField para especificar etiquetas, bordes, iconos, sugerencias, errores y estilos. Esto es útil para comunicarse con el usuario a medida que ingresa datos. Para la propiedad de borde que se muestra aquí, estoy implementando dos formas de personalizarla, con UnderlineInputBorder y con OutlineInputBorder:

```
// Campo de texto
Campo de texto(
```



FIGURA 6.17: BoxDecoration aplicado a un Contenedor

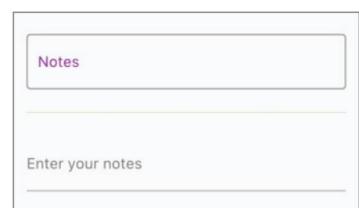


FIGURA 6.18: InputDecoration con OutlineInputBorder y borde por defecto

```
keyboardType: TextInputType.text, estilo:  
TextStyle( color:  
    Colors.grey.shade800, fontSize: 16.0, ),  
decoration:  
  
InputDecoration( labelText: "Notes",  
    labelStyle: TextStyle(color:  
        Colors.purple), //borde: UnderlineInputBorder (), borde:  
    EsquemaInputBorder(),  
  
,  
  
campo de formulario de texto  
campoFormularioTexto(  
    decoración: EntradaDecoración(  
        labelText: 'Ingrese sus notas', ), ),
```

PRUÉBALO Continuación del proyecto de imágenes agregando decoradores

Aún editando el archivo `home.dart`, agregará las clases de widgets `BoxDecoratorWidget()` e `InputDecoratorsWidget()`.

1. Agregue los nombres de clase de widget BoxDecoratorWidget() y InputDecoratorsWidget() después de Clase de widget ImagesAndIconWidget() . Agregue un widget Divider() entre cada nombre de clase de widget.

```
    cuerpo: SafeArea(  
        hijo: SingleChildScrollView(  
            ni: relleno (  
                relleno: EdgeInsets.all(16.0), child:  
                    Column(  
                        children:  
                            <Widget>[ const  
                                ImagesAndIconWidget(), Divider(), const  
  
                                BoxDecoratorWidget(), Divider(), const  
  
                                InputDecoratorsWidget(), ],  
  
                            ),  
                            ),  
                            ),  
                            ),  
                ),  
            ),  
        ),  
    ),
```

2. Agregue la clase de widget `BoxDecoratorWidget()` después de la clase de widget `ImagesAndIconWidget()`. La clase de widget devuelve un widget de relleno con el widget de contenedor como elemento secundario. La propiedad de decoración `Container` usa la clase `BoxDecoration`. Usando las propiedades `BoxDecoration borderRadius, color y boxShadow`, crea una forma de botón redondeada como la de la Figura 6.17.

```
clase BoxDecoratorWidget extiende StatelessWidget {
```

```
Clave
clave, }): super(clave: clave);

@anular
Compilación del widget (contexto BuildContext) { return
Padding (
    relleno: Edgelsets.all (16.0), niño: Contenedor
    (altura: 100.0, ancho:
    100.0, decoración:
    BoxDecoration (

        borderRadius: BorderRadius.all(Radius.circular(20.0)), color: Colors.orange,
        boxShadow: [ BoxShadow( color:
        Colors.grey,
        blurRadius:
        10.0, offset: Offset(0.0, 10.0),

    )], ), ), );
}

} }
```

3. Agregue la clase de widget InputDecoratorsWidget() después de la clase de widget BoxDecorationWidget()

Tomas un TextField y usas TextStyle para cambiar las propiedades de color y fontSize . La clase InputDecoration se usa para establecer los valores de labelText, labelStyle, border y enableBorder para personalizar las propiedades del borde. Estoy usando el OutlineInputBorder aquí, pero también podría usar la clase UnderlineInputBorder en su lugar. Dejé el borde UnderlineInputBorder y enableBorder OutlineInputBorder() comentado, lo que le permite probar ambas clases.

El siguiente código agrega dos widgets TextField personalizados con dos decoraciones diferentes. El primer TextField personaliza diferentes propiedades de InputDecoration para mostrar una etiqueta de notas moradas con el OutlineInputBorder(). El segundo widget TextField usa la decoración sin personalizar la propiedad del borde.

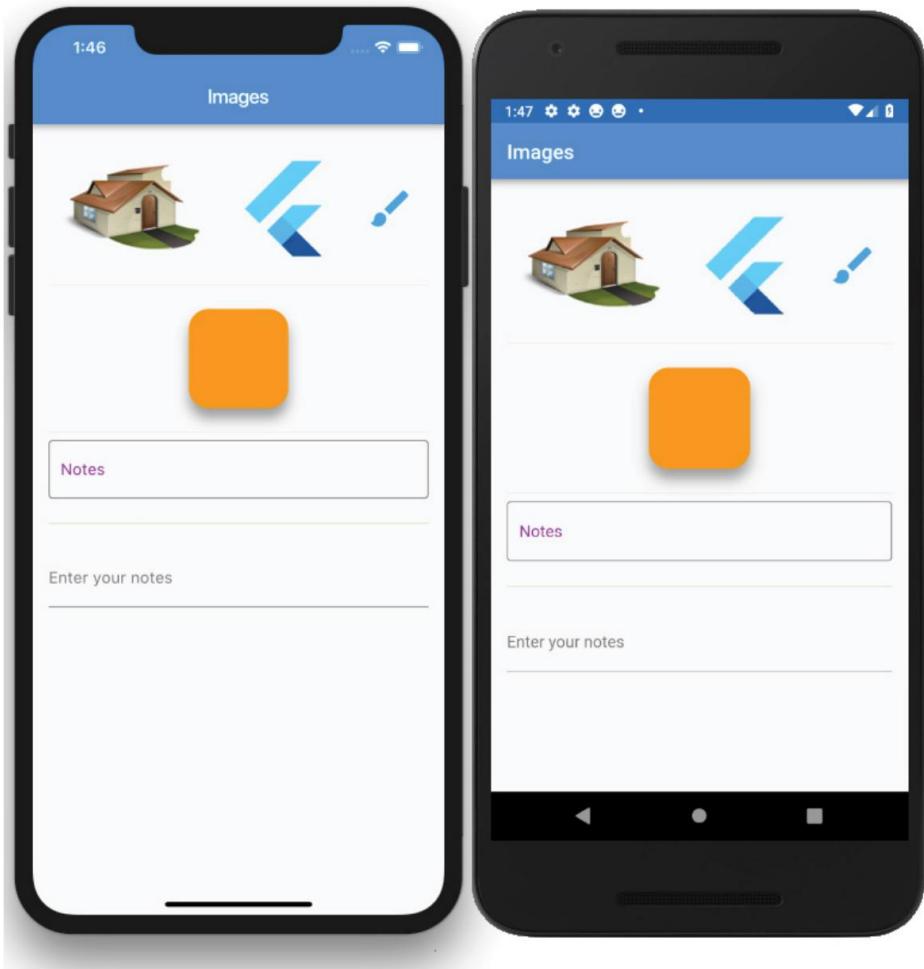
```
class InputDecoratorsWidget extiende StatelessWidget {
    const InputDecoratorsWidget({
        Clave
        clave, }): super(clave: clave);

    @anular
    Compilación del widget (contexto BuildContext)
    { columna de retorno
        (hijos: <Widget>[ TextField
            (tipo de
                teclado: TextInputType.text, estilo: TextStyle
                (color: Colors.grey.shade800,
                fontSize: 16.0,
            ),
            decoración: EntradaDecoración(
```

```
labelText: "Notas",
labelStyle: TextStyle(color: Colors.purple), //border:
UnderlineInputBorder(), //enabledBorder:
OutlineInputBorder(borderSide: BorderSide(color:
Colors.purple)), borde:
OutlineInputBorder(), ), ), Divider( color:

Colors.lightGreen, height: 50.0, ),
TextField(
decoration: InputDecoration(labelText: 'Ingrese sus notas'), ), ], );}

}
```



Cómo funciona

Los decoradores son invalables para mejorar la apariencia de los widgets. BoxDecoration proporciona muchas formas de dibujar un cuadro con borde, cuerpo y boxShadow . InputDecoration se usa en un campo de texto o en un TextFormField. No solo permite la personalización del borde, la etiqueta, el ícono y los estilos, sino que también brinda a los usuarios comentarios sobre la entrada de datos con sugerencias, errores, íconos y más.

USO DEL WIDGET DE FORMULARIO PARA VALIDAR CAMPOS DE TEXTO

Hay diferentes formas de usar widgets de campo de texto para recuperar, validar y manipular datos. El widget de formulario es opcional, pero los beneficios de usar un widget de formulario son validar cada campo de texto como un grupo. Puede agrupar widgets de TextFormField para validarlos manual o automáticamente. El widget TextFormField envuelve un widget TextField para proporcionar validación cuando se incluye en un widget de formulario .

Si todos los campos de texto pasan el método de validación FormState , entonces devuelve verdadero. Si algún campo de texto contiene errores, muestra el mensaje de error apropiado para cada campo de texto y el método de validación de FormState devuelve falso. Este proceso le brinda la posibilidad de usar FormState para verificar si hay errores de validación en lugar de verificar cada campo de texto en busca de errores y no permitir la publicación de datos no válidos.

El widget de formulario necesita una clave única para identificarlo y se crea mediante GlobalKey. Este valor de GlobalKey es único en toda la aplicación.

En el siguiente ejemplo, creará un formulario con dos TextFormField (Figura 6.19) para ingresar un artículo y la cantidad a ordenar. Creará una clase de pedido para almacenar el artículo y la cantidad y completar el pedido una vez que pase la validación.

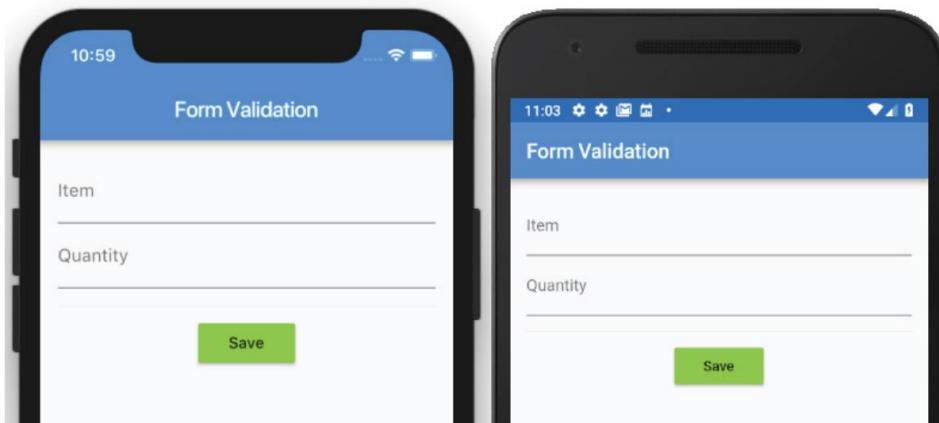


FIGURA 6.19: El diseño de Form y TextFormField

PRUÉBALO Creación de la aplicación de validación de formularios

Cree un nuevo proyecto de Flutter y asígnele el nombre ch6_form_validation. Puede seguir las instrucciones del Capítulo 4. Para este proyecto, necesita crear solo la carpeta de páginas . El objetivo de esta aplicación es mostrar cómo validar los valores de entrada de datos.

1. Abra el archivo home.dart y agregue al cuerpo un widget SafeArea con Column como elemento secundario. En los elementos secundarios de la columna , agregue el widget Form() , que modificó en el paso 7.

```
cuerpo: SafeArea(  
    niño: Columna( niños:  
        <Widget>[
```

```
        Forma(), ], ), ),
```

2. Cree la clase Order después de la clase _HomeState extends State<Home> {...}. La clase de orden mantendrá el artículo como un valor de cadena y la cantidad como un valor int .

```
class _HomeState extiende State<Home> {  
    //...  
}  
  
orden de clase {  
    Elemento de  
    cadena; cantidad int;  
}
```

3. Después de que la clase _HomeState extienda la declaración State<Home> y antes de @override, agregue las variables _formStateKey para el valor GlobalKey y _order para iniciar la clase Order .

Cree la clave única para el formulario mediante GlobalKey<FormState> y márquela como final , ya que no cambiará.

```
class _HomeState extiende State<Home> { final  
    GlobalKey<FormState> _formStateKey = GlobalKey<FormState>();  
  
    // Orden de Guardar  
    Pedido _pedido = Pedido();
```

4. Cree el método _validateItemRequired(valor de cadena) que acepta un valor de cadena . Utilice el operador ternario para verificar si el valor está establecido en está vacío y, en caso afirmativo, devuelva 'Elemento requerido'. De lo contrario, devuelve nulo.

```
Cadena _validateItemRequired(valor de cadena) {  
    valor de retorno. isEmpty ? 'Artículo requerido': nulo;  
}
```

5. Cree el método _validateItemCount(valor de cadena) que acepta un valor de cadena . Utilizar el operador ternario para convertir String a int. Luego verifique si int es mayor que cero; si no es así, devuelva 'Se requiere al menos un elemento'.

```
Cadena _validateItemCount (valor de cadena) {  
    // Comprobar si el valor no es nulo y convertirlo a entero
```

```

    int _valueAsInteger = value.isEmpty ? 0: int.tryParse(valor); devolver _valueAsInteger == 0?
    'Se requiere al menos un elemento': nulo;
}

```

6. Cree el método `_submitOrder()` llamado por el widget `FlatButton` para verificar si todos los campos de `TextField` pasan la validación y llame a `Form save()` para recopilar valores de todos los `TextFields` a la clase `Order`.

```

void _submitOrder() {
    if(_formStateKey.currentState.validate())
        { _formStateKey.currentState.save(); print('Artículo
            de pedido: ${order.item}'); print('Cantidad del pedido:
            ${pedido.cantidad}'); } }

```

7. Agregue al widget `Form()` una variable de clave privada llamada `_formStateKey`, establezca `autovalidate` en `true`, agregue `Padding` para la propiedad secundaria y agregue `Column` como elemento secundario de `Padding`.

Establecer `autovalidate` en verdadero permite que el widget `Form()` verifique la validación de todos los campos a medida que el usuario ingresa información y muestra un mensaje apropiado. Si la validación automática se establece en falso, no se realiza ninguna validación hasta que se llama manualmente al método `_formStateKey.currentState.validate()`.

```

Formulario( clave:
    _formStateKey,
    autovalidate: true, child: Padding(
        relleno: EdgeInsets.all(16.0), child:
        Column( children:
            <Widget>[
                ],
            ),
            ),
        ),
),

```

8. Agregue dos widgets `TextField` a la lista Columna secundaria de `Widget`. La primera `TextField` es una descripción de artículo, y el segundo `TextField` es una cantidad de artículos para ordenar.

9. Agregue una clase `InputDecoration` con `hintText` y `labelText` para cada `TextField`.

```

hintText: 'Café expreso',
labelText: 'Artículo',

```

10. Agregue una llamada para los métodos `validator` y `onSaved`. Se llama al método `validator` para validar los caracteres a medida que se ingresan, y el método `Form save()` llama al método `onSaved` para recopilar valores de cada `TextField`.

Para el validador, pase el valor ingresado en el widget `TextField` nombrando el valor de la variable entre paréntesis y use la sintaxis de flecha gruesa (`=>`) para llamar al método `_validateItemRequired(value)`. La sintaxis de la flecha gruesa es una abreviatura de `{ return mycustomexpression; }`.

```

validador: (valor) => _validateItemRequired(valor),

```

Tenga en cuenta que en el paso 2 creó una clase de pedido para contener los valores del artículo y la cantidad que recopilarán los métodos onSaved . Cuando se llama al método Form save() , se llama a todos los métodos TextField Form onSaved y los valores se recopilan en la clase Order , como order. elemento = valor.

onSaved: (valor) => order.item = valor,

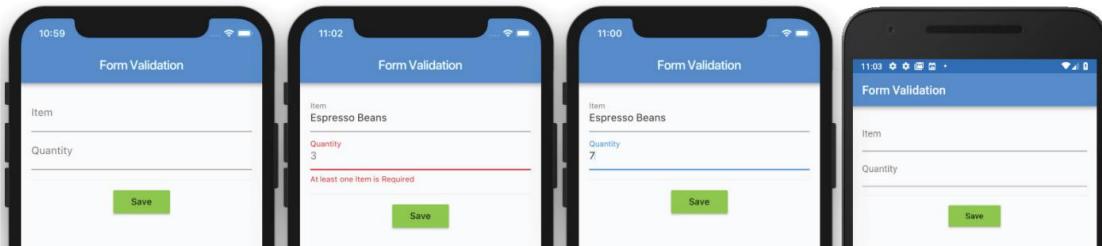
El siguiente código muestra ambos TextFormFieldFields:

```
CampoFormularioTexto(  
    decoración: InputDecoration( hintText:  
        'Espresso', labelText: 'Item', ),  
    validador: (valor) =>  
  
        _validateItemRequired(valor), onSaved: (valor) => order.item = valor, ),  
    TextFormField(  
  
        decoration: EntradaDecoración(  
            hintText: '3',  
            labelText: 'Cantidad', ), validador:  
  
        (valor) => _validateItemCount(valor), onSaved: (valor) => order.quantity  
        = int.tryParse(value), ),
```

Observe que usa int.tryParse() para convertir el valor de cantidad de String a int.

11. Agregue un Divider y un RaisedButton después del último TextFormField. Para onPressed, llame al Método _submitOrder() creado en el paso 6.

```
Divisor (altura: 32.0,), RaisedButton  
(hijo: Texto  
    ('Guardar'), color:  
    Colors.lightGreen, onPressed: () =>  
        _submitOrder(), ),
```



Cómo funciona

Al recuperar datos de los campos de entrada, el widget de formulario es una ayuda increíble y usó la clase `GlobalKey` para asignar una clave única para identificarlo. Use el widget de formulario para agrupar los widgets de `TextField` para validarlos manual o automáticamente. El método de validación de `FormState` valida los datos y, si pasa, devuelve verdadero. Si el método de validación de `FormState` falla, devuelve falso y cada campo de texto muestra el mensaje de error correspondiente. Cada propiedad del validador `TextField` tiene un método para verificar el valor apropiado. Cada propiedad `TextField` `onSaved` pasa el valor ingresado actualmente a la clase `Order`. En una aplicación del mundo real, tomaría los valores de la clase `Order` y los guardaría en una base de datos localmente o en un servidor web. En los Capítulos 14, 15 y 16, aprenderá cómo implementar Cloud Firestore para acceder a datos desde un servidor web.

COMPROBACIÓN DE LA ORIENTACIÓN

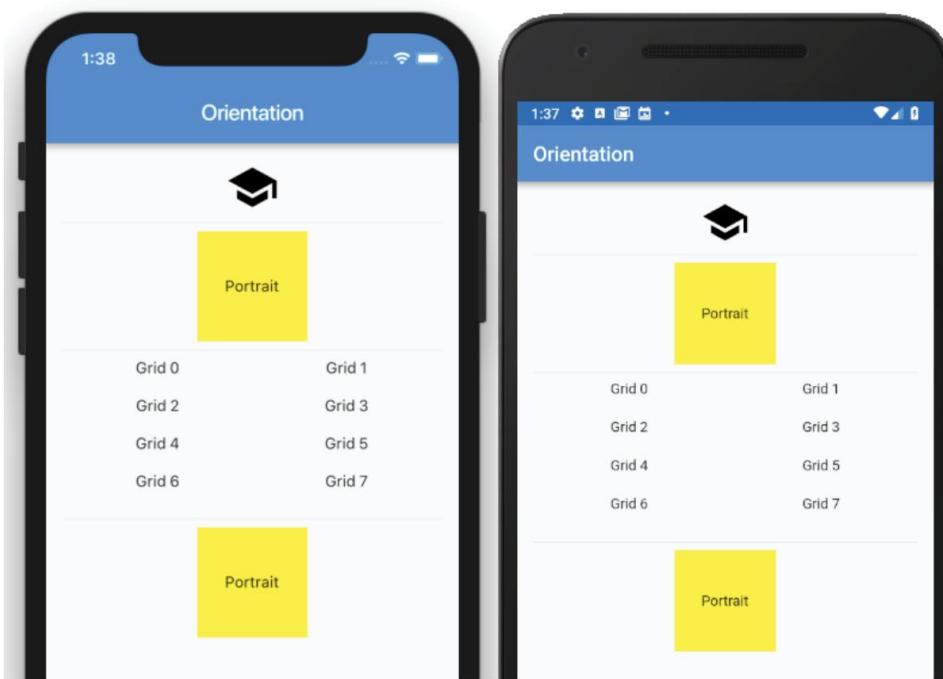
En ciertos escenarios, conocer la orientación del dispositivo ayuda a diseñar la interfaz de usuario adecuada. Hay dos formas de averiguar la orientación, `MediaQuery.of(context).orientation` y `OrientationBuilder`.

Una gran nota sobre `OrientationBuilder`: devuelve la cantidad de espacio disponible para que el padre determine la orientación. Esto significa que no garantiza la orientación real del dispositivo. Prefiero usar `MediaQuery` para obtener la orientación real del dispositivo debido a su precisión.

PRUÉBALO Creación de la aplicación de orientación

Cree un nuevo proyecto de Flutter y asígnele el nombre `ch6_orientation`. Puede seguir las instrucciones del Capítulo 4. Para este proyecto, solo necesita crear la carpeta de páginas .

En este ejemplo, el diseño de la interfaz de usuario cambiará según la orientación. Cuando el dispositivo está en modo vertical, mostrará un ícono, y cuando esté en modo horizontal, mostrará dos íconos. Echará un vistazo a un widget de contenedor que crecerá en tamaño y cambiará de color, y usará un widget de `GridView` para mostrar dos o cuatro columnas. Por último, agregué el widget `OrientationBuilder` para mostrar que cuando `OrientationBuilder` no es un widget principal, la orientación correcta no se calcula correctamente. Pero si coloca `OrientationBuilder` como padre, funciona correctamente; tenga en cuenta que el uso de `SafeArea` no afecta el resultado. La siguiente imagen muestra el proyecto final.



1. Abra el archivo home.dart y agregue al cuerpo un SafeArea con SingleChildScrollView como niño. Agregue Padding como elemento secundario de SingleChildScrollView. Agregue una columna como elemento secundario del relleno. En la propiedad Columna secundaria , agregue la clase de widget denominada OrientationLayoutIconsWidget(), que creará a continuación. Asegúrese de agregar la palabra clave const antes del nombre de la clase del widget para aprovechar el almacenamiento en caché para mejorar el rendimiento.

```
cuerpo: SafeArea(  
    hijo: SingleChildScrollView(  
        niño: relleno (  
            relleno: EdgeInsets.all(16.0), child:  
                Column( children:  
                    <Widget>[  
                        const OrientationLayoutIconsWidget(), ],  
  
                    ),  
                    ),  
                    ),  
                    ),  
                    ),
```

2. Agregue la clase de widget OrientationLayoutIconsWidget() después de que la clase Inicio se extienda Widget sin estado {...}. La primera variable para inicializar es la orientación actual llamando a MediaQuery.of() después de Widget build (contexto BuildContext).

```
class OrientationLayoutIconsWidget extiende StatelessWidget { const  
    OrientationLayoutIconsWidget({  
        Clave  
        clave, }) : super(clave: clave);
```

```
        @anular
        Creación de widgets (contexto BuildContext)
        { Orientación _orientación = MediaQuery.of(contexto).orientación; devolver Contenedor();
        });

    }
```

3. En función de la Orientación actual , devuelve un diseño diferente de los widgets de iconos . Use un operador ternario para verificar si la Orientación es vertical y, de ser así, devuelva un solo ícono de Fila . Si la orientación es horizontal, devuelve una fila de dos widgets de iconos . Reemplace el contenedor de retorno actual () con el siguiente código:

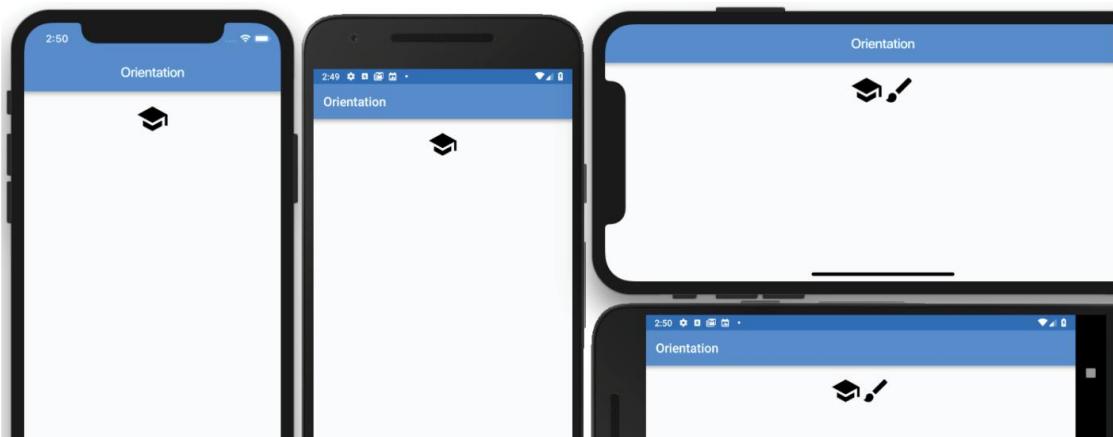
```
return _orientación == Orientación retrato? Row( mainAxisAlignment:
    MainAxisAlignment.center, children: <Widget>[ Icon( Icons.school,
    size: 48.0, ), ],
    );
    :
    Row( mainAxisAlignment: MainAxisAlignment.center, children:
    <Widget>[ Icon( Icons.school,
    size:
    48.0, ),
    Icon( Icons.brush,
    size:
    48.0, ), ],
    );
    );
```

4. Juntando todo el código, obtienes lo siguiente:

```
class OrientationLayoutIconsWidget extiende StatelessWidget { const
    OrientationLayoutIconsWidget({
        Clave
        clave, }) : super(clave: clave);

        @anular
        Creación de widgets (contexto BuildContext)
        { Orientación _orientación = MediaQuery.of(contexto).orientación; return _orientación ==
        Orientación retrato? Fila(
            mainAxisAlignment: MainAxisAlignment.center, children:
            <Widget>[ Icon( Icons.school,
            size:
            48.0, ), ],
            );
            );
```

```
)  
    : Fila(  
mainAxisAlignment: MainAxisAlignment.center, children:  
<Widget>[ Icon( Icons.school,  
size:  
    48.0,),  
Icon( Icons.brush,  
  
size:  
    48.0, ), ], );  
  
}  
}
```



- Después de OrientationLayoutIconsWidget(), agregue un widget Divider y la clase de widget OrientationLayoutWidget() para crear.

Los pasos son similares a los anteriores, pero en lugar de usar filas e íconos, está usando contenedores: obtenga el modo Orientación y, para el retrato, devuelva un widget Contenedor amarillo con un ancho de 100,0 píxeles. Cuando se gira el dispositivo, el paisaje devuelve un widget de contenedor verde con un ancho de 200,0 píxeles.

```
class OrientationLayoutWidget extiende StatelessWidget {  
  const OrientationLayoutWidget({  
    Clave  
    clave, }) : super(clave: clave);  
  
  @anular  
  Creación de widgets (contexto BuildContext) { Orientación  
    _orientación = MediaQuery.of(contexto).orientación;  
  
    return _orientación == Orientación.retrato? Contenedor (alineación:  
      Alignment.center,
```

```
        color: Colores.amarillo,
        altura: 100.0,
        ancho: 100.0,
        hijo: Texto('Retrato'),
    )
    : Contenedor
    (alineación: Alignment.center, color:
    Colors.lightGreen, height: 100.0,
    width: 200.0, child:
    Text('Landscape'), );

}
```

6. Después de OrientationLayoutWidget(), agregue un widget Divider y la clase de widget GridViewWidget() que creará.

Aunque observará más de cerca el widget GridView en el Capítulo 9, es apropiado usarlo ahora ya que es el ejemplo más cercano a la vida real. En modo vertical, el widget GridView muestra dos columnas y en modo horizontal, muestra cuatro columnas.

Hay algunos elementos a tener en cuenta aquí. Dado que el widget de GridView está dentro de un widget de columna , establezca el argumento de reducción del constructor de GridView.count en verdadero o romperá las restricciones. También establecí el argumento de la física en NeverScrollableScrollPhysics() o GridView desplazará a sus elementos secundarios desde dentro. Recuerde, tiene todos estos widgets dentro de un SingleChildScrollView.

```
class GridViewWidget extiende StatelessWidget { const
    GridViewWidget({
        Clave
        clave, }) : super(clave: clave),

    @anular
    Creación de widgets (contexto BuildContext)
    { Orientación _orientación = MediaQuery.of(contexto).orientación;

    return GridView.count(shrinkWrap:
        true, physics:
        NeverScrollableScrollPhysics(), crossAxisCount:
        _orientación == Orientation.portrait ? 2 : 4, childAspectRatio: 5.0, children:
        List.generate(8, (int index)
        { return Text("Grid $índice", textAlign:
        TextAlign.center,); }), );
```

7. Después de GridViewWidget(), agregue un widget Divider y la clase de widget OrientationBuilderWidget() que creará.

Como se mencionó anteriormente, uso MediaQuery.of() para obtener orientación porque es más preciso, pero es bueno saber cómo usar OrientationBuilder.

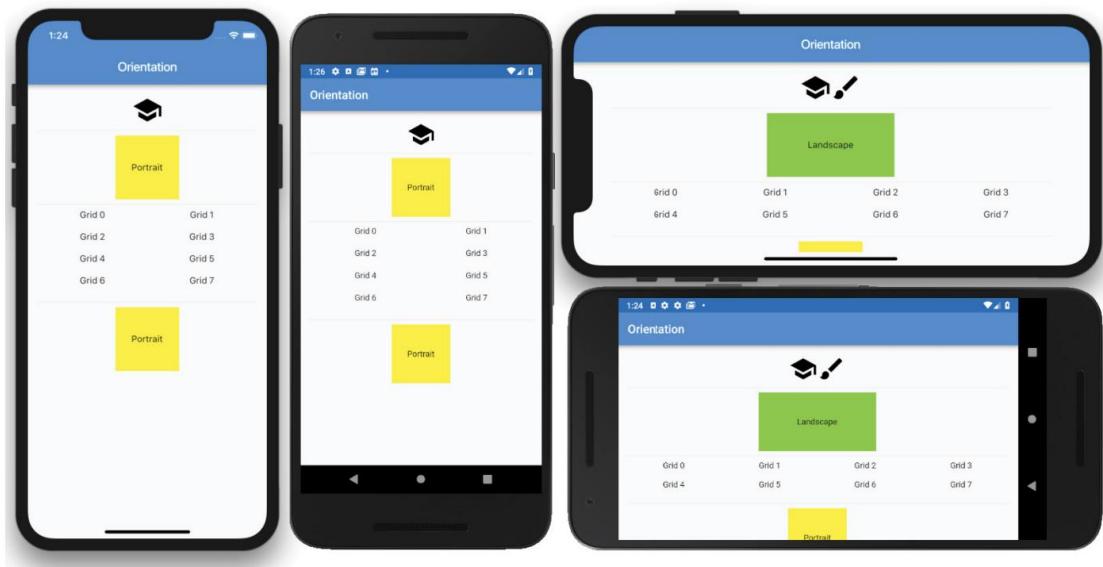
OrientationBuilder requiere que se pase una propiedad de constructor y no puede ser nulo. La propiedad del constructor toma dos parámetros: BuildContext y Orientation.

constructor: (contexto BuildContext, orientación Orientación) {}

Los pasos y el resultado son los mismos que con `_buildOrientationLayout()`. Use el operador ternario para verificar la orientación y el retrato, y devuelva un widget de contenedor amarillo con un ancho de 100.0 píxeles. Cuando se gira el dispositivo, el paisaje devuelve un widget de contenedor verde con un ancho de 200.0 píxeles.

Tenga en cuenta que OrientationBuilder corre el riesgo de no detectar correctamente el modo de orientación porque es un widget secundario y depende del tamaño de la pantalla principal en lugar de la orientación del dispositivo. Debido a esto, recomiendo usar MediaQuery.of() en su lugar.

```
// OrientationBuilder como hijo no proporciona la Orientación correcta. es decir, Hijo de la Columna...  
  
// OrientationBuilder como padre proporciona la clase de orientación correcta  
OrientationBuilderWidget extends StatelessWidget {  
    const OrientationBuilderWidget({  
        Clave  
        clave, }) : super(clave: clave);  
  
    @anular  
    Compilación del widget (contexto BuildContext)  
    { return OrientationBuilder (  
        constructor: (contexto BuildContext, orientación de Orientación) { orientación  
        de retorno == Orientación retrato ? Contenedor  
        (alineación:  
            Alignment.center, color: Colors.yellow,  
            height: 100.0, width: 100.0,  
            child: Text('Portrait'),  
  
        )  
        : Contenedor  
        (alineación: Alignment.center, color:  
            Colors.lightGreen, height: 100.0,  
            width: 200.0, child:  
  
            Text('Landscape'), ); }, );  
    }  
}
```



Cómo funciona

Puede detectar la orientación del dispositivo llamando a `MediaQuery.of(context).orientation`, que devuelve un valor vertical u horizontal. También está `OrientationBuilder`, que devuelve la cantidad de espacio disponible para que el padre determine la orientación. Recomiendo usar `MediaQuery` para recuperar la orientación correcta del dispositivo.

RESUMEN

En este capítulo, aprendió sobre los widgets (básicos) más utilizados. Estos widgets básicos son los componentes básicos para diseñar aplicaciones móviles. También exploraste diferentes tipos de botones para elegir según la situación. Aprendió a agregar activos a su aplicación a través de `AssetBundle` enumerando elementos en el archivo `pubspec.yaml`. Usó el widget `Imagen` para cargar imágenes desde el dispositivo local o un servidor web a través de una cadena de URL. Viste cómo el widget `Icon` te da la posibilidad de cargar iconos usando la biblioteca de fuentes `MaterialIcons`.

Para modificar la apariencia de los widgets, aprendiste a usar `BoxDecoration`. Para mejorar la retroalimentación de los usuarios sobre la entrada de datos, implementó `InputDecoration`. La validación de múltiples entradas de datos de campos de texto puede ser engorrosa, pero puede usar el widget de formulario para validarlas manual o automáticamente. Por último, el uso de `MediaQuery` para averiguar la orientación actual del dispositivo es extremadamente poderoso en cualquier aplicación móvil para diseñar widgets según la orientación.

En el próximo capítulo, aprenderá a usar animaciones. Comenzará usando widgets como `Animat` `edContainer`, `AnimatedCrossFade` y `AnimatedOpacity` y terminará con el poderoso `AnimationController` para una animación personalizada.

LO QUE APRENDISTE EN ESTE CAPÍTULO

TEMA	CONCEPTOS CLAVE
Uso de widgets básicos	Aprendió a usar Scaffold, SafeArea, AppBar, Container, Text, RichText, Column, Row, Column and Row Nesting, Buttons, FloatingActionButton, FlatButton, RaisedButton, IconButton, PopupMenuButton y ButtonBar.
Uso de imágenes	Aprendiste a usar AssetBundle, Image e Icon.
Usando decoradores	Aprendió a usar Decoración, Decoración de caja y Decoración de entrada.
Uso de formularios para el campo de texto validación	Aprendió a usar el widget de formulario para validar cada TextFormField como un grupo.
Detección de orientación	Aprendió a usar MediaQuery.of(context).orientation y OrientationBuilder para detectar la orientación del dispositivo.

7

Agregar animación a una aplicación

LO QUE APRENDERÁS EN ESTE CAPÍTULO

Cómo usar AnimatedContainer para cambiar gradualmente los valores con el tiempo

Cómo usar AnimatedCrossFade para realizar un fundido cruzado entre dos widgets secundarios

Cómo usar AnimatedOpacity para mostrar u ocultar la visibilidad del widget mediante el desvanecimiento animado con el tiempo

Cómo utilizar AnimationController para crear animaciones personalizadas

Cómo usar AnimationController para controlar animaciones escalonadas

En este capítulo, aprenderá cómo agregar animación a una aplicación para transmitir acción, lo que puede mejorar la experiencia del usuario (UX) si se usa correctamente. Demasiadas animaciones sin transmitir la acción adecuada pueden empeorar la UX. Flutter tiene dos tipos de animación: basada en la física y Tween. Este capítulo se centrará en las animaciones Tween.

La animación basada en la física se utiliza para imitar el comportamiento del mundo real. Por ejemplo, cuando se deja caer un objeto y golpea el suelo, rebota y continúa avanzando, pero con cada rebote, continúa disminuyendo la velocidad con rebotes más pequeños y eventualmente se detiene. A medida que el objeto se acerca al suelo con cada rebote, la velocidad aumenta, pero la altura del rebote disminuye.

Tween es la abreviatura de "en el medio", lo que significa que la animación tiene puntos de inicio y finalización, una línea de tiempo y una curva que especifica el tiempo y la velocidad de la transición. La belleza es que el marco calcula automáticamente la transición desde el principio hasta el punto final.

USO DE UN CONTENEDOR ANIMADO

Comencemos con una animación simple utilizando el widget `AnimatedContainer`. Este es un widget de contenedor que cambia gradualmente los valores durante un período de tiempo. El constructor `AnimatedContainer` tiene argumentos llamados duración, curva, color, alto, ancho, hijo, decoración, transformación y muchos otros.

PRUÉBALO Creación de la aplicación `AnimatedContainer`

En este proyecto, animará el ancho de un widget de contenedor tocándolo. Por ejemplo, podría usar este tipo de animación para animar un gráfico de barras horizontales.

1. Cree un nuevo proyecto de Flutter y asígnele el nombre `ch7_animations`. Puede seguir las instrucciones del Capítulo 4, "Creación de una plantilla de proyecto inicial". Para este proyecto, debe crear solo las carpetas de páginas y widgets .
2. Cree un nuevo archivo Dart en la carpeta de `widgets` . Haga clic con el botón derecho en la carpeta de `widgets` , seleccione Nuevo Archivo Dart, ingrese `animado_contenedor.dart` y haga clic en el botón Aceptar para guardar.
3. Importe la biblioteca `material.dart` , agregue una nueva línea y luego comience a escribir `st`. Se abre la ayuda de autocompletado. Seleccione la abreviatura `stf` y asígnele el nombre `AnimatedContainerWidget`.
4. Después de que la clase `_AnimatedContainerWidgetState` extienda `State<AnimatedContainerWidget>` y antes de `@override`, agregue las variables `_height` y `_width` y el método `_increaseWidth()` .

Las variables `_alto` y `_ancho` son de tipo doble.

```
doble _altura = 100.0;
```

El método `_increaseWidth()` llama al método `setState()` para notificar al marco que el valor de `_width` ha cambiado y programa una compilación para el estado de este objeto para volver a dibujar el subárbol. Si no llama a `setState()`, el valor de `_width` aún cambia, pero el widget `AnimatedContainer` no se volverá a dibujar con el nuevo valor.

```
class _AnimatedContainerWidgetState extiende State<AnimatedContainerWidget> { double _height =  
100.0; doble _ancho = 100.0;  
  
_increaseWidth()  
{ setState()  
{ _width = _width >= 320.0 ? 100.0 : _width += 50.0; };}  
}  
  
@anular  
Compilación del widget (contexto BuildContext) {
```

Cuando se carga la página, las variables `_height` y `_width` se inician con valores de 100,0 píxeles. Cuando se toca `FlatButton` , la propiedad `onPressed` llama al método `_increaseWidth()` .

Aquí utilizará 320,0 píxeles como el ancho máximo permitido, pero este podría haber sido el ancho del dispositivo. Con cada evento de toque, aumenta el ancho actual en 50,0 píxeles a partir de 100,0 píxeles. A medida que aumenta el ancho, una vez que supera los 320,0 píxeles, restablece el tamaño a 100,0 píxeles. Para calcular el nuevo `_width` de `AnimatedContainer`, use el operador ternario. Si `_width` es mayor o igual a 320,0 píxeles, establezca `_width` en los 100,0 píxeles originales. Esto animará `AnimatedContainer` de vuelta al tamaño original. De lo contrario, tome el valor actual de `_width` y agregue 50,0 píxeles. Tenga en cuenta que los signos más e igual (`+ =`) se utilizan para tomar el valor actual y sumarle.

Tenga en cuenta que los valores de alto y ancho son variables privadas, `_alto` y `_ancho`, ya que los está guiando con el símbolo de subrayado.

El elemento secundario `FlatButton` es la etiqueta que muestra el mensaje "Toque para aumentar el ancho". Estoy usando `\n` para continuar el texto en la siguiente línea y usando el signo `$` para pasar el valor de `_width`.

Una vez que se presiona `FlatButton`, agrega una llamada al método `_increaseWidth()` en la propiedad `onPressed`. Ignore las líneas onduladas rojas del editor de código, ya que aún no ha creado las variables y el método.

El argumento de duración toma un objeto `Duration()` para especificar el tipo de tiempo a usar, como microsegundos, milisegundos, segundos, minutos, horas y días.

duración: Duración (milisegundos: 500),

El argumento de la curva toma una clase `Curves` y usa `Curves.elasticOut`. Algunos de los tipos de curvas disponibles son rebote hacia adentro, rebote hacia afuera, rebote hacia afuera, entrada suave, entrada suave, salida suave , entrada elástica, entrada elástica y salida elástica.

curva: `Curvas.elasticOut`,

Aquí está la clase de widget `_AnimatedContainerWidget()` completa :

```
importar 'paquete: flutter/material.dart';

class AnimatedContainerWidget extiende StatefulWidget {
  const AnimatedContainerWidget({
    Clave
    clave, }) : super(clave: clave);

  @anular
  _AnimatedContainerWidgetState createState() => _AnimatedContainerWidgetState();
}

class _AnimatedContainerWidgetState extiende State<AnimatedContainerWidget> { double _height = 100.0; doble
  _ancho = 100.0;

  void _aumentarAncho()
  { establecerEstado(() {
      _ancho = _ancho >= 320.0 ? 100.0 : _ancho += 50.0; });

  }
```

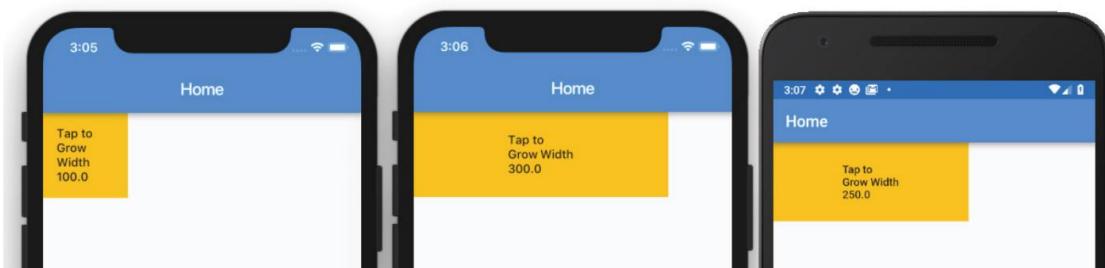
```
@anular
Compilación del widget (contexto BuildContext) {
    return Fila( niños:
        <Widget>[
            Contenedor animado(
                duración: Duración (milisegundos: 500), curva:
                    Curves.elasticOut, color: Colors.amber,
                height: _height, width: _width,
                child:FlatButton( child:
                    Text("Toque
para\Increcer
ancho\n$_ancho"), onPressed: () { _increaseWidth(); }, ), ], );
    }
}
```

5. Abra el archivo home.dart e importe el archivo animation_container.dart en la parte superior de la página.

```
importar 'paquete: flutter/material.dart'; import
'paquete:ch7_animations/widgets/animated_container.dart';
```

6. Agregue al cuerpo un widget SafeArea con Column como elemento secundario. En los hijos de la columna , agregue la llamada a la clase de widget AnimatedContainerWidget().

```
cuerpo: SafeArea( niño:
    Columna( niños:
        <Widget>[
            WidgetContenedorAnimado(), ], ), ),
```



Cómo funciona

El constructor `AnimatedContainer` toma un argumento de duración y usa la clase `Duration` para especificar 500 milisegundos. Los 500 milisegundos equivalen a medio segundo. El argumento de la curva le da a la animación un efecto de resorte usando `Curves.elasticOut`. El argumento `onPressed` llama al método `_increaseWidth()` para cambiar la variable `_width` dinámicamente. El método `setState()` notifica al marco Flutter que el estado interno del objeto cambió y hace que el marco programe una compilación para este objeto `State`. El widget `AnimatedContainer` se anima automáticamente entre el antiguo valor de `_width` y el nuevo valor de `_width`.

USO DE CROSSFADE ANIMADO

El widget `AnimatedCrossFade` proporciona un excelente fundido cruzado entre dos widgets secundarios. El constructor `AnimatedCrossFade` toma la duración, `firstChild`, `secondChild`, `crossFadeState`, `sizeCurve` y muchos otros argumentos.

PRUÉBELO Adición del widget AnimatedCrossFade

Este ejemplo crea un fundido cruzado entre colores y tamaño tocando el widget. El widget se desvanecerá cambiando el tamaño y el color de amarillo a verde.

1. Cree un nuevo archivo Dart en la carpeta de widgets . Haga clic con el botón derecho en la carpeta de widgets , seleccione Nuevo Archivo Dart, ingrese `animation_cross_fade.dart` y haga clic en el botón Aceptar para guardar.
2. Importe la biblioteca `material.dart` , agregue una nueva línea y comience a escribir `st`. Se abre la ayuda de autocompletado. Seleccione la abreviatura `sifl` y asígnele el nombre `AnimatedCrossFadeWidget`.
3. Después de que la clase `_AnimatedCrossFadeWidgetState` extienda `State<AnimatedCrossFadeWidget>` y antes de `@override`, agregue una variable para `_crossFadeStateShowFirst` y el método `_crossFade()` .

La variable `_crossFadeStateShowFirst` es de tipo booleano (bool).

```
bool _crossFadeStateShowFirst = verdadero;
```

El método `_crossFade()` llama a `setState()` para notificar al marco que el valor de `_crossFadeState ShowFirst` ha cambiado y programa una compilación para el estado de este objeto para volver a dibujar el subárbol. Si no llama a `setState()`, el valor de `_width` aún cambia, pero el widget `AnimatedCrossFade` no se volverá a dibujar con el nuevo valor.

```
class _AnimatedCrossFadeWidgetState extiende State<AnimatedCrossFadeWidget> { bool  
_crossFadeStateShowFirst = true;
```

```
void _crossFade()  
{ setState()  
  { _crossFadeStateShowFirst = _crossFadeStateShowFirst ? falso : verdadero; };  
  
}  
  
@anular  
Compilación del widget (contexto BuildContext) {
```

Cuando se carga la página, la variable `_crossFadeStateShowFirst` se inicia con un valor de verdadero. Cuando se toca `FlatButton`, la propiedad `onPressed` llama al método `_crossFade()`.

Para rastrear qué elemento secundario (`firstChild`, `secondChild`) debe mostrar y animar el widget `AnimatedCrossFade`, use el operador ternario. Si `_crossFadeStateShowFirst` es verdadero, establezca el valor de `_crossFadeStateShowFirst` en falso. De lo contrario, configúrelo en verdadero ya que el valor actual es falso.

4. Para mantener limpia la interfaz de usuario, agregue cada widget de animación en una fila separada. Incrustar el Widget `AnimatedCrossFade` en una fila con una pila como elemento secundario. La razón para usar una pila es agregar un `FlatButton` sobre el widget `AnimatedCrossFade` para darle una etiqueta y un evento `onPressed`. También podría usar un widget de `GestureDetector`.

```
@anular  
Compilación del widget (contexto BuildContext) {  
return  
Row( children:  
  
<Widget>[ Stack( alineación:  
  Alignment.center, children: <Widget>[  
    Fundido cruzado animado(  
      duración: Duración (milisegundos: 500), sizeCurve:  
      Curves.bounceOut, crossFadeState:  
      _crossFadeStateShowFirst ? CrossFadeState.showFirst :  
      CrossFadeState.showSecond,  
      firstChild: Container( color:  
        Colors.amber, height:  
        100.0, width:  
        100.0, ),  
  
      secondChild: Container( color:  
        Colors.lime, height: 200.0,  
        width: 200.0, ), ),  
      Positioned.fill( hijo:  
  
FlatButton(  
  child: Text('Toque para\nFade Color & Size'), onPressed:  
  () { _crossFade(); }, ), ), ],
```

```
    ), ], );  
}
```

5. Usa una duración de 500 milisegundos como en la animación anterior.

```
duración: Duración (milisegundos: 500),
```

6. Para sizeCurve, use Curves.bounceOut para ver cómo la clase Curves afecta las animaciones.

```
sizeCurve: Curvas.bounceOut,
```

7. Para decidir qué widget de contenedor mostrar cuando se complete la animación, configure la cruz

Valor del argumento FadeState utilizando el operador ternario para verificar si el valor `_crossFadeState ShowFirst` es verdadero; luego muestre `CrossFadeState.showFirst`; de lo contrario, muestre `CrossFadeState.showSecond`.

```
crossFadeState: _crossFadeStateShowFirst? CrossFadeState.showFirst : CrossFadeState.showSecond,
```

8. La animación se desvanecerá entre colores y tamaños; para `firstChild` y `secondChild`, use un Contenedor. El contenedor `firstChild` tiene una propiedad de color `Colors.amber` y valores de alto y ancho de 100,0 píxeles. El contenedor `secondChild` tiene una propiedad de color `Colors.lime` y valores de alto y ancho de 200,0 píxeles.

```
firstChild: Container( color:  
    Colors.amber, height: 100.0,  
    width: 100.0, ),  
secondChild:
```

```
Container( color: Colors.lime, height:  
200.0, width: 200.0, ),
```

9. Agregue el segundo elemento secundario Stack y llame al constructor `Positioned.fill` con un elemento secundario de Botón Plano. El uso de `Positioned.fill` permite que el widget `FlatButton` se redimensione al tamaño máximo del widget Stack . Para la propiedad `onPressed` , agregue una llamada al método `_cross Fade()` .

```
Positioned.fill (hijo:  
FlatButton ( child:  
Text('Toque para\nFade Color & Size'), onPressed: ()  
{ _crossFade(); }, ), ),
```

Aquí está la clase de widget `full_AnimatedCrossFadeWidget()` :

```
importar 'paquete: flutter/material.dart';  
  
clase AnimatedCrossFadeWidget extiende StatefulWidget {  
const AnimatedCrossFadeWidget({
```

```
Clave
clave, }) : super(clave: clave);

@anular
_AnimatedCrossFadeWidgetState createState() => _AnimatedCrossFadeWidgetState(); }

class _AnimatedCrossFadeWidgetState extiende State<AnimatedCrossFadeWidget> { bool _crossFadeStateShowFirst
= true;

void _crossFade() { setState(() {
    { _crossFadeStateShowFirst = _crossFadeStateShowFirst ? falso : verdadero; });
}

}

@anular
Compilación del widget (contexto BuildContext) {
    return
        Row( children:
            <Widget>[ Stack( alineación: Alignment.center,
                children: <Widget>[
                    Fundido cruzado animado(
                        duración: Duración (milisegundos: 500), sizeCurve:
                        Curves.bounceOut, crossFadeState:
                        _crossFadeStateShowFirst ? CrossFadeState.showFirst :
                        CrossFadeState.showSecond, firstChild:
                            Container( color: Colors.amber,
                                height: 100.0, width: 100.0, ),
                            secondChild:
                                Container( color:
                                    Colors.lime, height: 200.0, width:
                                    200.0, ), ), Positioned.fill( hijo:
                                    FlatButton(
                                        child: Text('Toque para\nFade Color & Size'), onPressed: () {
                                        { _crossFade(); }, ), ], ), );
                ],
            ],
        );
}
}}}
```

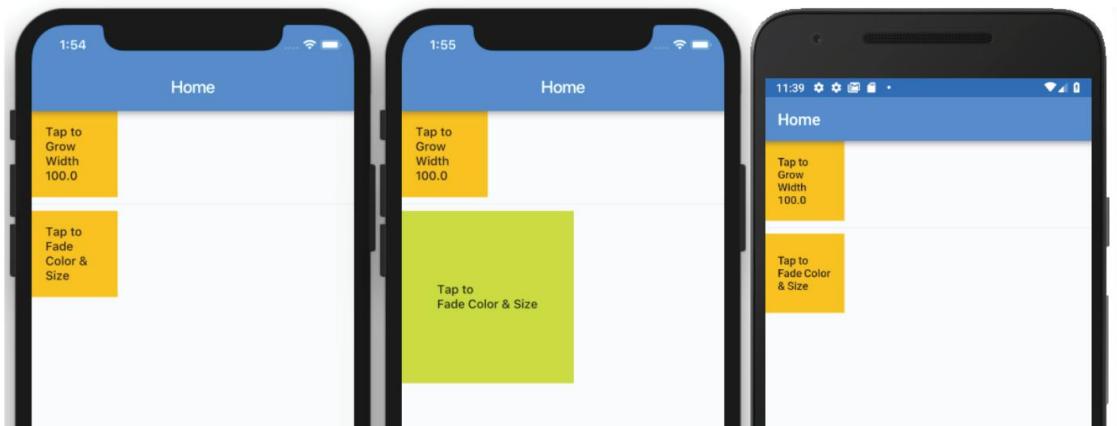
10. Continúe editando el archivo home.dart e importe el archivo animation_cross_fade.dart en la parte superior de la pagina

```
importar 'paquete: flutter/material.dart'; import  
'paquete:ch7_animations/widgets/animated_container.dart'; import 'paquete:ch7_animations/  
widgets/animated_cross_fade.dart';
```

11. Justo después de la clase de widget AnimatedContainerWidget() , agregue un widget Divider() . En el siguiente línea, agregue una llamada a la clase de widget AnimatedCrossFadeWidget().

```
cuerpo: SafeArea( niño:  
    Columna( niños:  
        <Widget>[  
            WidgetContenedorAnimado(),  
            Divisor(),  
            AnimatedCrossFadeWidget(), ], ), ),
```

Observe cuán limpio se ve el código y cómo mejora la legibilidad al separar cada sección principal del árbol de widgets. Más importante aún, solo se está reconstruyendo la clase de widget en ejecución, y las otras clases de widget de animación no, lo que le brinda un gran rendimiento.



Cómo funciona

El constructor AnimatedCrossFade toma un argumento de duración y usa la clase Duration para especificar 500 milisegundos. El argumento sizeCurve le da a la animación entre el tamaño de los dos niños un efecto de resorte usando Curves.bounceOut. El argumento crossFadeState establece que el widget secundario se muestre una vez que se complete la animación. Al usar la variable _crossFadeStateShowFirst , se muestra el elemento secundario crossFadeState correcto. Los argumentos firstChild y secondChild contienen los dos widgets para animar.

USO DE LA OPACIDAD ANIMADA

Si necesita ocultar u ocultar parcialmente un widget, `AnimatedOpacity` es una excelente manera de animar el desvanecimiento con el tiempo. El constructor `AnimatedOpacity` toma la duración, la opacidad, la curva y los argumentos secundarios . Para este ejemplo, no usa una curva; ya que desea un fundido de entrada y salida suave, no es necesario.

PRUÉBALO Agregando el Widget AnimatedOpacity

El ejemplo utiliza la opacidad para atenuar parcialmente un widget de contenedor . El widget animará el valor de opacidad desde totalmente visible hasta casi difuminado.

1. Cree un nuevo archivo Dart en la carpeta de widgets . Haga clic con el botón derecho en la carpeta de widgets , seleccione Nuevo Archivo Dart, ingrese opacity_animada.dart y haga clic en el botón Aceptar para guardar.
2. Importe la biblioteca material.dart , agregue una nueva línea y luego comience a escribir st. Se abre la ayuda de autocompletado. Seleccione la abreviatura stful y asignele el nombre `AnimatedOpacityWidget`.
3. Después de que la clase `_AnimatedOpacityWidgetState` extienda `State<AnimatedOpacityWidget>` y antes de `@override`, agregue la variable para `_opacity` y el método `_animatedOpacity()` .

La variable `_opacity` es de tipo doble.

```
doble _opacidad = 1.0;
```

El método `_animatedOpacity()` llama a `setState()` para notificar al marco que el valor de `_opacity` ha cambiado y programa una compilación para el estado de este objeto para volver a dibujar el subárbol. Si no llama a `setState()`, el valor de `_opacity` aún cambia, pero el widget `AnimatedOpacity` no se vuelve a dibujar con el nuevo valor.

```
class _AnimatedOpacityWidgetState extiende State<AnimatedOpacityWidget> { double _opacity  
= 1.0;  
  
void _animatedOpacity()  
{ setState()  
{ _opacity = _opacity == 1.0 ? 0.3 : 1.0; };  
  
}  
  
@anular  
Compilación del widget (contexto BuildContext) {
```

Cuando se carga la página, la variable `_opacity` se inicia con un valor de 1,0, que es totalmente visible. Cuando se toca el widget `FlatButton` , la propiedad `onPressed` llama al método `_animatedOpac ity()` .

Para calcular la opacidad del widget, use el operador ternario. Si el valor de `_opacity` es 1,0, establezca `_opacity` en 0,3. De lo contrario, configúrelo en 1,0 ya que el valor actual es 0,3.

4. Para mantener limpia la interfaz de usuario, agregue cada widget de animación en una fila separada. Incrustar la opacidad animada widget en una fila.

```
@anular
Compilación del widget (contexto BuildContext) {
    return Fila( niños:
        <Widget>[ Opacidad
            Animada( duración:
                Duración(milisegundos: 500), opacidad: _opacidad, hijo:
                    Contenedor( color:
                        Colores.ámbar, altura:
                            100.0, ancho: 100.0, hijo:
                                FlatButton( hijo:
                                    Texto( 'Toque para
                                        desvanecerse'), onPressed:
                                        () { _animatedOpacity(); }, ), ), ], );
    }
}
```

5. Usa una duración de 500 milisegundos como en las animaciones anteriores.

duración: Duración (milisegundos: 500),

6. La animación anima el valor de opacidad del widget secundario AnimatedOpacity , que es un Contenedor en este caso. Según el valor de opacidad, el widget Contenedor se desvanece o se desvanece. Un valor de opacidad de 1,0 es completamente visible y un valor de opacidad de 0,0 es invisible. Vas a animar la variable _opacity de 1.0 a 0.3 y viceversa.

opacidad: _opacidad,

7. Agregue un widget Contenedor como elemento secundario del widget AnimatedOpacity . Agregue un widget FlatButton como elemento secundario del widget Container con onPressed llamando al método _animatedOpacity() .

```
child: Container( color:
    Colors.amber, height: 100.0,
    width: 100.0, child:
        FlatButton( child:
            Text('Tap to Fade'),
            onPressed: () { _animatedOpacity(); }, ), ),
```

Aquí está la clase de widget _buildAnimatedOpacity() completa :

```
importar 'paquete: flutter/material.dart';

class AnimatedOpacityWidget extiende StatefulWidget {
    const AnimatedOpacityWidget({
        Clave
        clave, }) : super(clave: clave);

    @anular
    _AnimatedOpacityWidget createState() => _AnimatedOpacityWidgetState();
}

class _AnimatedOpacityWidgetState extiende State<AnimatedOpacityWidget> { double _opacity = 1.0;

void _animatedOpacity() { setState()
{
    _opacidad = _opacidad == 1.0 ? 0.3 : 1.0;
}

}
@anular
Compilación del widget (contexto BuildContext) {
    return Fila( niños:
        <Widget>[ Opacidad
            Animada( duración:
                Duración(milisegundos: 500), opacidad: _opacidad, hijo:
                    Contenedor( color:
                        Colores.ámbar, altura:
                            100.0, ancho: 100.0, hijo:
                                FlatButton( hijo:
                                    Texto( 'Toque para
                                        desvanecerse'), onPressed:
                                        () { _animatedOpacity(); }, ),
                            ),
                            ),
                            ],
                            );
}
}
```

8. Continúe editando el archivo home.dart e importe el archivo animation_opacity.dart a la parte superior de la pagina

```
importar 'paquete: flutter/material.dart'; import
'paquete:ch7_animations/widgets/animated_container.dart';
```

```
import 'paquete:ch7_animations/widgets/animated_cross_fade.dart'; import 'paquete:ch7_animations/widgets/animated_opacity.dart';
```

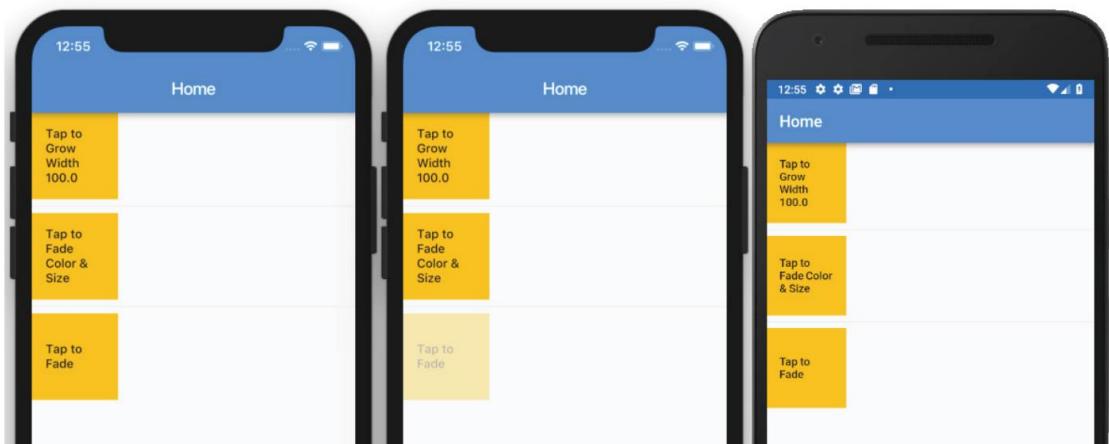
9. Justo después de la clase de widget AnimatedCrossFadeWidget() , agregue un widget Divider() . En el siguiente línea, agregue la llamada a la clase de widget AnimatedOpacityWidget().

cuerpo: SafeArea(niño:

Columna(niños:

```
<Widget>[  
    WidgetContenedorAnimado(),  
    Divisor(),  
    AnimatedCrossFadeWidget(),  
    Divisor(),  
    Widget de opacidad animada(), ], ), ),
```

Nuevamente, observe cuán limpio se ve el código y cómo mejora la legibilidad al separar cada sección principal del árbol de widgets.



Cómo funciona

El widget AnimatedOpacity toma un parámetro de duración y usa la clase Duration para especificar 500 milisegundos. El parámetro de opacidad es un valor de 0,0 a 1,0. El valor de opacidad de 1,0 es completamente visible y, a medida que el valor cambia hacia cero, comienza a desvanecerse. Una vez que llega a cero, es invisible.

USO DEL CONTROLADOR DE ANIMACIÓN

La clase `AnimationController` le brinda una mayor flexibilidad en la animación. La animación se puede reproducir hacia adelante o hacia atrás, y puede detenerla. La animación de lanzamiento utiliza una simulación física como un resorte.

La clase `AnimationController` produce valores lineales para una duración determinada e intenta mostrar un nuevo fotograma a unos 60 fotogramas por segundo. La clase `AnimationController` necesita una clase `Ticker Provider` pasando el argumento `vsync` en el constructor. El `vsync` evita que las animaciones fuera de pantalla consuman recursos innecesarios. Si la animación necesita solo un controlador de animación, utilice `SingleTickerProviderStateMixin`. Si la animación necesita varios controladores `AnimationCon`, use `TickerProviderStateMixin`. La clase `Ticker` está impulsada por `ScheduleBinding`. `ScheduleFrameCallback` informando una vez por cuadro de animación. Está tratando de sincronizar la animación para que sea lo más fluida posible.

El objeto predeterminado `AnimationController` varía de 0,0 a 1,0, pero si necesita un rango diferente, puede usar la clase `Animation` (usando `Tween`) para aceptar un tipo diferente de datos. La clase de animación se inicia estableciendo los valores de propiedad de inicio y fin de la clase `Tween` (intermedia) . Por ejemplo, tiene un globo que flota desde la parte inferior hasta la parte superior de la pantalla, y establecería el valor inicial de la clase `Tween` de 400,0, la parte inferior de la pantalla y el valor final de 0,0, la parte superior de la pantalla. Luego puede encadenar el método de animación `Tween` , que devuelve una clase de Animación . En pocas palabras, anima la interpolación en función de la animación, como una clase `CurvedAnimation` .

La clase `AnimationController` al principio puede parecer compleja de usar debido a las diferentes clases necesarias. Los siguientes son los pasos básicos que debe seguir para crear una animación personalizada (que se muestra en la Figura 7.1) o, finalmente, en el ejemplo, varias animaciones que se ejecutan al mismo tiempo.

1. Agregue el controlador de animación.
2. Agregar animación.
3. Inicie `AnimationController` con Duración (milisegundos, segundos, etc.).
4. Inicie la animación con `Tween` con valores de inicio y finalización y encadene el método de animación con una `CurvedAnimation` (para este ejemplo).
5. Use `AnimatedBuilder` con `Animación` usando un Contenedor con un globo para comenzar Animación llamando a `AnimationController.forward()` y `.reverse()` para ejecutar la animación hacia atrás. El widget `AnimatedBuilder` se usa para crear un widget que realiza una animación reutilizable.

Como puede ver, una vez que desglosa los pasos, se vuelve más manejable y menos complicado.

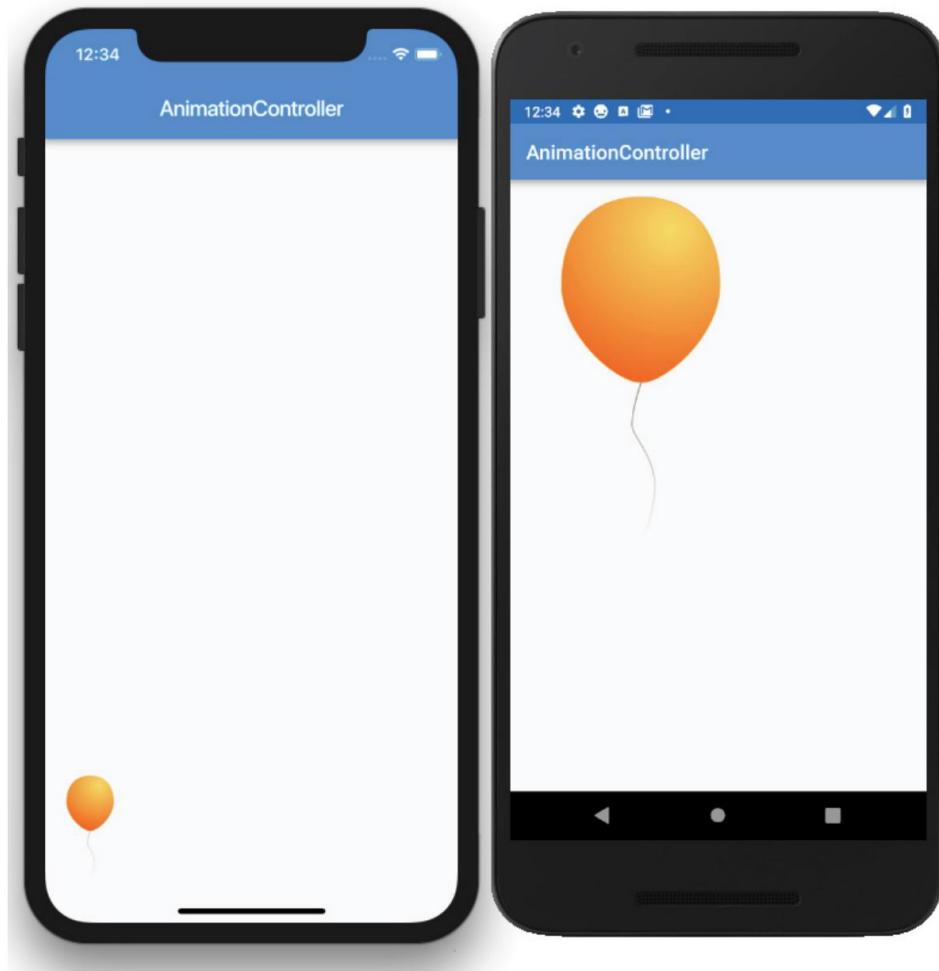


FIGURA 7.1: Lo que está construyendo con AnimationController

PRUÉBALO Creación de la aplicación AnimationController

En este proyecto, animarás un globo que comienza siendo pequeño en la parte inferior de la pantalla y, a medida que se infla, flota hacia la parte superior, lo que genera unas bonitas animaciones primaverales. Al tocar el globo con GestureDetector , la animación se invierte y muestra el globo desinflándose y flotando hacia abajo hasta la parte inferior de la pantalla. Cada vez que se toca el globo, la animación comienza de nuevo.

1. Cree un nuevo proyecto de Flutter y asígnele el nombre ch7_animation_controller. Puede seguir las instrucciones del Capítulo 4. Para este proyecto, necesita crear solo las páginas, los widgets y las carpetas de activos/ímágenes .

2. Abra el archivo pubspec.yaml y, en activos , agregue la carpeta de imágenes .

Para agregar activos a su aplicación, agregue una sección de activos, como esta: activos:

- activos/ imágenes/

Agregue los activos de la carpeta y las imágenes de la subcarpeta en la raíz del proyecto y luego copie el archivo GoogleFlutter-Balloon.png de inicio en la carpeta de imágenes .

3. Haga clic en el botón Guardar y, según el editor que esté utilizando, se ejecutarán automáticamente los paquetes flutter get. Una vez finalizado, muestra un mensaje de Proceso finalizado con el código de salida 0. Sin embargo, no ejecuta automáticamente el comando por usted. Abre la ventana de Terminal (ubicada en la parte inferior de tu editor) y escribe flutter packages get.

4. Cree un nuevo archivo Dart en la carpeta de widgets . Haga clic con el botón derecho en la carpeta de widgets , seleccione Nuevo Archivo de dardo, ingrese animation_balloon.dart y haga clic en el botón Aceptar para guardar.

5. Importe la biblioteca material.dart , agregue una nueva línea y luego comience a escribir st. Se abre la ayuda de autocompletado. Seleccione la abreviatura stful y asígnele el nombre AnimatedBalloonWidget.

6. Es apropiado crear las variables AnimationController y Animation antes de poder hacer referencia a ellos en su código. Declare TickerProviderStateMixin a la clase _AnimatedBalloonWidgetState agregando TickerProviderStateMixin. El argumento vsync del controlador de animación lo usará. Esto hace referencia a vsync , es decir, esta referencia de la clase _AnimatedBalloonWidgetState .

```
class _AnimatedBalloonWidgetState extiende State<AnimatedBalloonWidget> con  
TickerProviderStateMixin {
```

7. Después de que la clase _AnimatedBalloonWidgetState extienda State<AnimatedBalloonWidget> y antes de @override, cree dos AnimationControllers para manejar la duración del globo flotando hacia arriba y el inflado del globo inflado. Cree dos animaciones para manejar el rango de movimiento real y el tamaño de inflación. Anule los métodos initState() y dispose() para iniciar AnimationController y deséchelos cuando se cierre la página.

Una nota importante sobre el uso de dos AnimationControllers es que podría haber usado un AnimationController y hacer uso de la curva Interval() para escalonar la Animación. Una animación escalonada usa Interval() para comenzar y finalizar animaciones secuencialmente o para superponerse entre sí. En la sección "Uso de animaciones escalonadas" de este capítulo, creará una aplicación de animación escalonada.

```
class _AnimatedBalloonWidgetState extiende State<AnimatedBalloonWidget> con  
TickerProviderStateMixin {  
    Controlador de animación _controllerFloatUp;  
    AnimationController _controllerGrowSize;  
    Animación<doble> _animationFloatUp;  
    Animación<doble> _animationGrowSize;  
  
    @anular  
    void initState() {  
        super.initState();
```

```

_controllerFloatUp = AnimationController(duración: Duración(segundos: 4), vsync: esto);

_controllerGrowSize = AnimationController(duración: Duración(segundos: 2), vsync: esto); }

@anular
anular disponer () {

    _controllerFloatUp.dispose();
    _controllerGrowSize.dispose();
    super.dispose();
}

```

Tenga en cuenta que en el método initState() puede llamar a la clase de animación después del controlador de animación para iniciar la animación a medida que se carga la página, pero en su lugar va a colocar la clase de animación en la compilación de widgets (contexto BuildContext) . El motivo de esto es usar la clase Media Query para obtener el tamaño de pantalla para colocar y dimensionar el globo en consecuencia. El método initState() no contiene el objeto de contexto Widget que requiere MediaQuery . Sin embargo, cuando se llama a Widget build (BuildContext context) , la animación comienza al cargar la página. Si el usuario gira el dispositivo, se vuelve a llamar a Widget build (BuildContext context) y el globo cambiará de tamaño en consecuencia y ejecutará la animación si el globo se encuentra en la parte inferior de la pantalla.

- Después de compilar el widget (contexto BuildContext), cree la altura, el ancho y la posición inferior de la página del globo mediante MediaQuery.of(context).size. Llegué a las siguientes fórmulas para calcular las dimensiones al probar diferentes opciones para obtener el mejor aspecto según el tamaño y la orientación de la pantalla de los diferentes dispositivos:

```

Compilación del widget (contexto BuildContext)
{ double _balloonHeight = MediaQuery.of(context).size.height / 2; double
    _balloonWidth = MediaQuery.of(context).size.height / 3; double
    _balloonBottomLocation = MediaQuery.of(context).size.height -
    altura del globo;

```

Cree la clase Animation declarando una Tween con una CurvedAnimation. Para el valor FloatUp de _animation , la propiedad de inicio es la variable _balloonBottomLocation con la propiedad final en 0.0. Esto significa que la animación flotante hacia arriba comienza en la parte inferior de la pantalla y se mueve hasta la parte superior.

Para el valor _animationGrowSize , la propiedad de inicio es de 50,0 píxeles. Establezca el valor final en la variable _balloonWidth . Esto significa que puede comenzar el ancho del globo en 50,0 píxeles y aumentarlo hasta el valor máximo de _balloonWidth calculado por MediaQuery. Usa la curva elasticInOut para darle una hermosa animación primaveral que parezca un rápido inflado de aire.

- Llame a _controllerFloatUp.forward() y _controllerGrowSize.forward() para iniciar el animación.

```

_animationFloatUp = Tween(inicio: _balloonBottomLocation, fin: 0.0).
animate(CurvedAnimation(padre: _controllerFloatUp, curve: Curves.fastOutSlowIn)); _animationGrowSize =
Tween(inicio: 50,0, fin: _balloonWidth). animate(CurvedAnimation(padre:
_controllerGrowSize, curve: Curves.elasticInOut));

```

```
_controllerFloatUp.forward();
_controllerGrowSize.forward(); }
```

10. Cree AnimatedBuilder y echemos un vistazo a un desglose estructural de alto nivel para el Argumentos de AnimatedBuilder que muestran cómo se pasa el hijo de AnimatedBuilder (GestureDetector con imagen) al constructor, que es el widget para recibir la animación.

```
return AnimatedBuilder( animación:
    _animationFloatUp, builder: (context, child)
{ return Container( child: child, ); }, child:
    GestureDetector(*
    Image*) * );
```

El constructor AnimatedBuilder pasa el argumento de animación _animationFloatUp. Para el argumento del constructor , devuelva un widget de contenedor con la propiedad secundaria de child. Sé que suena extraño para la propiedad secundaria , pero esta es la forma en que el constructor vuelve a dibujar inteligentemente el control secundario que se está animando. El widget secundario que se pasa se declara a continuación y contendrá un widget GestureDetector y un widget secundario de Image (globo).

El constructor AnimatedBuilder tiene los argumentos animation, builder y child .

Agregue al widget GestureDetector() la propiedad onTap . Para el widget GestureDetector() , la propiedad onTap busca _controllerFloatUp.isCompleted (lo que significa que la animación está lista) y, si es así, inicia la animación al revés. Esto desinflará el globo y comenzará a flotar hasta la parte inferior de la página. La parte else maneja el globo que ya se encuentra en la parte inferior de la pantalla y comienza la animación hacia adelante flotando hacia arriba e inflando el globo de nuevo a su tamaño normal.

11. Agregue a la propiedad secundaria GestureDetector una llamada al constructor Image.asset() que carga el archivo BeginningGoogleFlutter-Balloon.png y establece la altura en _balloonHeight y el ancho en _balloonWidth.

```
devolver AnimatedBuilder(
    animación: _animationFloatUp, builder:
    (context, child) { return Container( child:
        child, margin:
            EdgeInsets.only( top:
                _animationFloatUp.value, ),
```

```
ancho: _animationGrowSize.value, ); }, child:  
  
GestureDetector( onTap: () { if  
  
    _controllerFloatUp.isCompleted) { _controllerFloatUp.reverse();  
    _controllerGrowSize.reverse(); } else  
    { _controllerFloatUp.forward();  
  
    _controllerGrowSize.forward(); } }, child:  
    Image.asset('activos/ímágenes/  
  
BeginningGoogleFlutter-  
Balloon.png', altura: _balloonHeight, ancho: _balloonWidth),  
  
, );
```

12. Abra el archivo home.dart e importe el archivo animation_balloon.dart en la parte superior de la página.

```
importar 'paquete: flutter/material.dart'; import  
'paquete:ch7_animation_controller/widgets/animated_balloon.dart';
```

13. Agregue al cuerpo un SafeArea con SingleChildScrollView como elemento secundario.

14. Agregue Padding como elemento secundario de SingleChildScrollView.

15. Agregue una columna como elemento secundario de Padding.

16. En los elementos secundarios Column , agregue la llamada a la clase de widget AnimatedBalloonWidget(). Tenga en cuenta que estoy usando NeverScrollableScrollPhysics() para detener SingleChildScrollView para desplazar el contenido.

```
cuerpo: SafeArea(  
    hijo: SingleChildScrollView (física:  
        NeverScrollableScrollPhysics(), hijo: relleno (  
  
        relleno: EdgeInsets.all(16.0), child:  
            Column( children:  
  
                <Widget>[ AnimatedBalloonWidget(), ], ), ), ), ), ),
```



Cómo funciona

Declarar TickerProviderStateMixin a la clase de widget AnimatedBalloonWidget le permitió establecer el argumento vsync de AnimationController . Agregó AnimationController y declaró la variable _controllerFloatUp para animar la acción flotante hacia arriba y hacia abajo. Declaró la variable AnimationController _controllerGrowSize para animar las acciones de inflado y desinflado.

Declaró la variable _animationFloatUp para contener el valor de la animación Tween para mostrar el globo flotando hacia arriba o hacia abajo configurando el margen superior del widget Contenedor . Declaró la variable _animationGrowSize para contener el valor de la animación Tween para mostrar el globo inflándose o desinflándose al establecer el valor de ancho del widget Contenedor .

El constructor AnimatedBuilder toma la animación, el constructor y los argumentos secundarios . A continuación, pasó la animación _animationFloatUp al constructor AnimatedBuilder . El argumento del constructor AnimatedBuilder devuelve un widget de contenedor con el elemento secundario como un widget de imagen envuelto en un widget de GestureDetector .

En el ejemplo anterior, mostré cómo puede usar varios AnimationControllers para ejecutarse al mismo tiempo con diferentes valores de Duración . En la siguiente sección, usará un AnimationController para una animación escalonada.

Uso de animaciones escalonadas

Una animación escalonada desencadena cambios visuales en orden secuencial. Los cambios de animación pueden ocurrir uno tras otro; pueden tener espacios sin animaciones y superponerse entre sí. Una clase AnimationController controla varios objetos Animation que especifican la animación en una línea de tiempo (Intervalo). Ahora verá un ejemplo del uso de una clase AnimationController y la propiedad de curva Interval() para iniciar diferentes animaciones en diferentes momentos. Como se señaló en la sección anterior, una animación escalonada usa Interval() para comenzar y finalizar animaciones secuencialmente o para superponerse entre sí.

PRUÉBALO Creación de la aplicación de animaciones escalonadas

En este proyecto, volverá a crear la animación del globo para duplicar el ejemplo anterior, pero usará solo un `AnimationController` para aprovechar las animaciones escalonadas. Al usar `Interval()`, marca la hora de inicio y finalización de cada animación para escalonar las animaciones.

Al igual que en el proyecto anterior, animará un globo que comienza siendo pequeño en la parte inferior de la pantalla y, a medida que se infla, flota hacia la parte superior, lo que genera unas bonitas animaciones primaverales. Al tocar el globo con `GestureDetector`, la animación se invierte y muestra el globo desinflándose y flotando hacia abajo hasta la parte inferior de la pantalla. Cada vez que se toca el globo, la animación comienza de nuevo.

1. Cree un nuevo proyecto de Flutter y asignele el nombre `ch7_ac_staggered_animations`. Puede seguir las instrucciones del Capítulo 4. Para este proyecto, necesita crear solo las páginas, los widgets y las carpetas de activos/ imágenes .

2. Abra el archivo `pubspec.yaml` y, en activos, agregue la carpeta de imágenes .

Para agregar activos a su aplicación, agregue una sección de activos, como esta: activos:

- activos/ imágenes/

Agregue los activos de la carpeta y las imágenes de la subcarpeta en la raíz del proyecto y luego copie el archivo `GoogleFlutter-Balloon.png` de inicio en la carpeta de imágenes .

3. Cree un nuevo archivo Dart en la carpeta de widgets . Haga clic con el botón derecho en la carpeta de widgets , seleccione Nuevo Archivo de dardo, ingrese `animation_balloon.dart` y haga clic en el botón Aceptar para guardar. Importe la biblioteca `material.dart` , agregue una nueva línea y luego comience a escribir `st`. Se abre la ayuda de autocompletado. Seleccione la abreviatura `stful` y asignele el nombre `AnimatedBalloonWidget`.

4. Es apropiado crear las variables `AnimationController` y `Animation` antes de poder hacer referencia a ellos en el código. Declare `SingleTickerProviderStateMixin` a la clase `_HomeState` agregando `SingleTickerProviderStateMixin`. El constructor `AnimationController` toma el argumento `vsync` . Esto hace referencia al argumento `vsync` , es decir, esta referencia de la clase `_HomeState` .

```
class _AnimatedBalloonWidgetState extiende State<AnimatedBalloonWidget> con  
SingleTickerProviderStateMixin {
```

5. Cree un `AnimationController` solo para manejar la duración de la animación. Cree dos animaciones para manejar el rango de movimiento real y el tamaño creciente. Anule los métodos `initState()` y `dispose()` para iniciar `AnimationController` y deséchelos cuando se cierre la página. class `_HomeState` extiende `State<Home>` con

```
SingleTickerProviderStateMixin { AnimationController _controller; Animación<doble> _animationFloatUp;  
Animación<doble> _animationGrowSize;
```

```
@anular  
void initState() {  
super.initState();
```

```
_controller = AnimationController(duración: Duración(segundos: 4), vsync: esto); }

@anular
anular disponer () {

    _controlador.dispose();
    super.dispose();
}
```

Tenga en cuenta que, como se indicó en la sección anterior, en el método initState() puede llamar a Animation solo después de AnimationController solo para iniciar la animación a medida que se carga la página, pero en su lugar lo colocará en el método _animatedBalloon(). El motivo es usar la clase MediaQuery para obtener el tamaño de pantalla para colocar y dimensionar el globo en consecuencia. El método initState() no contiene el objeto de contexto Widget que requiere MediaQuery. Sin embargo, cuando se llama al método _animate Balloon() , la animación comienza al cargar la página. Si el usuario gira el dispositivo, se vuelve a llamar al método _animateBalloon() y el globo cambia de tamaño en consecuencia y ejecuta la animación si el globo se encuentra en la parte inferior de la pantalla.

6. Despues de compilar el widget (contexto BuildContext), cree la altura, el ancho y la posición inferior de la página del globo mediante MediaQuery.of(context).size. Llegué a las siguientes fórmulas para calcular las dimensiones al probar diferentes opciones para obtener la mejor apariencia estética según el tamaño y la orientación de la pantalla de los diferentes dispositivos.

```
Compilación del widget (contexto BuildContext) { double
    _balloonHeight = MediaQuery.of(context).size.height / 2; double _balloonWidth =
    MediaQuery.of(context).size.height / 3; double _balloonBottomLocation =
    MediaQuery.of(context).size.height -
    altura del globo;
```

Crear una interpolación de animación es casi lo mismo que en el ejemplo anterior, excepto que el padre de CurvedAnimation es el _controller. La curva usa Interval() para marcar la hora de inicio y fin de cada animación. Cero significa comienzo y animación 1.0 significa fin.

El valor de duración de _controller es de cuatro segundos, el inicio de _animationGrowSize Interval es 0,0 y el final es 0,5. Esto significa que el globo comienza a inflarse tan pronto como comienza la animación, pero termina en 0,5, lo que significa dos segundos. Puede pensar en el inicio y fin del intervalo como un porcentaje de la duración total de la animación.

```
void _animationFloatUp = Tween(inicio: _balloonBottomLocation, fin: 0.0).animate(
    CurvedAnimation (padre:
        _controlador, curva: Intervalo
        (0.0, 1.0, curva: Curves.fastOutSlowIn),
    ),
);
```

```
void _animationGrowSize = Tween(begin: 50.0, end: _balloonWidth).animate(AnimaciónCurva( padre: _controlador,  
curva: Intervalo(0.0, 0.5,  
curva: Curves.elasticInOut), ), );
```

7. Cree solo el AnimatedBuilder y veamos el desglose estructural de alto nivel para el Constructor de AnimatedBuilder que muestra cómo se pasa el argumento secundario de AnimatedBuilder (GestureDetector con imagen) al constructor, que es el widget que recibe el argumento de animación .

```
return AnimatedBuilder( animación:  
_animationFloatUp, builder: (context, child)  
{ return Container( child: child, ); }, child:  
GestureDetector(*  
Image*) */ );
```

El uso de AnimatedBuilder es casi lo mismo que el ejemplo anterior, con la diferencia de que usa solo un AnimationController solo para iniciar la animación hacia adelante o hacia atrás con la variable _controller.

El constructor AnimatedBuilder toma la animación, el constructor y los argumentos secundarios .

Agregue al widget GestureDetector() la propiedad onTap . Para GestureDetector(), la devolución de llamada de la propiedad onTap busca _controllerFloatUp.isCompleted (la animación está lista) y, en caso afirmativo, inicia la animación al revés. Esto desinflará el globo y hará que comience a flotar hasta la parte inferior de la página. La parte else maneja el globo que ya se encuentra en la parte inferior de la pantalla; comienza la animación haciendo flotar el globo hacia arriba e inflándolo de nuevo a su tamaño normal. Notarás que dado que solo tienes un AnimationController, usas la variable _controller para revertir o adelantar la animación.

8. Agregue al elemento secundario GestureDetector una llamada al constructor Image.asset() que carga el BeginningGoogleFlutter-Balloon.png y establece el valor de alto en _balloonHeight y el valor de ancho en _balloonWidth.

```
devolver AnimatedBuilder(  
animación: _animationFloatUp, builder:  
(context, child) { return Container( child:  
child, margin:  
EdgeInsets.only( top:  
_animationFloatUp.value, ),
```

```
ancho: _animationGrowSize.value, ); }, child:  
  
GestureDetector( onTap: () { if  
  
    _controller.isCompleted) { _controller.reverse(); }  
    else { _controller.forward(); } },  
    child:  
        Image.asset('assets/images/  
  
BeginningGoogleFlutter-Balloon .png',  
        alto: _balloonHeight, ancho: _balloonWidth),  
  
, );
```

9. Abra el archivo home.dart e importe el archivo animation_balloon.dart en la parte superior de la página:

```
importar 'paquete: flutter/material.dart'; importar  
'paquete:ch7_ac_staggered_animations/widgets/animated_balloon.dart';
```

10. Agregue al cuerpo un widget SafeArea con SingleChildScrollView como elemento secundario.

11. Agregue Padding como elemento secundario de SingleChildScrollView. Agregue un widget de columna como elemento secundario de Padding.

12. En los hijos de la columna , agregue la llamada al método _animateBalloon(). Tenga en cuenta que estoy usando NeverScrollableScrollPhysics() para detener SingleChildScrollView para desplazar el contenido.

```
cuerpo: SafeArea(  
    hijo: SingleChildScrollView (física:  
        NeverScrollableScrollPhysics(), hijo: relleno (  
  
        relleno: EdgeInsets.all(16.0), child:  
            Column( children:  
  
                <Widget>[ AnimatedBalloonWidget(), ], ), ), ), ), ),
```

Cómo funciona

Declarar SingleTickerProviderStateMixin a la clase de widget AnimatedBalloonWidget le permite establecer el argumento vsync de AnimationController . SingleTickerProviderStateMixin permite solo un AnimationController . Agregó AnimationController y declaró la variable _controller para animar tanto el flotar hacia arriba o hacia abajo como el inflado o desinflado del globo.

Declaró la variable _animationFloatUp para contener el valor de la animación Tween para mostrar el globo flotando hacia arriba o hacia abajo configurando el margen superior del widget Contenedor . También declaró la variable _animationGrowSize para contener el valor de la animación Tween para mostrar el globo inflando o desinflando configurando el valor de ancho del widget Contenedor .

El constructor `AnimatedBuilder` toma argumentos de animación, constructor y niño . A continuación, pasó la animación `_animationFloatUp` al constructor `AnimatedBuilder` . El argumento del constructor `AnimatedBuilder` devuelve un widget de contenedor con el elemento secundario como una imagen envuelta en un widget de `GestureDetector` .

RESUMEN

En este capítulo, aprendió cómo agregar animaciones a su aplicación para mejorar la UX. Implementó `AnimatedContainer` para animar el ancho de un widget de contenedor con un hermoso efecto de resorte usando `Curves.elasticOut`. Agregó el widget `AnimatedCrossFade` para realizar un fundido cruzado entre dos widgets secundarios. El color está animado de ámbar a verde mientras que al mismo tiempo el widget aumenta o disminuye en ancho y alto. Para atenuar un widget hacia adentro, hacia afuera o parcialmente, agregó el widget `AnimatedOpacity` . El widget `AnimatedOpacity` usa la propiedad de opacidad pasada durante un período de tiempo (Duración) para atenuar el widget. La clase `AnimationController` permite la creación de animaciones personalizadas.

Aprendiste a usar múltiples `AnimationControllers` con diferentes duraciones. Usaste dos clases de Animación para controlar la flotación hacia arriba o hacia abajo y el inflado y desinflado del globo al mismo tiempo. La animación se crea utilizando Tween con valores iniciales y finales . También usó una clase `CurvedAnimation` diferente para un efecto no lineal como `Curves.fastOutSlowIn` para flotar hacia arriba o hacia abajo y `Curves.elasticInOut` para inflar o desinflar el globo. Por último, usó una clase `AnimationController` con varias clases de Animación para crear animaciones escalonadas, que dan un efecto similar al del ejemplo anterior.

En el próximo capítulo, aprenderá las muchas formas de usar la navegación, como `Navigator`, `Hero Animation`, `BottomNavigationBar`, `BottomAppBar`, `TabBar`, `TabBarView` y `Drawer`.

LO QUE APRENDISTE EN ESTE CAPÍTULO

TEMA	CONCEPTOS CLAVE
Contenedor animado	Esto cambia gradualmente los valores con el tiempo.
Fundido cruzado animado	Este fundido cruzado entre dos widgets secundarios.
Opacidad animada	Esto muestra u oculta la visibilidad del widget al animar el desvanecimiento con el tiempo.
constructor animado	Esto se usa para crear un widget que realiza una animación reutilizable.
Controlador de animación	Esto crea animaciones personalizadas usando TickerProviderStateMixin, Único TickerProviderStateMixin, AnimationController, Animation, Tween y CurvedAnimation.

8

Creación de la navegación de una aplicación

LO QUE APRENDERÁS EN ESTE CAPÍTULO

Cómo usar el widget Navigator para navegar entre páginas

Cómo la animación de héroe permite que la transición de un widget vuele a su lugar desde una página a otro

Cómo mostrar una lista horizontal de BottomNavigationBarItems que contiene un ícono y un título en la parte inferior de la página

Cómo mejorar el aspecto de una barra de navegación inferior con BottomAppBar widget, que permite habilitar una muesca

Cómo mostrar una fila horizontal de pestañas con TabBar

Cómo usar TabBarView junto con TabBar para mostrar la página del pestaña seleccionada

Cómo Drawer permite al usuario deslizar un panel desde la izquierda o la derecha

Cómo utilizar el constructor ListView para crear rápidamente una breve lista de elementos

Cómo usar el constructor ListView con el widget Drawer para mostrar una lista de menú

En este capítulo, aprenderá que la navegación es un componente principal en una aplicación móvil. Una buena navegación crea una excelente experiencia de usuario (UX) al facilitar el acceso a la información. Por ejemplo, imagine hacer una entrada de diario y, al intentar seleccionar una etiqueta, no está disponible, por lo que debe crear una nueva. ¿Cierras la entrada y vas a Ajustes Etiquetas para añadir una nueva? Eso sería torpe. En cambio, el usuario necesita la capacidad de agregar una nueva etiqueta sobre la marcha y navegar adecuadamente para seleccionar o agregar una etiqueta desde su posición actual. Al diseñar una aplicación, siempre tenga en cuenta cómo navegaría el usuario a diferentes partes de la aplicación con la menor cantidad de toques.

La animación durante la navegación a diferentes páginas también es importante si ayuda a transmitir una acción, en lugar de ser simplemente una distracción. ¿Qué quiere decir esto? El hecho de que pueda mostrar animaciones elegantes no significa que deba hacerlo. Usa animaciones para mejorar la UX, no para frustrar al usuario.

USO DEL NAVEGADOR

El widget Navigator administra una pila de rutas para moverse entre páginas. Opcionalmente, puede pasar datos a la página de destino y volver a la página original. Para comenzar a navegar entre páginas, utilice los métodos `Navigator.push`, `pushNamed` y `pop`. (Aprenderá cómo usar el método `pushNamed` en la sección "Uso de la ruta de Navigator con nombre" de este capítulo). Navigator es increíblemente inteligente; muestra navegación nativa en iOS o Android. Por ejemplo, en iOS, al navegar a una nueva página, generalmente desliza la siguiente página desde el lado derecho de la pantalla hacia la izquierda. En Android, cuando navega a una página nueva, normalmente desliza la página siguiente desde la parte inferior de la pantalla hacia la parte superior. Para resumir, en iOS, la nueva página se desliza desde la derecha y en Android, se desliza desde la parte inferior.

El siguiente ejemplo le muestra cómo usar el método `Navigator.push` para navegar a la página Acerca de. El método `push` pasa los argumentos `BuildContext` y `Route`. Para impulsar un nuevo argumento de ruta, crea una instancia de la clase `MaterialPageRoute` que reemplaza la pantalla con la transición de animación de la plataforma adecuada (iOS o Android). En el ejemplo, la propiedad `fullscreenDialog` se establece en `true` para presentar la página Acerca de como un cuadro de diálogo modal de pantalla completa.

Al establecer la propiedad `fullscreenDialog` en verdadero, la barra de aplicaciones de la página Acerca de incluye automáticamente un botón de cierre. En iOS, la transición de diálogo modal presenta la página deslizándose desde la parte inferior de la pantalla hacia la parte superior, y esta también es la opción predeterminada para Android.

```
Navigator.push( contexto,  
    MaterialPageRoute( fullscreenDialog:  
        true, builder: (context) => About(), ), );
```

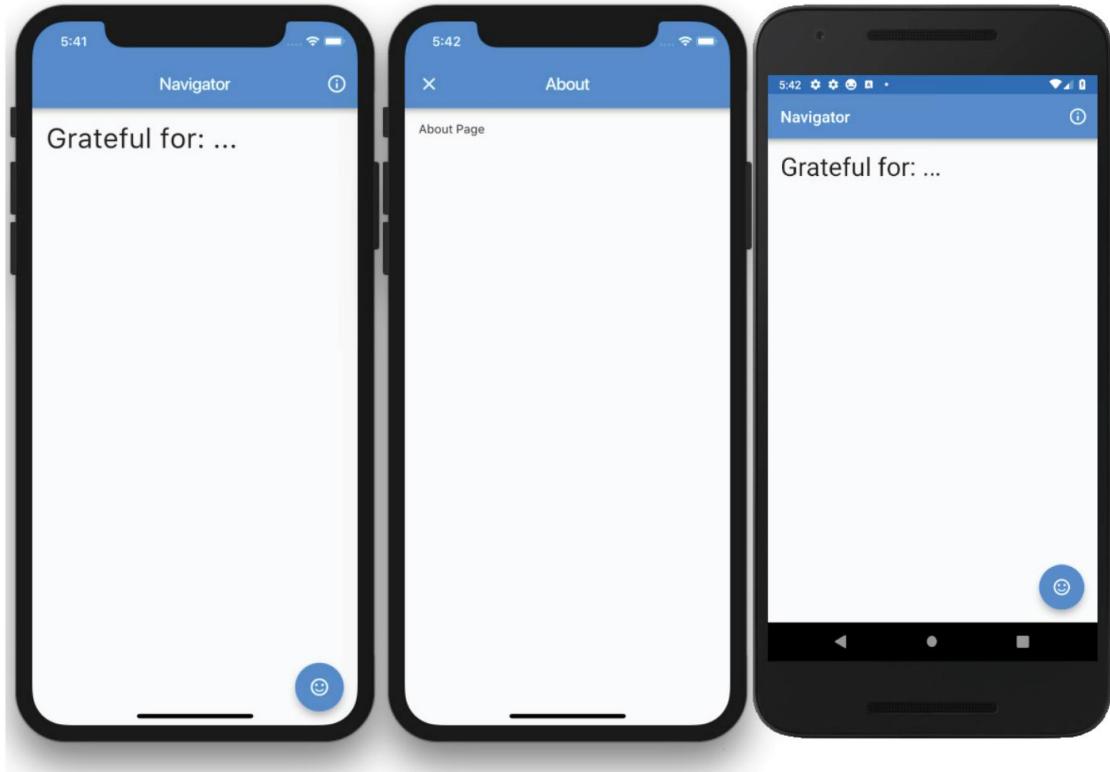
El siguiente ejemplo muestra cómo usar el método `Navigator.pop` para cerrar la página y volver a la página anterior. Llama al método `Navigator.pop(context)` pasando el argumento `BuildContext`, y la página se cierra deslizándose desde la parte superior de la pantalla hacia la parte inferior. El segundo ejemplo muestra cómo pasar un valor a la página anterior.

```
// Cerrar página  
Navigator.pop(context);  
  
// Cierra la página y pasa un valor a la página anterior  
Navigator.pop(context,  
'Done');
```

PRUÉBELO Crear la aplicación Navigator, Parte 1—La página Acerca de

El proyecto tiene una página de inicio principal con un `FloatingActionButton` para navegar a una página de agradecimiento pasando un valor de botón de opción predeterminado. La página de agradecimiento muestra tres botones de radio para seleccionar un valor y luego pasarlo a la página de inicio y actualizar el widget de texto con el valor apropiado.

AppBar tiene un IconButton de acciones que navega a la página Acerca de pasando un argumento fullscreenDialog establecido en verdadero para crear un cuadro de diálogo modal de pantalla completa . El cuadro de diálogo modal muestra un botón de cierre en la parte superior izquierda de la página y se anima desde la parte inferior. En esta primera parte, desarrollará la navegación desde la página principal hasta la página Acerca de y viceversa.



1. Cree un nuevo proyecto de Flutter y asígnele el nombre ch8_navigator. Consulte las instrucciones del Capítulo 4, "Creación de una plantilla de proyecto inicial". Para este proyecto, solo necesita crear la carpeta de páginas .

2. Abra el archivo home.dart y agregue un IconButton a la lista de widgets de acciones de la barra de aplicaciones .

La propiedad IconButton onPressed llamará al método _openPageAbout() y pasará los argumentos context y fullscreenDialog . No se preocupe por las líneas rojas onduladas debajo del nombre del método; creará ese método en pasos posteriores. El widget Navigator necesita el argumento de contexto , y el argumento fullscreenDialog se establece en verdadero para mostrar la página Acerca de como un modal de pantalla completa. Si establece el argumento fullscreenDialog en falso, la página Acerca de muestra una flecha hacia atrás en lugar de un ícono de botón de cierre.

```
appBar:  
  AppBar( título: Text('Navegador'),  
    acciones:
```

```
    <Widget>[ IconButton( icon: Icon(Icons.info_outline),
```

```
onPressed: () => _openPageAbout(  
    contexto: contexto,  
    pantalla completaDiálogo:
```

```
verdadero, ), ), ], ),
```

3. Agregue un SafeArea con Padding como elemento secundario al cuerpo.

```
cuerpo: SafeArea  
(hijo: relleno (),),
```

4. En el elemento secundario Padding, agregue un widget Text() , con el texto 'Agradecido por: \$_howAreYou' . Observe la variable \$_howAreYou , que contendrá el valor devuelto cuando navegue hacia atrás (Navigator.pop) desde la página de agradecimiento. Agregue una clase TextStyle con un valor fontSize de 32,0 pixeles.

```
cuerpo: SafeArea(  
    niño: relleno (  
        padding: EdgeInsets.all(16.0), child:  
            Text('Agradecido por: $_howAreYou', style: TextStyle(fontSize: 32.0),), ), ),
```

5. A la propiedad Scaffold floatingActionButton , agregue un widget FloatingActionButton() con un onPressed que llame al método _openPageGratitude(context: context) , que pasa el argumento de contexto .

6. Para el elemento secundario FloatingActionButton(), agregue el ícono llamado sentiment_satisfied , y para la información sobre herramientas, agregue la descripción "Acerca de" .

```
botón de acción flotante: Botón de acción flotante (onPressed:  
    () => _openPageGratitude (contexto: contexto), información sobre  
    herramientas: 'Acerca  
    de', hijo: Ícono (Icons.sentiment_satisfied),),
```

7. En la primera línea después de la definición de la clase _HomeState , agregue una variable de cadena llamada _howAreYou con un valor predeterminado de '...'.

```
Cadena _cómo estás = "...";
```

8. Continúe agregando el método _openPageAbout() que acepta BuildContext y bool named parámetros con los valores predeterminados establecidos en false.

```
void _openPageAbout({BuildContext context, bool fullscreenDialog = false}) {}
```

9. En el método openPageAbout() , agregue un método Navigator.push() con un contexto y un segundo argumento de MaterialPageRoute(). La clase MaterialPageRoute() pasa el argumento Dialog de pantalla completa y un constructor que llama a la página About() que creará en pasos posteriores.

```
void _openPageAbout({BuildContext context, bool fullscreenDialog = false}) {  
    Navegador.push(
```

```
contexto,  
RutaPáginaMaterial(  
    fullscreenDialog: fullscreenDialog, constructor:  
    (contexto) => Acerca de(), ), );  
  
}
```

10. En la parte superior de la página home.dart , importe la página about.dart que creará a continuación.

```
importar 'acerca de.dart';
```

Navigator es simple de usar pero también poderoso. Examinemos cómo funciona. Navigator.push() pasa dos argumentos: contexto y MaterialPageRoute. Para el primer argumento, pasa el argumento de contexto . El segundo argumento, MaterialPageRoute(), le brinda la potencia necesaria para navegar a otra página mediante una animación específica de la plataforma. Solo se requiere el constructor para navegar con el argumento fullscreenDialog opcional .

11. Cree un nuevo archivo llamado about.dart en la carpeta lib/pages . Dado que esta página solo muestra información mación, cree una clase StatelessWidget llamada Acerca de.

12. Para el cuerpo, agregue el área segura habitual con relleno y la propiedad secundaria como un widget de texto .

```
// about.dart import  
'paquete:flutter/material.dart';  
  
clase Acerca de extiende StatelessWidget {  
    @anular  
    Creación de widgets (contexto BuildContext) { return  
        Scaffold (  
            appBar: AppBar(título:  
                Texto('Acerca de'),  
            ),  
            cuerpo: SafeArea(  
                niñ: Relleno (relleno:  
                    const EdgeInsets.all (16.0), niñ: Texto ('Acerca de la  
                    página'),),  
                ),  
            );  
        }  
    }  
}
```

Cómo funciona

Agrega a AppBar un IconButton debajo de la propiedad de acciones . La propiedad icon para IconButton se establece en Icons.info_outline, el método _openPageAbout() pasa el contexto y el argumento fullscreenDialog se establece en true. También agrega a Scaffold un FloatingActionButton que llama al método _openPageGratitude() . El método _openPageAbout() usa el Navegador. método push() para pasar el contexto y MaterialPageRoute. MaterialPageRoute pasa el argumento fullscreenDialog establecido en verdadero y el constructor llama a la página Acerca de () . La clase de página Acerca de es un StatelessWidget con Scaffold y AppBar; la propiedad body tiene un SafeArea con Padding como elemento secundario que muestra un widget de texto con el texto "Acerca de la página".

PRUÉBALO Creando la aplicación Navigator, Parte 2—La Página de Gratitud

La segunda parte de la aplicación es navegar a la página de gratitud pasando un valor predeterminado para seleccionar el botón de opción apropiado . Una vez que navega de regreso a la página de inicio, el valor del botón de radio recién seleccionado se transfiere y se muestra en el widget de texto .

1. Abra el archivo home.dart y, después del método _openPageAbout() , agregue _openPage

Método de gratitud() . El método _openPageGratitude() toma dos parámetros: un contexto y una variable bool fullscreenDialog con un valor predeterminado falso. En este caso, la página de agradecimiento no es un diálogo de pantalla completa. Al igual que MaterialPageRoute anterior , el constructor abre la página.

En este caso, es la página de agradecimiento.

Tenga en cuenta que al pasar datos a la página de gratitud y esperar a recibir una respuesta, el método se marca como asíncrono para esperar una respuesta de Navigator.push mediante la palabra clave await .

```
void _openPageGratitude(  
    {Contexto BuildContext, bool fullscreenDialog = false}) asíncrono {  
    final String _gratitudeResponse = esperar Navigator.push(  
        
```

El constructor MaterialPageRoute crea el contenido de la ruta. En este caso, el contenido es la página de agradecimiento, que acepta un parámetro int radioGroupValue con un valor de -1. El valor -1 le dice a la página de clase de Gratitud que no seleccione ningún botón de opción . Si pasa un valor como 2, selecciona el botón de radio apropiado que corresponde a este valor.

```
constructor: (contexto) => Gratitud( radioGroupValue:  
    -1, ),
```

Una vez que el usuario descarta la página de agradecimiento, se completa la variable _gratitudeResponse . Usa el ?? (doble signo de interrogación si es nulo) para verificar _gratitudeResponse en busca de un valor válido (no nulo) y completar la variable _howAreYou . El widget de texto se completa con el valor _howAreYou en la página de inicio con el valor de gratitud seleccionado apropiado o una cadena vacía. En otras palabras, si el valor de _gratitudeResponse no es nulo , la variable _howAreYou se completa con el valor de _gratitudeResponse ; de lo contrario, la variable _howAreYou se completa con una cadena vacía.

```
_howareyou = _gratitudeResponse ?? ";
```

Aquí está el código completo del método _openPageGratitude() :

```
void _openPageGratitude(  
    {Contexto BuildContext, bool fullscreenDialog = false}) asíncrono {  
    final String _gratitudeResponse = esperar Navigator.push(  
        contexto,  
        RutaPáginaMaterial(  
            fullscreenDialog: fullscreenDialog, constructor:  
            (contexto) => Gratitud (radioGroupValue: -1, ),
```

```
), ); _howareyou = _gratitudeResponse ?? "; }
```

2. En la parte superior de la página home.dart , agregue la página thanks.dart que creará a continuación.

```
import 'gratitud.dart';
```

3. Cree un nuevo archivo llamado gratitud.dart en la carpeta lib/pages . Dado que esta página se modificará data (estado), cree una clase StatefulWidget llamada Gratitud.

Para recibir datos pasados desde la página de inicio, modifique la clase Gratitud agregando una variable int final denominada radioGroupValue . Tenga en cuenta que la variable final no comienza con un guión bajo. Cree un constructor con nombre que requiera este parámetro. La clase _GratitudeState accede a la variable radioGroupValue y extiende State<Gratitude> llamando a widget.radioGroupValue .

```
clase Gratitud extiende StatefulWidget {  
    final int radioGroupValue;  
  
    Gratitud ({Clave clave, @required this.radioGroupValue}) : super(clave: clave);  
  
    @anular  
    _GratitudeState createState() => _GratitudeState();  
}
```

4. Para Scaffold AppBar, agregue un IconButton a la lista de acciones de Widget. Establezca el ícono IconButton en Icons.check con la propiedad onPressed llamando a Navigator.pop , que devuelve _selectedGratitude a la página de inicio.

```
appBar: AppBar(título:  
    Texto('Agradecimiento'), acciones:  
    <Widget>[ IconButton( icon:  
        Icon(Icons.check),  
        onPressed: () => Navigator.pop(context,  
            _selectedGratitude), ),  
  
    ],  
,
```

5. Para el cuerpo, agregue la propiedad habitual SafeArea y Padding with child como Fila.

La lista de filas secundarias de Widget contiene tres widgets de radio y texto alternados . El widget Radio toma las propiedades value, groupValue y onChanged . La propiedad de valor es el valor de ID para el botón de opción . La propiedad groupValue contiene el valor del botón de opción actualmente seleccionado . onChanged pasa el valor de índice seleccionado al método personalizado _radio onChanged() que maneja qué botón de opción está seleccionado actualmente. Después de cada botón de opción , hay un widget de texto que actúa como una etiqueta para el botón de opción .

Aquí está el código fuente de cuerpo completo:

```
cuerpo: SafeArea(
    child: Padding( padding:
        const EdgeInsets.all(16.0), child: Row( children:
            <Widget>[ Radio( value: 0,
                groupValue:
                    _radioGroupValue, onChanged: (index) =>
                    _radioOnChanged(index), ), Text( 'Familia'), Radio( valor: 1,
                groupValue:
                    _radioGroupValue,
                    onChanged: (índice) =>
                    _radioOnChanged(index), ), Text('Friends'), Radio( value: 2,
                groupValue:
                    _radioGroupValue,
                    onChanged: (índice ) =>
                    _radioOnChanged(indice), ), Texto('Café'),
            ],
        ),
    ),
),
```

6. En la primera línea después de la definición de la clase `_HomeState` , agregue tres variables: `_gratitudeList`, `_selectedGratitude` y `_radioGroupValue`, y el método `_radioOnChanged()` .

`_gratitudeList` es una lista de valores de cadena .

`Lista<String> _gratitudeList = Lista();`

`_selectedGratitude` es una variable de cadena que contiene el valor del botón de opción seleccionado .

`String _agradecimiento seleccionado;`

`_radioGroupValue` es un int que contiene el ID del valor del botón de opción seleccionado .

`int _radioGroupValue;`

7. Cree el método `_radioOnChanged()` tomando un int para el índice seleccionado del botón de opción .

En el método, llama a `setState()` para que los widgets de Radio se actualicen con el valor seleccionado . La variable `_radioGroupValue` se actualiza con el índice. La variable `_selectedGratitude` (valor de ejemplo `Coffee`) se actualiza tomando el valor de la lista `_gratitudeList[index]` por el índice seleccionado (posición en la lista).

```
void _radioOnChanged(int index) { setState()
{ _radioGroupValue
= index; _selectedGratitude =
_gratitudeList[index]; print('_selectedRadioValue
$_selectedGratitude'); };
```

8. Anule initState() para inicializar _gratitudeList. Dado que se pasa el _radioGroupValue desde la página de inicio, inicialícelo con widget.radioGroupValue, que es la variable final que se pasa desde la página de inicio.

```
_gratitudeList..add('Familia')..add('Amigos')..add('Café'); _radioGroupValue =  
widget.radioGroupValue;
```

El siguiente es el código que declara todas las variables y métodos:

```
class _GratitudeState extiende Estado<Gratitud> {  
    Lista<String> _gratitudeList = Lista();  
    String _agradecimiento seleccionado;  
    int _radioGroupValue;  
  
    void _radioOnChanged(index int) { setState()  
    {  
        _radioGroupValue = índice;  
        _gratitud seleccionada = _gratitudeList[index];  
        print('_selectedRadioValue $_selectedGratitude');});  
  
    }  
  
    @anular  
    void initState() {  
        super.initState();  
  
        _gratitudeList..add('Familia')..add('Amigos')..add('Café'); _radioGroupValue =  
        widget.radioGroupValue;  
    }  
}
```

Aquí está el código fuente completo del archivo thanks.dart :

```
importar 'paquete: flutter/material.dart';  
  
clase Gratitud extiende StatefulWidget {  
    final int radioGroupValue;  
  
    Gratitud ({Clave clave, @required this.radioGroupValue}) : super(clave: clave);  
  
    @anular  
    _GratitudeState createState() => _GratitudeState(); }  
  
class _GratitudeState extiende Estado<Gratitud> {  
    Lista<String> _gratitudeList = Lista();  
    String _agradecimiento seleccionado;  
    int _radioGroupValue;  
  
    void _radioOnChanged(int index) { setState()  
    { _radioGroupValue  
        = index; _selectedGratitude =  
        _gratitudeList[index]; print('_selectedRadioValue  
        $_selectedGratitude');});  
    }  
}
```

```
@anular
void initState() {
    super.initState();

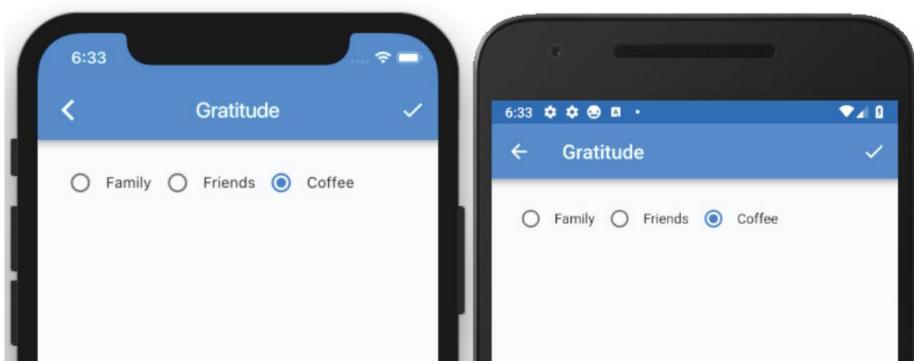
    _gratitudeList..add('Familia')..add('Amigos')..add('Café'); _radioGroupValue =
    widget.radioGroupValue; }

@anular
Creación de widgets (contexto BuildContext) { return
Scaffold (
    appBar: AppBar( title:
        Text('Gratitud'), acciones:
        <Widget>[ IconButton( icon:
            Icon(Icons.check),
            onPressed: () => Navigator.pop(context,
                _selectedGratitude), ), ], ), cuerpo: ÁreaSegura(


child: Padding( padding:
    const EdgeInsets.all(16.0), child: Row( children:

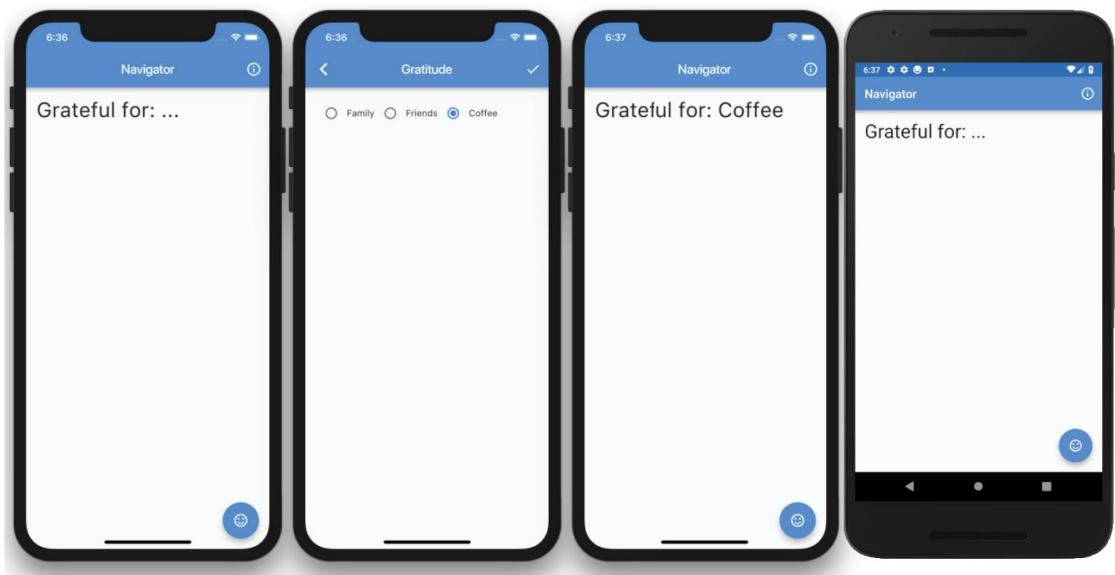
    <Widget>[ Radio( value: 0,
        groupValue:
            _radioGroupValue, onChanged: (index) =>
            _radioOnChanged(index), ), Text( 'Familia'), Radio( valor: 1,
        groupValue:
            _radioGroupValue,
            onChanged: (indice) =>
            _radioOnChanged(indice), ), Text("Friends"), Radio( value: 2,
        groupValue:
            _radioGroupValue,
            onChanged: (índice ) =>
            _radioOnChanged(index), ), Text('Café'), ], ), ); }

})}
```



Cómo funciona

Tiene toda la aplicación creada con una página de inicio que puede navegar a la página Acerca de como un cuadro de diálogo de pantalla completa. El fullscreenDialog le da a la página Acerca de un botón de acción de cierre predeterminado. Al tocar el FloatingActionButton de la página de inicio, el constructor Navigator MaterialPageRoute crea el contenido de la ruta, en este caso, la página de agradecimiento. A través del constructor Gratitud, los datos se pasan a los botones de radio no seleccionados . Desde la página de gratitud, una lista de botones de radio le da la opción de seleccionar una gratitud. Al tocar el botón de acción de AppBar (casilla de verificación IconButton), el método Navigator.pop pasa el valor de gratitud seleccionado al widget Texto de inicio . Desde la página de inicio, llamó al método Navigator.push usando la palabra clave await y el método ha estado esperando para recibir un valor. Una vez que se llama al método Navigator.pop de la página Acerca de , devuelve un valor a la variable _gratitudeResponse de la página de inicio . El uso de la palabra clave await es una característica potente y sencilla de implementar.



Uso de la ruta del navegador con nombre

Una forma alternativa de usar Navigator es hacer referencia a la página a la que está navegando por el nombre de la ruta. El nombre de la ruta comienza con una barra inclinada y luego viene el nombre de la ruta. Por ejemplo, el nombre de ruta de la página Acerca de es '/acerca de'. La lista de rutas está integrada en el widget MaterialApp(). Las rutas tienen un Map of String y WidgetBuilder donde String es el nombre de la ruta, y WidgetBuilder tiene un constructor para construir el contenido de la ruta por el nombre de clase (Acerca de) de la página para abrir.

```
rutas: <String, WidgetBuilder>{ '/acerca de':  
    (contexto BuildContext) => Acerca de(), '/gratitud': (contexto  
    BuildContext) => Agradecimiento(), },
```

Para llamar a la ruta, se llama al método Navigator.pushNamed() pasando dos argumentos. El primer argumento es el contexto y el segundo es el nombre de la ruta .

```
Navigator.pushNamed(contexto, '/acerca de');
```

USO DE ANIMACIÓN DE HÉROES

El widget Hero es una excelente animación lista para usar que transmite la acción de navegación de un widget que se desplaza de una página a otra. La animación del héroe es una transición de elementos compartidos (animación) entre dos páginas diferentes.

Para visualizar la animación, imagina ver a un superhéroe volando en acción. Por ejemplo, tiene una lista de entradas de diario con una foto en miniatura, el usuario selecciona una entrada y ve la transición de la miniatura de la foto a la página de detalles moviéndose y creciendo a tamaño completo. La miniatura de la foto es el superhéroe y, cuando se toca, entra en acción moviéndose de la página de lista a la página de detalles y aterriza perfectamente en la ubicación correcta en la parte superior de la página de detalles que muestra la foto completa. Cuando se descarta la página de detalles, el widget Hero vuelve a la página, la posición y el tamaño originales. En otras palabras, la animación muestra la miniatura de la foto moviéndose y creciendo desde la página de lista hasta la página de detalles, y una vez que se descarta la página de detalles, la animación y el tamaño se invierten. El widget Hero tiene todas estas funciones integradas; no hay necesidad de escribir código personalizado para manejar el tamaño y la animación entre páginas.

Para continuar con el escenario anterior, envuelve el widget Imagen de la página de lista como un elemento secundario del widget Hero y asigna un nombre de propiedad de etiqueta . Repita los mismos pasos para la página de detalles y asegúrese de que el valor de la propiedad de la etiqueta sea el mismo en ambas páginas.

```
// Página de lista  
Héroe  
    (etiqueta: 'foto1', niño:  
    Imagen (imagen:  
        AssetImage ("assets/images/coffee.png"), ), ),
```

```
// página de detalles  
Héroe(
```

```
etiqueta: 'foto1', hijo:  
Contenedor( hijo:  
    Imagen( imagen:  
        ImagenActivo("activos/imágenes/café.png"), ), ), ),
```

El widget secundario de héroe está marcado para la animación de héroe. Cuando el navegador empuja o abre una ruta de página, se reemplaza todo el contenido de la pantalla. Esto significa que durante la transición de la animación, el widget Hero no se muestra en la posición original en las rutas nueva y antigua, sino que se mueve y cambia de tamaño de una página a otra. Cada etiqueta de héroe debe ser única y coincidir tanto en la página de origen como en la de destino.

PRUÉBALO Creación de la aplicación Hero Animation

En este ejemplo, el widget Hero tiene un ícono como elemento secundario envuelto en un GestureDetector. También se podría usar un InkWell en lugar del GestureDetector para mostrar una animación de toque de material. El widget InkWell es un componente material que responde a los gestos táctiles mostrando un efecto de salpicadura (onda).

Profundizará en GestureDetector e InkWell en el Capítulo 11, “Aplicación de la interactividad”. Cuando se toca el ícono , se llama a Navigator.push para navegar a la pantalla de detalles, llamada Fly.

Dado que la animación del widget Hero es como un superhéroe volando, la pantalla de detalles se llama Fly.

1. Cree un nuevo proyecto de Flutter y asignele el nombre ch8_hero_animation. Nuevamente, puede seguir las instrucciones del Capítulo 4. Para este proyecto, necesita crear solo la carpeta de páginas .
2. Abra el archivo home.dart y agregue al cuerpo un SafeArea con Padding como elemento secundario.

```
cuerpo: SafeArea (hijo:  
    relleno (,),
```

3. En el elemento secundario Padding, agregue el widget GestureDetector() , que creará a continuación.

```
cuerpo: SafeArea(  
    niño: Relleno (relleno:  
        const EdgeInsets.all (16.0), niño: GestureDetector (,  
            ),  
            ),
```

4. Agregue al niño GestureDetector un widget Hero con la etiqueta 'format_paint'; la etiqueta puede ser cualquier identificación única. El elemento secundario del widget Hero es un ícono format_paint con color verde claro y un valor de tamaño de 120,0 píxeles. Tenga en cuenta que podría haber usado un widget InkWell() en lugar de GestureDetector(). El widget InkWell() muestra una respuesta de bienvenida cuando se toca, pero el widget GestureDetector() no muestra la respuesta táctil. Para la propiedad onTap , llame a Navigator.push para abrir la página de detalles llamada Fly.

```
GestureDetector (hijo:  
    Héroe (etiqueta:  
        'format_paint',
```

```
child:  
    Icon( Icons.format_paint,  
    color: Colors.lightGreen, size:  
    120.0, ), ), onTap:  
    ()  
  
{ Navigator.push( context,  
  
    MaterialPageRoute(builder: (context) => Fly()), ), },
```

5. En la parte superior de la página home.dart , importe la página fly.dart que creará a continuación.

```
importar 'volar.dardo';
```

Aquí está el código fuente completo del archivo Home.dart :

```
importar 'paquete: flutter/material.dart'; importar 'volar.dardo';  
  
class Home extiende StatelessWidget {  
    @anular  
    Creación de widgets (contexto BuildContext) { return  
        Scaffold (  
            barra de aplicaciones: barra de aplicaciones (  
                título: Texto ('Animación de héroe'),  
            ),  
            cuerpo: SafeArea(  
                niñ o: relleno (  
                    relleno: EdgeInsets.all(16.0), hijo:  
                    GestureDetector(  
                        niñ o: Héroe  
                            (etiqueta: 'format_paint', niñ o:  
                            Icono  
                                (Iconos.format_paint, color:  
                                Colors.lightGreen, tamaño: 120.0,  
                            ),  
                        ),  
                    ),  
                ),  
                onTap: ()  
                { Navigator.push( contexto,  
  
                    MaterialPageRoute(constructor: (contexto) => Fly()), ); },  
  
                ),  
            ),  
        );  
    }  
}
```

6. Cree un nuevo archivo llamado fly.dart en la carpeta lib/pages . Dado que esta página solo muestra información, cree una clase StatelessWidget llamada Fly. Para el cuerpo, agregue el SafeArea habitual con un conjunto secundario al widget Hero() .

```
cuerpo: SafeArea( niño:  
    Héroe(), ),
```

7. Para calcular el ancho del ícono, después de Widget build(BuildContext context) {, agregue un variable doble llamada _width establecida por MediaQuery.of(context).size.shortestSide / 2. La propiedad shortestSide devuelve el menor ancho o alto de la pantalla, y se divide por dos para que sea la mitad del tamaño.

La razón por la que calcula el ancho es solo para cambiar el tamaño del ancho del ícono según el tamaño y la orientación del dispositivo. Si usó una Imagen en su lugar, este cálculo no sería necesario; se puede hacer usando BoxFit.fitWidth.

```
double _width = MediaQuery.of(context).size.shortestSide / 2;
```

8. Agregue al widget Hero una etiqueta de 'format_paint' con un Contenedor para el niño. Tenga en cuenta que para que el widget Hero funcione correctamente, la etiqueta debe tener el mismo nombre que le dio en el widget Hero secundario de GestureDetector en el archivo home.dart . El contenedor secundario es un ícono de formato_pintura con color verde claro y un valor de tamaño de la variable de ancho . Para la propiedad de alineación del contenedor , use Alignment.bottomCenter. Puede experimentar usando diferentes valores de Alineación para ver las variaciones de animación del héroe en el trabajo.

```
Héroe  
(etiqueta: 'format_paint', niño:  
Contenedor (  
    alineación: Alignment.bottomCenter, child:  
  
Icon( Icons.format_paint, color:  
    Colors.lightGreen, size: _width, ),  
  
,  
)
```

Aquí está el código fuente completo del archivo Fly.dart :

```
importar 'paquete: flutter/material.dart';  
  
clase Fly extiende StatelessWidget { @override  
  
    Creación de widgets (contexto BuildContext) { double  
        _width = MediaQuery.of(context).size.shortestSide / 2;  
  
        andamio de vuelta(  
            appBar: AppBar(título:  
                Texto('Volar'), ),
```

```
cuerpo:  
  SafeArea( child:  
    Hero( tag: 'format_paint',  
      child: Container(  
        alineación: Alignment.bottomCenter, child:  
  
          Icon( Icons.format_paint,  
            color: Colors.lightGreen, size:  
              _width, ), ), ), );  
  
})
```



Cómo funciona

La animación de héroe es una poderosa animación incorporada para transmitir una acción al animar automáticamente un widget de una página a otra al tamaño y posición correctos. En la página de inicio, declara un GestureDetector con el widget Hero como widget secundario . El widget Hero establece un ícono como elemento secundario. onTap llama al método Navigator.push() , que navega a la página Fly . Todo lo que necesita hacer en la página Fly es declarar el widget al que está animando como elemento secundario del widget Hero . Cuando navega de regreso a la página de inicio, el héroe anima el ícono a la posición original.

USO DE LA BARRA DE NAVEGACIÓN INFERIOR

BottomNavigationBar es un widget de Material Design que muestra una lista de BottomNavigationBarBarItems que contiene un ícono y un título en la parte inferior de la página (Figura 8.1). Cuando se selecciona BottomNavigationBarItem , se crea la página adecuada.

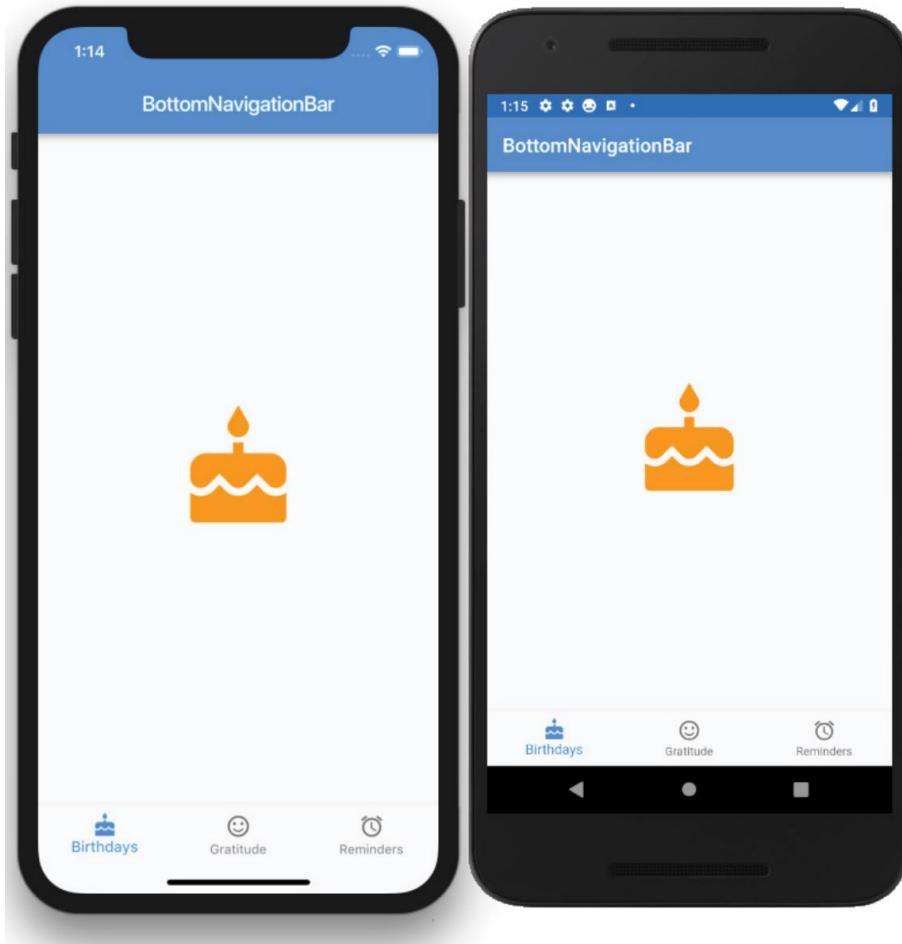


FIGURA 8.1: BottomNavigationBar final con íconos y títulos

PRUÉBELO Creación de la aplicación BottomNavigationBar En este ejemplo, BottomNavigationBar tiene tres BottomNavigationBarItems que reemplazan la página actual con la seleccionada. Hay diferentes formas de mostrar la página seleccionada; utiliza una clase Widget como variable.

1. Cree un nuevo proyecto Flutter y asigne el nombre ch8_bottom_navigation_bar. Como siempre, puede seguir las instrucciones del Capítulo 4. Para este proyecto, necesita crear solo la carpeta de páginas .

2. Abra el archivo home.dart y agregue al cuerpo un SafeArea con Padding como elemento secundario.

```
cuerpo: SafeArea( hijo:  
    relleno (.),,
```

3. En el elemento secundario Padding, agregue la variable Widget _currentPage, que creará a continuación. Tenga en cuenta que esta vez está utilizando una clase Widget para crear la variable _currentPage que contiene cada página seleccionada, ya sea la clase Gratitude, Reminders o Birthdays StatelessWidget .

```
cuerpo: SafeArea(  
    niño: Relleno (relleno:  
        const EdgeInsets.all (16.0), niño: página actual,  
        -  
    ), ),
```

4. Agregue a la propiedad bottomNavigationBar de Scaffold un widget BottomNavigationBar . Para la propiedad currentIndex , use la variable _currentIndex , que se creará más adelante.

```
bottomNavigationBar: BottomNavigationBar(  
    índiceActual: _índiceActual,
```

La propiedad items es una lista de BottomNavigationBarItems. Cada BottomNavigationBarItem toma una propiedad de ícono y una propiedad de título .

5. Agregue a la propiedad de elementos tres BottomNavigationBarItem s con torta de íconos, sentiment_satisfied y access_alarm. Los títulos son 'Cumpleaños', 'Gratitud' y 'Recordatorios'.

```
elementos: [  
    BottomNavigationBarItem(  
        ícono: Icon(Icons.pastel), título:  
        Text('Cumpleaños'), ),
```

6. Para la propiedad onTap , la devolución de llamada devuelve el índice actual del elemento activo. Asigne un nombre a la variable índiceSeleccionado.

```
onTap: (índice seleccionado) => _cambiar página (índice seleccionado),
```

Aquí está el código completo de BottomNavigationBar :

```
bottomNavigationBar: BottomNavigationBar(índice actual:  
    _índiceactual, elementos: [ Elemento de la  
    barra de  
        navegación inferior( ícono:  
        Icon(Icons.cake), título:  
        Text('Cumpleaños'), ),
```

```
BottomNavigationBarItem( icon:
    Icon(Icons.sentiment_satisfied), title: Text('Gratitude'), ),
BottomNavigationBarItem( icon:
    Icon(Icons.access_alarm), title:
    Text('Reminders'), ), ], onTap: (selectedIndex )
=> _cambiarPágina(indice seleccionado), ),
```

7. Agregue el método _changePage(int selectedIndex) después de Scaffold(). El _changePage()

El método acepta un valor int del índice seleccionado. El índice seleccionado se usa con el método setState() para establecer las variables _currentIndex y _currentPage .

_currentIndex es igual a selectedIndex, y _currentPage es igual a la página de List _listPages que corresponde al índice seleccionado .

Es importante tener en cuenta que la variable Widget _currentPage muestra cada página seleccionada sin necesidad de un widget Navigator . Este es un gran ejemplo del poder de personalizar widgets según sus necesidades.

```
void _cambiarPágina(int índiceSeleccionado) { setState(() {
  _índiceActual =
  índiceSeleccionado; _páginaActual =
  _listarPáginas[índiceSeleccionado]; })}
```

8. En la primera línea después de la definición de la clase _HomeState , agregue las variables _currentIndex, _listPages y _currentPage. La lista _listPages contiene el nombre de clase de cada página .

```
int _índiceActual = 0;
Lista _listarPáginas = Lista();
Widget _páginaactual;
```

9. Anule initState() para agregar cada página a la lista _listPages e inicialice _currentPage con la página Birthdays() . Tenga en cuenta el uso de la notación en cascada; los puntos dobles le permiten realizar una secuencia de operaciones en el mismo objeto.

```
@anular
void initState() {
  super.initState();

_listPages ..add(Cumpleaños()) ..add(Agradecimiento()) ..add(Recordatorios()); _currentPage = Cumpleaños(); }
```

10. Agregue a la parte superior del archivo home.dart las importaciones para cada página que se creará a continuación.

```
importar 'paquete: flutter/material.dart'; import 'gratitud.dart';
importar 'recordatorios.dart'; importar
'cumpleaños.dart';
```

Aquí está todo el archivo home.dart :

```
importar 'paquete: flutter/material.dart'; import 'gratitud.dart';
importar 'recordatorios.dart'; importar
'cumpleaños.dart';

class Home extiende StatefulWidget {
    @anular
    _HomeState createState() => _HomeState();
}

class _HomeState extiende State<Home> { int
    _currentIndex = 0;
    Lista _listaPaginas = Lista();
    Widget _páginaactual;

    @anular
    void initState() {
        super.initState();

        _listPages ..add(Cumpleaños()) ..add(Agradecimiento()) ..add(Recordatorios()); _currentPage = Cumpleaños();
    }

    void _cambiarPágina(int índiceSeleccionado) { setState(() {
        _índiceActual =
            índiceSeleccionado; _páginaActual =
            _listarPáginas[índiceSeleccionado]; });
    }

    @anular
    Creación de widgets (contexto BuildContext) { return
        Scaffold (
            appBar: AppBar(título:
                Text('BottomNavigationBar'), ), cuerpo: SafeArea(
                    niño: relleno (
                        relleno: EdgeInsets.all(16.0), hijo:
                            _currentPage, ),
            ),
        ),
    }
}
```

```
bottomNavigationBar: BottomNavigationBar( currentIndex:  
    _currentIndex, items:  
  
    [ BottomNavigationBarItem( icon:  
        Icon(Icons.cake), title:  
        Text('Cumpleaños'), ),  
  
        BottomNavigationBarItem( icon:  
        Icon(Icons.sentiment_satisfied), title: Text('Gratitud  
' ), ), BottomNavigationBarItem( icon:  
        Icon(Icons.access_alarm), title:  
        Text('Recordatorios'), ), ], onTap:  
        (selectedIndex) =>  
  
        _changePage(selectedIndex), ), );  
  
})
```

11. Cree tres páginas StatelessWidget y llámelas Cumpleaños, Gratitud y Recordatorios. Cada página tendrá un Scaffold con Center() para el cuerpo. El centro secundario es un ícono con un valor de tamaño de 120,0 píxeles y una propiedad de color . Estos son los tres archivos de Dart, birthdays.dart, thanks.dart y Reminders.dart:

```
// cumpleaños.dart  
import 'paquete:flutter/material.dart';  
  
clase Cumpleaños extiende StatelessWidget  
{ @override  
Widget build (contexto BuildContext) { return  
Scaffold (body:  
Center ( child:  
Icon ( Icons.cake,  
size: 120.0,  
color:  
Colors.orange, ), );  
  
})  
  
// gratitud.dart import  
'paquete:flutter/material.dart';  
  
clase Gratitud extiende StatelessWidget { @override  
Widget build(BuildContext context) { return  
Scaffold( body:  
Center( child:  
Icon( Icons.sentiment_satisfied,
```

```
        tamaño: 120.0,  
        color: Colors.lightGreen, ), ), );  
  
    })  
  
// recordatorios.dart  
import 'paquete:flutter/material.dart';  
  
Recordatorios de clase extiende StatelessWidget  
{ @override  
Widget compilación (contexto BuildContext)  
{ return Scaffold  
    (body: Center  
        (child: Icon  
            (Icons.access_alarm,  
            size: 120.0,  
            color: Colors.purple, ), ), );  
  
    })
```



Cómo funciona

La propiedad de elementos `BottomNavigationBar` tiene una lista de tres `BottomNavigationBarItems`. Para cada `BottomNavigationBarItem`, establece una propiedad de ícono y una propiedad de título . el fondo

NavigationBar onTap pasa el valor de índice seleccionado al método _changePage . El método _changePage usa setState() para configurar _currentIndex y _currentPage para que se muestren. _currentIndex establece el BottomNavigationBarItem seleccionado y _currentPage establece la página actual para que se muestre desde la lista _listPages.

USO DE LA BARRA INFERIOR

El widget BottomAppBar se comporta de manera similar a BottomNavigationBar, pero tiene una muesca opcional en la parte superior. Al agregar un FloatingActionButton y habilitar la muesca, la muesca proporciona un agradable efecto 3D, por lo que parece que el botón está empotrado en la barra de navegación (Figura 8.2).

Por ejemplo, para habilitar la muesca, establece la propiedad de forma BottomAppBar en una clase de forma con muesca como la clase CircularNotchedRectangle() y establece la propiedad Scaffold floatingActionButtonLocation en FloatingActionButtonLocation.endDocked o center Docked. Agregue a la propiedad Scaffold floatingActionButton un widget de FloatingActionButton y el resultado muestra el FloatingActionButton incrustado en el widget de la barra BottomApp , que es la muesca.

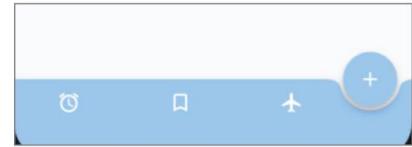


FIGURA 8.2: BottomAppBar con FloatingActionButton incrustado creando una muesca

```
BottomAppBar(forma:  
    CircularMuescasRectángulo(),  
)  
  
flotanteActionButtonLocation: FloatingActionButtonLocation.endAcoplado, flotanteActionButton:  
FloatingActionButton(  
    hijo: Icono(Iconos.añadir), ),
```

PRUÉBALO Creación de la aplicación BottomAppBar

En este ejemplo, BottomAppBar tiene una fila como elemento secundario con tres IconButton para mostrar elementos de selección. El objetivo principal es usar un FloatingActionButton para acoplarlo a BottomAppBar con una muesca. La muesca está habilitada por la propiedad de forma BottomAppBar establecida en CircularNotchedRectangle().

1. Cree un nuevo proyecto de Flutter y asigne el nombre ch8_bottom_app_bar . Nuevamente, puede seguir las instrucciones del Capítulo 4. Para este proyecto, solo necesita crear la carpeta de páginas .

2. Abra el archivo home.dart y agregue al cuerpo un SafeArea con un Contenedor como elemento secundario.

```
cuerpo: SafeArea (hijo:  
    Contenedor (),  
) ,
```

3. Agregue un widget BottomAppBar() a la propiedad Scaffold bottomNavigationBar .

```
bottomNavigationBar: BottomAppBar(),
```

4. Para habilitar la muesca, establezca dos propiedades.

Primero establezca la propiedad de forma BottomAppBar en CircularNotchedRectangle(). Establezca la propiedad de color en Colors.blue.shade200 y agregue una fila como elemento secundario.

A continuación, configure la ubicación del botón de acción flotante, que manejará en el paso 7.

```
bottomNavigationBar: BottomAppBar( color:  
    Colors.blue.shade200, forma:  
    CircularNotchedRectangle(), child: Row(, ),
```

5. Continúe agregando a la Fila una propiedad mainAxisAlignment como MainAxisAlignment.space Around. La constante spaceAround permite que los IconButton tengan un espacio uniforme entre ellos.

```
bottomNavigationBar: BottomAppBar( color:  
    Colors.blue.shade200, forma:  
    CircularNotchedRectangle(), child:  
  
    Row( mainAxisAlignment: MainAxisAlignment.spaceAround, children:  
        <Widget>[ ], ), ),
```

6. Agregue tres IconButton a la lista de filas secundarias . Después del último IconButton, agregue un Divider() para agregar un espacio uniforme a la derecha, ya que FloatingActionButton está anclado en el lado derecho de BottomAppBar. En lugar de un Divider(), podría haber usado un Contenedor con una propiedad de ancho .

```
bottomNavigationBar: BottomAppBar( color:  
    Colors.blue.shade200, forma:  
    CircularNotchedRectangle(), child:  
  
    Row( mainAxisAlignment: MainAxisAlignment.spaceAround, children:  
        <Widget>[ IconButton( icon:  
  
            Icon(Icons.access_alarm), color: Colors.white ,  
            onPressed: (){}, ),  
            IconButton( icon:  
  
                Icon(Icons.bookmark_border), color: Colors.white,  
                onPressed: (){}, ),  
                IconButton( icon:  
  
                    Icon(Icons.flight),  
                    color: Colors.white, onPressed: ()  
                    {}, ), Divider(), ], ), ),
```

7. Establezca la ubicación de la muesca de la propiedad floatingActionButtonLocation en FloatingActionButtonLocation.endDocked. También puede establecerlo en centerDocked.

ubicación del botón de acción flotante: ubicación del botón de acción flotante. enddocked,

8. Agregue un Botón de acción flotante a la propiedad Botón de acción flotante .

botón de acción flotante: Botón de acción flotante (Color de fondo: Colors.blue.shade200, onPressed: () {}, child: Icon(Icons.add),),

Aquí está el código fuente completo del archivo home.dart :

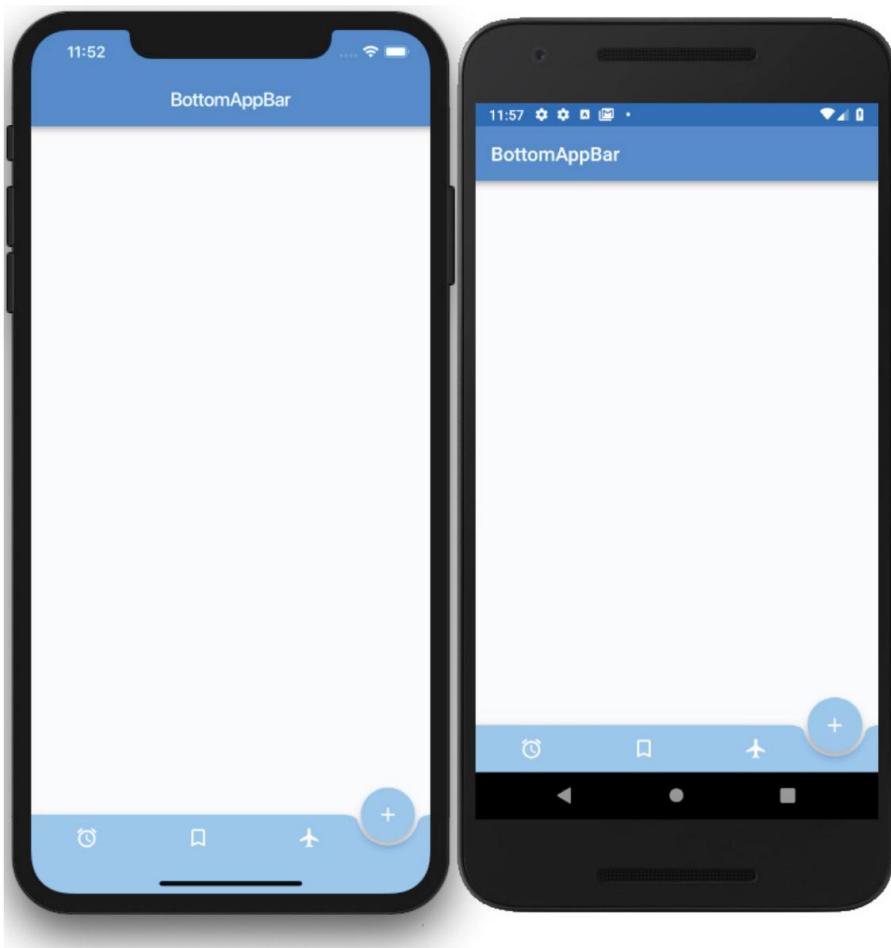
```
importar 'paquete: flutter/material.dart';

class Home extiende StatefulWidget {
    @anular
    _HomeState createState() => _HomeState();
}

class _HomeState extiende State<Home> {
    @anular
    Creación de widgets (contexto BuildContext) { return
        Scaffold (
            barra de aplicaciones: barra de aplicaciones (
                título: Text('BottomAppBar'), ), cuerpo:
                SafeArea( child:
                    Container(),
                ),
            bottomNavigationBar: BottomAppBar( color:
                Colors.blue.shade200, forma:
                CircularNotchedRectangle(), child: Row(
                    mainAxisAlignment: MainAxisAlignment.spaceAround, children:
                    <Widget>[ IconButton( icon:
                        Icon(Icons.access_alarm), color: Colors.white,
                        onPressed: (){}, ),
                    IconButton( icon:
                        Icon(Icons.bookmark_border), color : Colors.white,
                        onPressed: (){}, ),
                    IconButton( icon:
                        Icon(Icons.flight),
                        color: Colors.white, onPressed: () {}
                        {}, ), Divider(),
                ],
            ],
        );
    }
}
```

);

}}



Cómo funciona

Para habilitar la muesca, se deben establecer dos propiedades para el widget Scaffold . El primero es usar BottomAppBar con la propiedad de forma establecida en CircularNotchedRectangle(). El segundo es establecer la propiedad floatActionButtonLocation en FloatingActionButtonLocation.endDocked o centerDocked.

USO DEL TABBAR Y TABBARVIEW

El widget TabBar es un widget de Material Design que muestra una fila horizontal de pestañas. La propiedad de pestañas toma una lista de widgets y agrega pestañas utilizando el widget de pestaña . En lugar de usar el widget de pestañas , puedes crear un widget personalizado que muestre el poder de Flutter. La pestaña seleccionada se marca con una línea de selección inferior.

El widget TabBarView se usa junto con el widget TabBar para mostrar la página de la pestaña seleccionada. Los usuarios pueden deslizar hacia la izquierda o hacia la derecha para cambiar el contenido o tocar cada pestaña.

Tanto el widget TabBar (Figura 8.3) como TabBarView toman una propiedad de controlador de TabController. El controlador de pestañas es responsable de sincronizar las selecciones de pestañas entre un TabBar y un TabBarView. Dado que TabController sincroniza las selecciones de pestañas, debe declarar SingleTickerProviderStateMixin a la clase. En el Capítulo 7, "Aregar animación a una aplicación", aprendió cómo implementar la clase Ticker que está controlada por el informe Schedule Binding.scheduleFrameCallback una vez por cuadro de animación. Está tratando de sincronizar la animación para que sea lo más fluida posible.

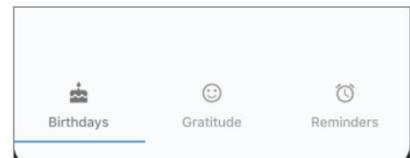


FIGURA 8.3: TabBar en la propiedad Scaffold bottomNavigationBar

PRUÉBALO Creación de la aplicación TabBar y TabBarView

En este ejemplo, el widget TabBar es el elemento secundario de una propiedad bottomNavigationBar . Esto coloca la barra de pestañas en la parte inferior de la pantalla, pero también puede colocarla en la barra de aplicaciones o en una ubicación personalizada. Cuando usa una TabBar en combinación con TabBarView, una vez que se selecciona una pestaña , muestra automáticamente el contenido apropiado. En este proyecto, el contenido está representado por tres páginas separadas. Creará las mismas tres páginas que creó en el proyecto BottomNavigationBar .

1. Cree un nuevo proyecto de Flutter y asignele el nombre ch8_tabbar. Una vez más, puede seguir las instrucciones del Capítulo 4. Para este proyecto, necesita crear solo la carpeta de páginas .
2. Abra el archivo home.dart y agregue al cuerpo un SafeArea con TabBarView como elemento secundario. El La propiedad del controlador TabBarView es una variable TabController llamada _tabController. Agregue a la propiedad secundaria TabBarView las páginas Birthdays(), Gratitude() y Reminders() que creará en el paso 3.

cuerpo: SafeArea
(hijo: TabBarView (

```
controlador: _tabController, niños: [
```

```
    cumpleaños(),
    Gratitud(),
    Recordatorios(), ], ), ),
```

3. Esta vez, primero creará las páginas a las que está navegando. Al igual que en la aplicación `BottomNavigationBar`, cree tres páginas StatelessWidget y llámelas Cumpleaños, Gratitud y Recordatorios. Cada página tiene un Scaffold con Center() para el cuerpo. El elemento secundario del centro es un ícono con un tamaño de 120,0 píxeles y un color.

Los siguientes son los tres archivos de Dart, birthdays.dart, gratitud.dart y recordatorios.dart:

```
// cumpleaños.dart import
'paquete:flutter/material.dart';

cumpleaños clase extiende StatelessWidget {
  @anular

  Widget compilación (contexto BuildContext) { return
    Scaffold (cuerpo: Center
      (
        niño: ícono (
          Iconos.cake,
          tamaño: 120.0,
          color: Colors.orange, ),

      ),
    );
  }
}

// gratitud.dart import
'paquete:flutter/material.dart';

clase Gratitud extiende StatelessWidget {
  @anular

  Widget compilación (contexto BuildContext) { return
    Scaffold (cuerpo: Center
      (
        niño: ícono
          (Iconos.sentiment_satisfied, tamaño:
            120.0, color:
            Colors.lightGreen, ),

      ),
    );
  }
}

// recordatorios.dart import
'paquete:flutter/material.dart';
```

```
Recordatorios de clase extiende StatelessWidget
{ @override
Widget compilación (contexto BuildContext)
{ return Scaffold
  (body: Center
  (child: Icon
    (Icons.access_alarm,
    size: 120.0,
    color: Colors.purple, ), ), );
}

}
```

4. Importe cada página en el archivo home.dart .

```
importar 'paquete: flutter/material.dart'; importar
'cumpleaños.dart'; import
'gratitud.dart'; importar
'recordatorios.dart';
```

5. Declare TickerProviderStateMixin a la clase _HomeState agregando con
TickerProviderStateMixin. El argumento vsync de AnimationController lo usará.

```
class _HomeState extiende State<Home> con SingleTickerProviderStateMixin {...}
```

6. Declare una variable TabController con el nombre de _tabController. Anule el método initState() para
inicializar _tabController con el argumento vsync y un valor de longitud de 3. Esto hace referencia a
vsync , es decir, esta referencia de la clase _HomeState . La longitud representa el número de
pestañas a mostrar. Agregue un Listener al _tabController para detectar cuando se cambia una pestaña .
Luego anule el método dispose() para cuando la página se cierre para desechar correctamente el
_tabController.

Tenga en cuenta que en el método _tabChanged verifica indexIsChanging antes de mostrar qué
pestaña está tocada. Si no verifica indexIsChanging, el código se ejecuta dos veces.

```
class _HomeState extiende State<Home> con SingleTickerProviderStateMixin {
  TabController _tabController;

  @anular
  void initState() {
    super.initState();

    _tabController = TabController(vsync: this, longitud: 3);
    _tabController.addListener(_tabChanged);
  }

  @anular
  anular disponer () {
    super.dispose();

  @anular
  anular disponer()
  { _tabController.dispose();
```

```
super.dispose(); }

_tabController.dispose(); }

vacío _tabChanged () {
    // Comprobar si el índice del controlador de pestañas está cambiando; de lo contrario, recibimos el aviso dos veces if
    (_tabController.indexIsChanging) { print('tabChanged: $
        {_tabController.index}'); }

}
```

7. Agregue un TabBar como elemento secundario de la propiedad bottomNavigationBar Scaffold .

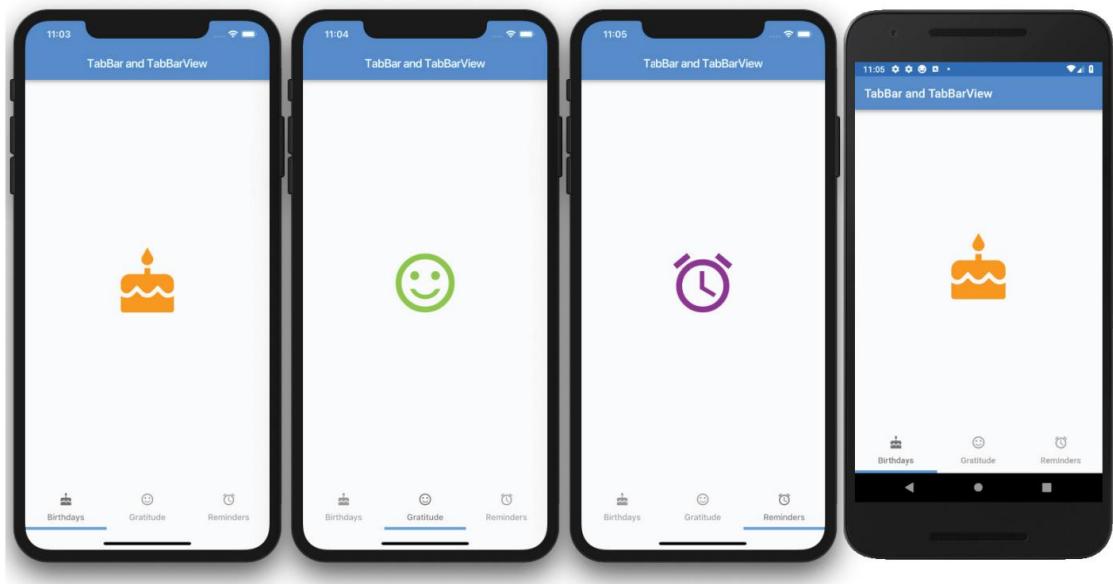
```
bottomNavigationBar: SafeArea(
    hijo: TabBar(), ),
```

8. Pase el _tabController para la propiedad del controlador TabBar . Personalicé labelColor y unselectedLabelColor usando, respectivamente, Colors.black54 y Colors.black38, pero síntetice libre de experimentar usando diferentes colores.

```
bottomNavigationBar: SafeArea( child:
    TabBar(
        controlador: _tabController, labelColor:
        Colors.black54, unselectedLabelColor:
        Colors.black38, ), ),
```

9. Agregue tres widgets de pestañas a la lista de widgets de pestañas . Personaliza el icono y el texto de cada pestaña .

```
bottomNavigationBar: SafeArea( child:
    TabBar(
        controlador: _tabController, labelColor:
        Colors.black54, unselectedLabelColor:
        Colors.black38, pestañas: [ Tab( icon: Icon(Icons.cake),
            text:
                'Cumpleaños', ), Tab( icon:
                    Icon(Icons.sentiment_satisfied),
            texto :
                'Agradecimiento', ), Tabulador( icon:
                    Icon(Icons.access_alarm),
            texto:
                'Recordatorios', ), ], ), ),
```



Cómo funciona

Cuando TabBar y TabBarView se usan juntos, la página asociada correcta se carga automáticamente.

Cuando el usuario desliza el TabBarView hacia la izquierda o hacia la derecha, se desplaza a la página correcta y selecciona la pestaña correspondiente en el TabBar. Todas estas potentes funciones están integradas; no se necesita codificación personalizada.

¿Cómo sabe qué página pertenece a qué pestaña? TabController es responsable de sincronizar las selecciones de pestañas entre TabBar y TabBarView . Dado que TabController sincroniza las selecciones de pestañas, debe declarar SingleTickerProviderStateMixin a la clase.

Tanto TabBar como TabBarView usan el mismo TabController. El TabController se inicia pasando un argumento vsync y un argumento de longitud . El argumento de longitud es el número de pestañas que se mostrarán.

Se agrega un TabController Listener opcional para escuchar los cambios de pestaña y tomar las medidas adecuadas si es necesario, tal vez guardando datos antes de que se cambie una pestaña . Cada pestaña se agrega a la lista de widgets de pestañas de TabBar al personalizar cada ícono y texto.

TabBarView es responsable de cargar la página adecuada cuando cambia la selección de pestañas . Las visitas a la página se enumeran como la propiedad secundaria del widget TabBarView .

USO DEL CAJÓN Y LISTVIEW

Quizás se pregunte por qué estoy cubriendo ListView en este capítulo de navegación. Bueno, funciona muy bien con el widget Cajón . Los widgets de ListView se utilizan con bastante frecuencia para seleccionar un elemento de una lista para navegar a una página detallada.

Drawer es un panel de Material Design que se desliza horizontalmente desde el borde izquierdo o derecho de Scaffold, la pantalla del dispositivo. Drawer se usa con la propiedad de cajón Scaffold (lado izquierdo) o la propiedad endDrawer (lado derecho). El cajón se puede personalizar para cada necesidad individual, pero generalmente tiene un encabezado para mostrar una imagen o información fija y un ListView para mostrar una lista de páginas navegables. Por lo general, se utiliza un cajón cuando la lista de navegación tiene muchos elementos.

Para configurar el encabezado del cajón , tiene dos opciones integradas, UserAccountsDrawerHeader o DrawerHeader. UserAccountsDrawerHeader está diseñado para mostrar los detalles de usuario de la aplicación configurando las propiedades currentAccountPicture, accountName, accountEmail, otherAccountsPictures y de decoración .

```
// Detalles del usuario
UserAccountsDrawerHeader (imagen
    de cuenta actual: ícono (iconos.cara), nombre de cuenta: texto
    ('Sandy Smith'), correo electrónico de cuenta: texto
    ('sandy.smith@nombrededominio.com'), otras imágenes de cuentas:
    <Widget>[ ícono (iconos).bookmark_border], ,
    decoración: BoxDecoration( imagen:

    DecorationImage( imagen:
        AssetImage('assets/images/
            home_top_mountain.jpg'), ajuste: BoxFit.cover, ), ), ),
```

DrawerHeader está diseñado para mostrar información genérica o personalizada configurando el relleno, el elemento secundario, la decoración y otras propiedades.

```
// Información genérica o personalizada
CajónEncabezado(
    relleno: EdgeInsets.zero, child:
    Icon(Icons.face), decoración:
    BoxDecoration(color: Colors.blue), ),
```

El constructor ListView estándar le permite crear rápidamente una breve lista de elementos. El próximo capítulo profundizará en cómo usar ListView. Consulte la Figura 8.4.

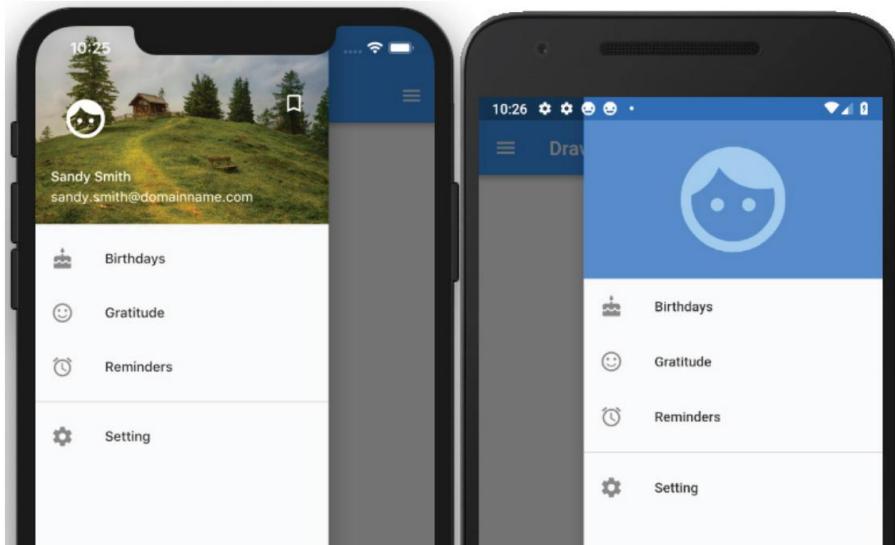


FIGURA 8.4: Cajón y ListView

PRUÉBALO Creación de la aplicación Drawer

En este ejemplo, el cajón se agrega a la propiedad `cajón` o `cajón final` del andamio. Las propiedades del cajón y del cajón final deslizan el cajón de izquierda a derecha (`TextDirection.ltr`) o de derecha a izquierda (`TextDirection rtl`). En este ejemplo, agregaré tanto el cajón como el cajón final para mostrar cómo usar ambos. Usa `UserAccountsDrawerHeader` para la propiedad del cajón (lado izquierdo) y `DrawerHeader` para la propiedad `endDrawer` (lado derecho).

Utiliza `ListView` para agregar el contenido del cajón y `ListTile` para alinear fácilmente el texto y los íconos para la lista del menú. En este proyecto, usa el constructor `ListView` estándar ya que tiene una pequeña lista de elementos de menú.

1. Cree un nuevo proyecto de Flutter y asígnele el nombre `ch8_drawer`, siguiendo las instrucciones del Capítulo 4.

Para este proyecto, debe crear las páginas, los widgets y las carpetas de activos/ímágenes . Copie la imagen `home_top_mountain.jpg` en la carpeta `assets/images` .

2. Abra el archivo `pubspec.yaml` y, en `activos` , agregue la carpeta de imágenes .

```
# Para agregar activos a su aplicación, agregue una sección de activos, como esta: activos:
```

```
- activos/ímágenes/
```

3. Agregue los activos de la carpeta y las imágenes de la subcarpeta en la raíz del proyecto y luego copie el archivo `home_top_mountain.jpg` en la carpeta de imágenes .
4. Haga clic en el botón Guardar; dependiendo del editor que estés usando, automáticamente ejecuta los paquetes flutter get. Una vez finalizado, muestra un mensaje de Proceso finalizado con código de salida 0.
Si no ejecuta automáticamente el comando por ti, abre la ventana Terminal (ubicada en la parte inferior de tu editor) y escribe flutter packages get.
5. Cree las páginas a las que está navegando primero. Cree tres páginas StatelessWidget y llame ellos Cumpleaños, Gratitud y Recordatorios. Cada página tiene un Scaffold con Center() para el cuerpo. El centro secundario es un ícono con un valor de tamaño de 120,0 píxeles y una propiedad de color .

```
cumpleaños clase extiende StatelessWidget {  
  @anular  
  Widget compilación (contexto BuildContext) { return  
    Scaffold (cuerpo: Center  
    (  
      niño: ícono (  
        Iconos.cake,  
        tamaño: 120.0,  
        color: Colors.orange, ),  
  
      ),  
    );  
  }  
}
```

6. Agregue AppBar a Scaffold, que es necesario para volver a la página de inicio. El a continuación se muestran los tres archivos de Dart, birthdays.dart, thanks.dart y Reminders.dart:

```
// cumpleaños.dart import  
'paquete:flutter/material.dart';  
  
cumpleaños clase extiende StatelessWidget {  
  @anular  
  Creación de widgets (contexto BuildContext) { return  
    Scaffold (  
      appBar: AppBar(título:  
        Texto('Cumpleaños'), ), cuerpo: Centro(  
  
      niño: ícono (  
        Iconos.cake,  
        tamaño: 120.0,  
        color: Colors.orange, ),  
  
      ),  
    );  
  }  
}
```

```
// gratitud.dart import
'paquete:flutter/material.dart';

clase Gratitud extiende StatelessWidget { @override

    Creación de widgets (contexto BuildContext)
    { return Scaffold (
        appBar: AppBar( title:
            Text('Gratitude'), ), body:

        Center( child:

            Icon( Icons.sentiment_satisfied, size:
                120.0, color:
                Colors.lightGreen, ), ), );

    }

// recordatorios.dart
import 'paquete:flutter/material.dart';

Recordatorios de clase extiende StatelessWidget
{ @override

    Creación de widgets (contexto BuildContext)
    { return Scaffold (
        appBar: AppBar( title:
            Text('Reminders'), ), body:

        Center( child:

            Icon( Icons.access_alarm,
                size: 120.0,
                color: Colors.purple, ), ), );

    }

}}
```

7. Los widgets del cajón izquierdo y derecho comparten la misma lista de menú y usted la escribirá primero. Cree un nuevo archivo Dart en la carpeta de widgets . Haga clic con el botón derecho en la carpeta de widgets y luego seleccione Nuevo Archivo Dart, ingrese menu_list_tile.dart y haga clic en el botón Aceptar para guardar.

8. Importe las clases material.dart, birthdays.dart, gratitud.dart y recordatorios.dart (páginas). Agregue una nueva línea y luego comience a escribir st; se abre la ayuda de autocompletado, así que seleccione la abreviatura stful y asignele el nombre de MenuListTileWidget.

```
importar 'paquete: flutter/material.dart'; importar
'paquete: ch8_drawer/pages/birthdays.dart'; importar 'paquete:
ch8_drawer/pages/gratitude.dart'; importar 'paquete: ch8_drawer/
pages/reminders.dart';
```

9. La compilación del widget (contexto BuildContext) devuelve una columna. La lista de widgets secundarios de columna contiene varios ListTile que representan cada elemento del menú. Agregue un widget Divider con la propiedad de color establecida en Colors.grey antes del último widget ListTile .

```
@anular
Widget compilación (contexto BuildContext)
{ columna de
    retorno (hijos:

        <Widget>[ ListTile(), ListTile(),
ListTile(),
ListTile(),

        Divider(color: Colors.grey), ListTile(), ], );
}
```

10. Para cada ListTile, establezca la propiedad principal como un ícono y la propiedad de título como un texto. Para la propiedad onTap , primero llame a Navigator.pop() para cerrar el cajón abierto y luego llame a Navigator.push() para abrir la página seleccionada.

```
@anular
Widget build(BuildContext context) { return
Column( children:

    <Widget>[ ListTile( encabezado:
Icon(Icons.cake), title:
Text('Cumpleaños'), onTap: ()
{ Navigator.pop(context);
Navigator.
push( contexto,
MaterialPageRoute( builder: (contexto) =>
Cumpleaños(), ), ); }, ), ListTile( principal:
Icon(Icons.sentiment_satisfied),
title:
Text('Gratitud'), onTap: ()

{ Navigator.pop(contexto);
Navigator.push( contexto,
MaterialPageRoute( builder: (contexto)
=> Gratitud(), ), ), ListTile( encabezado: Icon(Icons.alarm), title: Text(' recordatorios'),
```

```
onTap: ()  
    { Navigator.pop(context);  
  
    Navigator.push( contexto,  
        MaterialPageRoute( constructor: (contexto)  
  
        => Recordatorios( ), ), ); }, ),  
Divider(color:  
    Colors.grey), ListTile(principal:  
    Icon(Icons.settings), title:  
    Text('Setting'),  
    onTap: ()  
  
    { Navigator.pop(context); }, ), ], );  
}
```

11. Cree un nuevo archivo Dart en la carpeta de widgets . Haga clic derecho en la carpeta de widgets y luego seleccione Nuevo Dart File, ingrese left_drawer.dart y haga clic en el botón Aceptar para guardar.

12. Importe la biblioteca material.dart y la clase menu_list_tile.dart .

```
importar 'paquete: flutter/material.dart'; importar  
'paquete: ch8_drawer/widgets/menu_list_tile.dart';
```

13. Agregue una nueva línea y luego comience a escribir st; se abre la ayuda de autocompletado. Seleccione la abreviatura stful y asignele el nombre LeftDrawerWidget.

```
class LeftDrawerWidget extiende StatelessWidget {  
    const LeftDrawerWidget({  
        Clave  
        clave, }): super(clave: clave);  
  
    @anular  
    Widget compilación (contexto BuildContext)  
    { return Drawer ();  
  
}}
```

14. La compilación Widget (contexto BuildContext) devuelve un Cajón. El niño del cajón es un ListView widget de lista de niños de un widget UserAccountsDrawerHeader y una llamada a la clase de widget const MenuListTileWidget() . Para llenar todo el espacio del cajón , establezca la propiedad de relleno ListView en EdgeInsets.zero.

```
@anular  
Widget compilación (contexto BuildContext)  
{ return Drawer  
    (hijo: ListView  
    (relleno: EdgeInsets.zero,
```

```
    niños: <Widget>[
      UserAccountsDrawerHeader(), const
      MenuListTileWidget(), ], ), );  
  
}
```

15. Para UserAccountsDrawerHeader, configure currentAccountPicture, accountName, account Email, otherAccountsPictures y las propiedades de decoración .

```
@anular
Widget build (contexto BuildContext) { return
  Drawer (hijo:
    ListView (padding:
      EdgeInsets.zero, children:
        <Widget>[ UserAccountsDrawerHeader(
          ImagenCuentaActual: Icono(Icons.cara,
            tamaño: 48.0,
            color:
              Colores.blanco, ),
          nombreCuenta: Texto('Sandy Smith'),
          correoCorreoCuenta: Texto('sandy.smith@nombrededominio.com'),
          otrasImágenesCuentas: <Widget> [
            Icono
              (Iconos.marcador_borde,
              color: Colores.blanco,
            ) ],
          decoración: BoxDecoration( imagen:
            DecorationImage( imagen:
              AssetImage('assets/images/home_top_mountain.jpg'), fit: BoxFit.cover, ), ), ),
          const
        MenuListTileWidget(), ], ), );  
  
}
```

16. Cree un nuevo archivo Dart en la carpeta de widgets . Haga clic con el botón derecho en la carpeta de widgets y luego seleccione Nuevo Archivo Dart, ingrese right_drawer.dart y haga clic en el botón Aceptar para guardar. Importe la biblioteca material.dart y la clase menu_list_tile.dart .

```
importar 'paquete: flutter/material.dart'; importar
'paquete: ch8_drawer/widgets/menu_list_tile.dart';
```

17. Agregue una nueva línea y luego comience a escribir st; se abre la ayuda de autocompletado. Seleccione la abreviatura de `st` y asignele el nombre de `RightDrawerWidget`.

```
class RightDrawerWidget extends StatelessWidget {  
  const RightDrawerWidget({  
    Key  
    clave, }) : super(clave: clave);  
  
  @override  
  Widget build(BuildContext context)  
  { return Drawer();  
  
}
```

18. La compilación `Widget (contexto BuildContext)` devuelve un Cajón. El elemento secundario `Drawer` es una lista de elementos secundarios `ListView` de un widget `DrawerHeader` y una llamada a la clase de widget `const MenuListTileWidget()`. Para llenar todo el espacio del cajón , establezca la propiedad de relleno `ListView` en `EdgeInsets.zero`.

```
@override  
Widget build (BuildContext context) { return  
  Drawer (child:  
    ListView (padding:  
      EdgeInsets.zero, children:  
      <Widget>[ DrawerHeader(),  
      const  
      MenuListTileWidget(), ], ), );  
  
}
```

19. Para `DrawerHeader`, establezca las propiedades de relleno, secundarias y decorativas .

```
@override  
Widget build(BuildContext context) { return  
  Drawer( child:  
    ListView( padding:  
      EdgeInsets.zero, children:  
      <Widget>[ DrawerHeader(  
  
        relleno: EdgeInsets.zero, child:  
        Icon( Icons.face,  
          size: 128.0,  
          color:  
          Colors.white54, ), decoration:  
  
        BoxDecoration(color: Colors.blue), ), const  
  
        MenuListTileWidget(), ], ), );  
  
}
```

20. Abra el archivo home.dart e importe material.dart, birthdays.dart, thanks.dart y recordatorios.clases de dardos .

```
importar 'paquete: flutter/material.dart'; importar  
'cumpleaños.dart'; import 'gratitud.dart';  
importar 'recordatorios.dart';
```

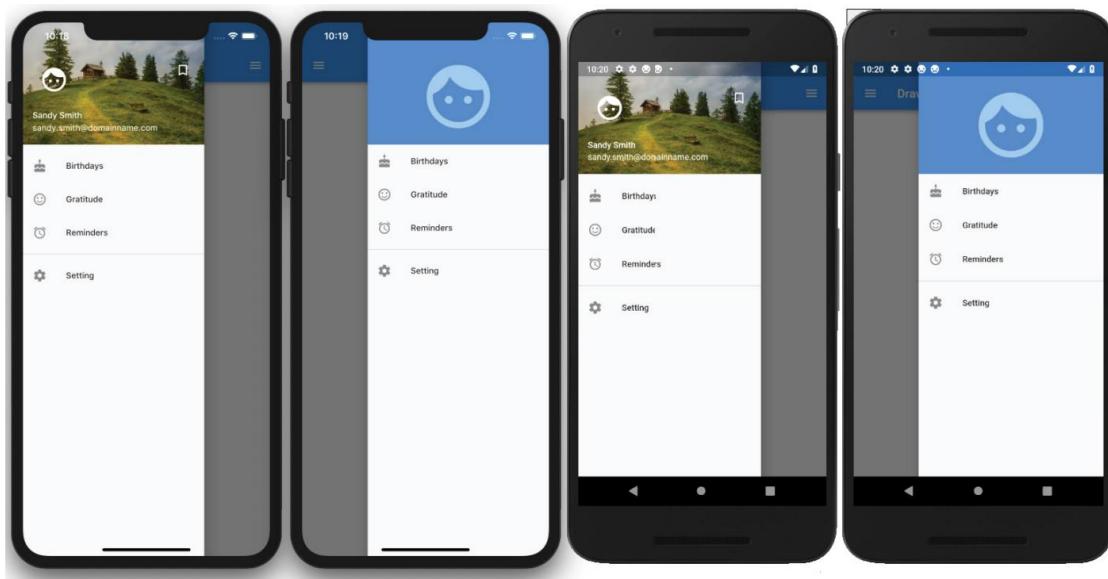
21. Agregue al cuerpo un SafeArea con un contenedor como elemento secundario.

```
cuerpo: SafeArea( hijo:  
    Contenedor(), ),
```

22. Agregue a la propiedad del cajón Scaffold una llamada a la clase de widget LeftDrawerWidget() y para el propiedad endDrawer una llamada a la clase de widget RightDrawerWidget() .

Tenga en cuenta que utiliza la palabra clave const antes de llamar a cada clase de widget para aprovechar el almacenamiento en caché y la reconstrucción de subárboles para un mejor rendimiento.

```
andamio de vuelta(  
    appBar: AppBar(título:  
        Texto('Cajón'),  
    ),  
    cajón: const LeftDrawerWidget(), endDrawer: const  
    RightDrawerWidget(), cuerpo: SafeArea( child: Container(),  
  
,  
);
```



Cómo funciona

Para agregar un Cajón a una aplicación, establezca la propiedad Cajón andamio o Cajón final . Las propiedades del cajón y del cajón final deslizan el cajón de izquierda a derecha (TextDirection.ltr) o de derecha a izquierda (TextDirection rtl).

El widget Drawer toma una propiedad secundaria y pasaste un ListView. El uso de ListView le permite crear una lista de elementos del menú desplazable. Para la lista de widgets secundarios de ListView , creó dos clases de widgets, una para crear el encabezado del cajón y otra para crear la lista de elementos del menú. Para configurar el encabezado del cajón , tiene dos opciones, UserAccountsDrawerHeader o DrawerHeader. Estos dos widgets le permiten configurar fácilmente el contenido del encabezado según los requisitos. Observó dos ejemplos para el encabezado del cajón llamando a la clase de widget adecuada LeftDrawerWidget() o RightDrawerWidget().

Para la lista de elementos del menú, utilizó la clase de widget MenuListTileWidget() . Esta clase devuelve un widget de columna que usa ListTile para construir su lista de menú. El widget ListTile le permite establecer las propiedades principales de ícono, título y toque . La propiedad onTap llama a Navigator.pop() para cerrar el cajón y llama a Navigator.push() para navegar a la página seleccionada.

Esta aplicación es un excelente ejemplo de creación de un árbol de widgets poco profundo mediante el uso de clases de widgets y su separación en archivos individuales para una máxima reutilización. También anidó clases de widgets con las clases de widgets izquierda y derecha llamando a la clase de lista de menú.

RESUMEN

En este capítulo, aprendió a usar el widget Navigator para administrar una pila de rutas para permitir la navegación entre páginas. Opcionalmente, pasó datos a la página de navegación y de regreso a la página original. La animación del héroe permite que la transición de un widget vuele de una página a otra. El widget para animar desde y hacia está envuelto en un widget Hero por una clave única.

Usó el widget BottomNavigationBar para mostrar una lista horizontal de BottomNavigationBarItems que contiene un ícono y un título en la parte inferior de la página. Cuando el usuario toca cada BottomNavigationBarItem, se muestra la página correspondiente. Para mejorar el aspecto de una barra de navegación inferior, usó el widget BottomAppBar y activó la muesca opcional. La muesca es el resultado de incrustar un FloatingActionButton en BottomAppBar configurando la forma de BottomAppBar en una clase CircularNotchedRectangle() y configurando la propiedad Scaffold floatingActionButton Location en FloatingActionButtonLocation.endDocked .

El widget TabBar muestra una fila horizontal de pestanas. La propiedad tabs toma una lista de widgets y las pestanas se agregan mediante el widget Tab . El widget TabBarView se usa junto con el widget TabBar para mostrar la página de la pestaña seleccionada. Los usuarios pueden deslizar hacia la izquierda o hacia la derecha para cambiar el contenido o tocar cada pestaña. La clase TabController manejó la sincronización de TabBar y seleccionó TabBar View. El TabController requiere el uso de con SingleTickerProviderStateMixin en la clase.

El widget del cajón permite al usuario deslizar un panel desde la izquierda o la derecha. El widget de cajón se agrega configurando el cajón Scaffold o la propiedad endDrawer . Para alinear fácilmente los elementos del menú en una lista, pase un

ListView como hijo del Cajón. Dado que esta lista de menú es corta, utilice el constructor ListView estándar en lugar de un constructor ListView , que se trata en el próximo capítulo. Tiene dos opciones de encabezado de cajón preconstruidas, UserAccountsDrawerHeader o DrawerHeader. Cuando el usuario toca uno de los elementos del menú, la propiedad onTap llama a Navigator.pop() para cerrar Drawer y llama a Navigator.push() para navegar a la página seleccionada.

En el próximo capítulo, aprenderá a usar diferentes tipos de listas. Echará un vistazo a ListView, GridView, Stack, Card y mi favorito: CustomScrollView para usar Slivers.

LO QUE APRENDISTE EN ESTE CAPÍTULO

TEMA	CONCEPTOS CLAVE
Navegador	El widget Navigator administra una pila de rutas para moverse entre páginas.
Animación de héroe	El widget Hero se usa para transmitir una animación de navegación y el tamaño de un widget para que se ubique de una página a otra. Cuando se descarta la segunda página, la animación se invierte a la página de origen.
Barra de navegación inferior	BottomNavigationBar muestra una lista horizontal de elementos BottomNavigationBarItem que contienen un ícono y un título en la parte inferior de la página.
BottomAppBar	El widget BottomAppBar se comporta como BottomNavigationBar, pero tiene una muesca opcional en la parte superior.
Barra de pestañas	El widget TabBar muestra una fila horizontal de pestañas. La propiedad tabs toma una lista de widgets y las pestañas se agregan mediante el widget Tab .
TabBarView	El widget TabBarView se usa junto con el widget TabBar para mostrar la página de la pestaña seleccionada.
Cajón	El widget del cajón permite al usuario deslizar un panel desde la izquierda o la derecha.
Vista de la lista	El constructor ListView estándar le permite crear rápidamente una breve lista de elementos.

9

Creación de listas de desplazamiento y efectos

LO QUE APRENDERÁS EN ESTE CAPÍTULO

How Card es una excelente manera de agrupar información con el contenedor redondeado esquinas y una sombra paralela

Cómo crear una lista lineal de widgets desplazables con ListView

Cómo mostrar mosaicos de widgets desplazables en formato de cuadrícula con GridView

Cómo Stack le permite superponer, colocar y alinear sus widgets secundarios

Cómo crear efectos de desplazamiento personalizados usando CustomScrollView y slivers

En este capítulo, aprenderá a crear listas de desplazamiento que ayuden a los usuarios a ver y seleccionar información. Comenzará con el widget Tarjeta en este capítulo porque se usa comúnmente junto con widgets con capacidad de lista para mejorar la interfaz de usuario (IU) y los datos de grupo. En el capítulo anterior, analizó el uso del constructor básico para ListView y, en este capítulo, usará ListView.builder para personalizar los datos. El widget GridView es un widget fantástico que muestra una lista de datos por un número fijo de mosaicos (grupos de datos) en el eje transversal.

El widget Stack se usa comúnmente para superponer, colocar y alinear widgets para crear una apariencia personalizada. Un buen ejemplo es un carrito de compras con la cantidad de artículos a comprar en el lado superior derecho.

El widget CustomScrollView le permite crear efectos de desplazamiento personalizados mediante el uso de una lista de widgets de astillas. Los fragmentos son útiles, por ejemplo, si tiene una entrada de diario con una imagen en la parte superior de la página y la descripción del diario debajo. Cuando el usuario desliza para leer más, el desplazamiento de la descripción es más rápido que el desplazamiento de la imagen, lo que crea un efecto de paralaje.

USO DE LA TARJETA

El widget Tarjeta es parte de Material Design y tiene esquinas redondeadas y sombras mínimas. Para agrupar y diseñar datos, la tarjeta es un widget perfecto para mejorar el aspecto de la interfaz de usuario. El widget Tarjeta se puede personalizar con propiedades como elevación, forma, color, margen y otras. La propiedad de elevación es un valor de doble, y cuanto mayor sea el número, mayor será la sombra proyectada. Aprendiste en el Capítulo 3, “Aprendizaje de los conceptos básicos de los dardos”, que un doble es un número que requiere una precisión de punto decimal, como 8,50. Para personalizar la forma y los bordes del widget Tarjeta , modifique la propiedad de forma . Algunas de las propiedades de forma son StadiumBorder, UnderlineInputBorder, OutlineInputBorder y otras.

```
Tarjeta
  (elevación: 8.0,
  color: Colors.white, margen:
  EdgeInsets.all (16.0), niño: Columna (
    mainAxisAlignment: MainAxisAlignment.center, children:
    <Widget>[ Text('
      'Barista',
      textAlign: TextAlign.center, estilo:
      TextStyle( fontWeight:
        FontWeight.bold, fontSize: 48.0, color:
        Colors.orange, ), '),
      Text( 'Travel Plans', textAlign:
        TextAlign.center, estilo:
        TextStyle(color: Colores.gris), ), ], ), ),
```

Las siguientes son algunas formas de personalizar la propiedad de forma de la tarjeta (Figura 9.1):

```
// Crear una forma de borde de
estadio: StadiumBorder(),

// Cree una tarjeta de esquinas cuadradas con una sola forma de borde inferior naranja:
UnderlineInputBorder (borderSide: BorderSide (color: Colors.deepOrange)),

// Crear tarjeta de esquinas redondeadas con forma de borde naranja:
OutlineInputBorder(borderSide: BorderSide(color: Colors.deepOrange. withOpacity(0.5))),
```

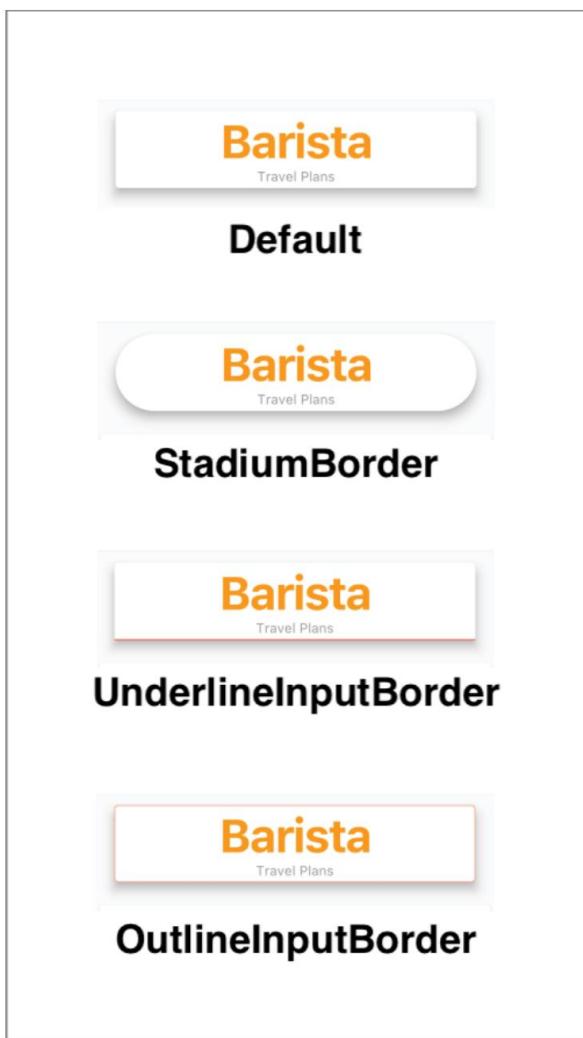


FIGURA 9.1: Personalizaciones de tarjetas

USO DE LISTVIEW Y LISTTILE

El constructor `ListView.builder` se utiliza para crear una lista de widgets lineal desplazable bajo demanda (Figura 9.2). Cuando tiene un gran conjunto de datos, solo se llama al constructor para los widgets visibles, lo que es excelente para el rendimiento. Dentro del constructor, utiliza la devolución de llamada de `itemBuilder` para crear la lista de widgets secundarios. Tenga en cuenta que se llama a `itemBuilder` solo si el argumento `itemCount` es mayor que cero, y se llama tantas veces como el valor de `itemCount`. Recuerde, la lista comienza en la fila 0, no en la 1. Si tiene 20 elementos en la lista, se repite de la fila 0 a la 19. El argumento `scrollDirection` tiene como valor predeterminado `Axis.vertical`, pero se puede cambiar a `Axis.horizontal`.

El widget ListTile se usa comúnmente con el widget

ListView para formatear y organizar fácilmente íconos, títulos y descripciones en un diseño lineal. Entre los principales lazos de propiedad se encuentran los lazos de propiedad inicial, final, de título y de subtítulo , pero hay otros. También puede usar las devoluciones de llamada onTap y onLongPress para ejecutar una acción cuando el usuario toca ListTile. Por lo general, las propiedades iniciales y finales se implementan con iconos, pero puede agregar cualquier tipo de widget.

En la Figura 9.2, los tres primeros ListTile muestran un ícono para la propiedad principal , pero para la propiedad final , un widget de texto muestra un valor porcentual. Los ListTile restantes muestran las propiedades iniciales y finales como íconos.

Otro escenario es usar la propiedad de subtítulos para mostrar una barra de progreso en lugar de una descripción de texto adicional, ya que estas propiedades aceptan widgets.

El primer ejemplo de código aquí muestra una tarjeta con la propiedad secundaria como ListTile para darle un marco agradable y un efecto de sombra. El segundo ejemplo muestra un ListTile base.

```
// Tarjeta con un widget ListTile
```

Tarjeta(hijo:

```
ListTile( encabezado:  
Icono(Icons.vuelo), título: Texto('Avión  
$índice'), subtítulo: Texto('Muy Genial'),  
final: Texto('${ index * 7}%), onTap: ()=>  
print('Tocado en la fila $index'), );
```

```
// ListTile  
ListTile  
(principal: Icon(Icons.directions_car), title:  
Texto('Car $index'), subtitle: Text('Muy  
Cool'), final: Icon(Icons.bookmark),  
onTap: ()=> print('Tocado en la Fila  
$índice'), );
```

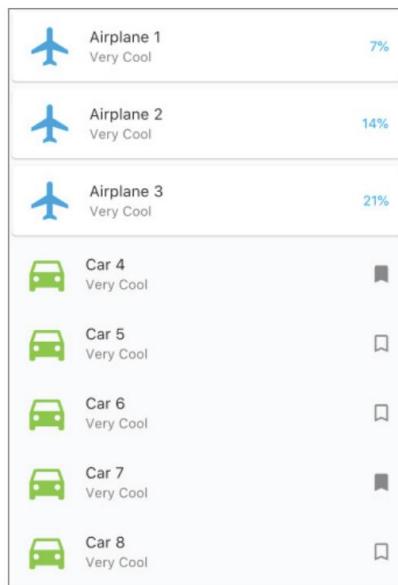


FIGURA 9.2: Diseño lineal de ListView utilizando ListTile

PRUÉBALO Creación de la aplicación ListView

En este ejemplo, el widget ListView usa el constructor para mostrar una Tarjeta para el encabezado y dos variaciones de ListTile para la lista de datos. ListTile puede mostrar widgets iniciales y finales . La propiedad principal muestra un ícono , pero podría haber mostrado una imagen. Para la propiedad final , el primer tipo de ListTile muestra los datos como un porcentaje y el segundo ListTile muestra un ícono de marcador seleccionado o no seleccionado. También establece un título y un subtítulo, y para onTap utiliza la declaración de impresión para mostrar el valor del índice de la fila tocada.

1. Cree un nuevo proyecto de Flutter y asigne el nombre ch9_listview. Puede seguir las instrucciones en Capítulo 4, "Creación de una plantilla de proyecto inicial". Para este proyecto, debe crear solo las carpetas de páginas y widgets . Cree Home Class como StatelessWidget ya que los datos no requieren cambios.

2. Abra el archivo home.dart y agregue al cuerpo un SafeArea con ListView.builder() como elemento secundario.

```
cuerpo: SafeArea( hijo:  
    ListView.builder(), ),
```

3. Establezca ListView.builder con el argumento itemCount establecido en 20. Para este ejemplo, especifica 20 filas de datos. Para la devolución de llamada de itemBuilder , pasa el BuildContext y el índice del widget como un valor int .

Para mostrar cómo crear diferentes tipos de widgets para enumerar cada fila de datos, verifiquemos la primera fila para el valor de índice de cero y llamemos a la clase de widget HeaderWidget(index: index) . Esta clase muestra una tarjeta con el texto "Plan de viaje de barista".

Para las filas primera, segunda y tercera, se llama a la clase de widget RowWithCardWidget(index: index) para mostrar ListTile como hijo de Card. Para el resto de las filas, llame a la clase de widget RowWidget(index: index) para mostrar un ListTile predeterminado.

Es importante comprender que las clases de widget a las que llama desde itemBuilder crean un widget único con el valor de índice pasado. itemBuilder repite el valor de itemCount , en este ejemplo 20 veces.

Creará las tres clases de widgets en el paso 5.

```
cuerpo: SafeArea  
    (hijo: ListView.builder (itemCount:  
        20, itemBuilder:  
            (contexto BuildContext, índice int) { if (índice == 0) {  
  
                devuelve HeaderWidget(indice: índice); } más  
                si (índice >= 1 && índice <= 3) {  
                    return RowWithCardWidget(indice: índice); } más  
                { return  
                    RowWidget(indice: índice);  
  
                } }, ), ),
```

4. Agregue en la parte superior del archivo las declaraciones de importación para las clases de widgets header.dart, row_with_card.dart y row.dart que creará a continuación.

```
importar 'paquete: flutter/material.dart'; importar  
'paquete: ch9_listview/widgets/header.dart'; import 'paquete:ch9_listview/  
widgets/row_with_card.dart'; importar 'paquete: ch9_listview/widgets/row.dart';
```

5. Cree un nuevo archivo Dart en la carpeta de widgets . Haga clic con el botón derecho en la carpeta de widgets y luego seleccione Nuevo Archivo Dart, ingrese header.dart y haga clic en el botón Aceptar para guardar.

6. Importe la biblioteca material.dart , agregue una nueva línea y luego comience a escribir st; se abre la ayuda de autocompletado, así que seleccione la abreviatura stless (StatelessWidget) y asígnele el nombre HeaderWidget.

7. Modifique la clase de widget HeaderWidget para devolver un Contenedor. Importe la biblioteca material.dart .

El contenedor secundario es una tarjeta con una elevación de 8,0 para mostrar una sombra profunda. La lista de elementos secundarios de la tarjeta devuelve dos elementos de texto . A propósito, dejé tres tipos de formas comentados para que pruebas y veas cómo cambian la forma y los bordes de la tarjeta.

Quería señalar que ListView itemBuilder en el archivo main.dart llama a esta clase, HeaderWidget (índice: índice) para cada elemento de fila. Para cada fila, se crea un widget y se agrega al árbol de widgets.

Tenga en cuenta que comenté tres formas diferentes de personalizar la forma de la tarjeta predeterminada para su prueba.

```
importar 'paquete: flutter/material.dart';

clase HeaderWidget extiende StatelessWidget { const
  HeaderWidget({ Clave clave,
    @required
    this.index,
  }) : super(clave: clave);

  índice int final;

  @anular
  Compilación del widget (contexto BuildContext) {
    return Container( relleno:
      EdgelInsets.all(16.0), altura: 120.0, niño:
        Tarjeta( elevación:
          8.0, color:
            Colors.white, //shape:
            StadiumBorder(), //shape:
            UnderlineInputBorder(borderSide:
              BorderSide(color: Colores
                .Naranja intenso)),
        //forma: EsquemaInputBorder(borderSide: BorderSide(color:
        Colors.deepOrange.withOpacity(0.5)), hijo: Columna(
          mainAxisAlignment: MainAxisAlignment.center, children:
            <Widget>[ Text( 'Barista',
              textAlign:
                TextAlign.center, estilo:
                TextStyle( fontWeight:
                  FontWeight.bold, fontSize: 48.0, color:
                  Colors.orange, ),
              ),
            Text( 'Planes de viaje',
              textAlign: TextAlign.center, estilo:
              TextStyle(color: Colors.grey), ),
            ],
        ),
    ),
  );
}
```

```
), ), );
```

```
}
```

8. Cree un nuevo archivo Dart en la carpeta de widgets . Haga clic derecho en la carpeta de widgets y luego seleccione Nuevo Archivo Dart, ingrese fila_con_tarjeta.dart y haga clic en el botón Aceptar para guardar.

9. Importe la biblioteca material.dart , agregue una nueva línea y luego comience a escribir st; se abre la ayuda de autocompletado, así que seleccione la abreviatura stless (StatelessWidget) y asígnele el nombre RowWithCardWidget.

10. Modifique el widget de clase RowWithCardWidget para devolver una tarjeta. Importe la biblioteca material.dart .

El hijo de la tarjeta es un ListTile , que es excelente para alinear el contenido fácilmente. Para la propiedad principal , devuelva un ícono. La propiedad final devuelve un widget de texto con interpolación de cadenas que toma el índice por siete para obtener un número. La propiedad del título devuelve un widget de texto con el valor del índice . La propiedad de subtítulo devuelve un widget de texto . Para la propiedad onTap , utiliza una declaración de impresión para mostrar el índice de la fila tocada.

Como recordatorio, se crea y agrega un widget al árbol de widgets para cada fila.

```
importar 'paquete: flutter/material.dart';
```

```
clase RowWithCardWidget extiende StatelessWidget {
  const RowWithCardWidget({ Key key,
    @required
    this.index,
  }) : super(clave: clave);

  índice int final;

  @anular
  Compilación del widget (contexto BuildContext) { tarjeta
    de retorno (hijo:
      ListTile (principal: Ícono
        (Iconos.vuelo,
        tamaño: 48.0,
        color:
        Colors.lightBlue,), título: Texto
        ('Avión $ índice'), subtítulo: Texto (' Very Cool'),
        final: Text( '${index * 7}%', style:
        TextStyle(color:
        Colors.lightBlue), ), // seleccionado: true, onTap: () { print('Topped on Row $
        índice'); }, ), );
}
```

```
}
```

11. Cree un nuevo archivo Dart en la carpeta de widgets . Haga clic derecho en la carpeta de widgets y luego seleccione Nuevo Dart File, ingrese row.dart y haga clic en el botón Aceptar para guardar.
12. Importe la biblioteca material.dart , agregue una nueva línea y luego comience a escribir st; se abre la ayuda de autocompletado, así que seleccione la abreviatura stless (StatelessWidget) y asígnele el nombre RowWidget.
13. Modifique la clase de widget RowWidget para devolver un ListTile. Importe la biblioteca material.dart .
14. Para la propiedad principal , devuelve un ícono. Para la propiedad final , devuelve un libro mark_border o un ícono de marcador. Para aleatorizar qué ícono devolver, use un operador ternario para calcular el módulo de índice (%) de 3 y verifique si es un número par. Si el número es par o impar, se muestra el ícono correspondiente. La propiedad del título devuelve un widget de texto con el valor del índice . La propiedad de subtítulo devuelve un widget de texto .

Para onTap, utiliza una declaración de impresión para mostrar el índice de la fila tocada.

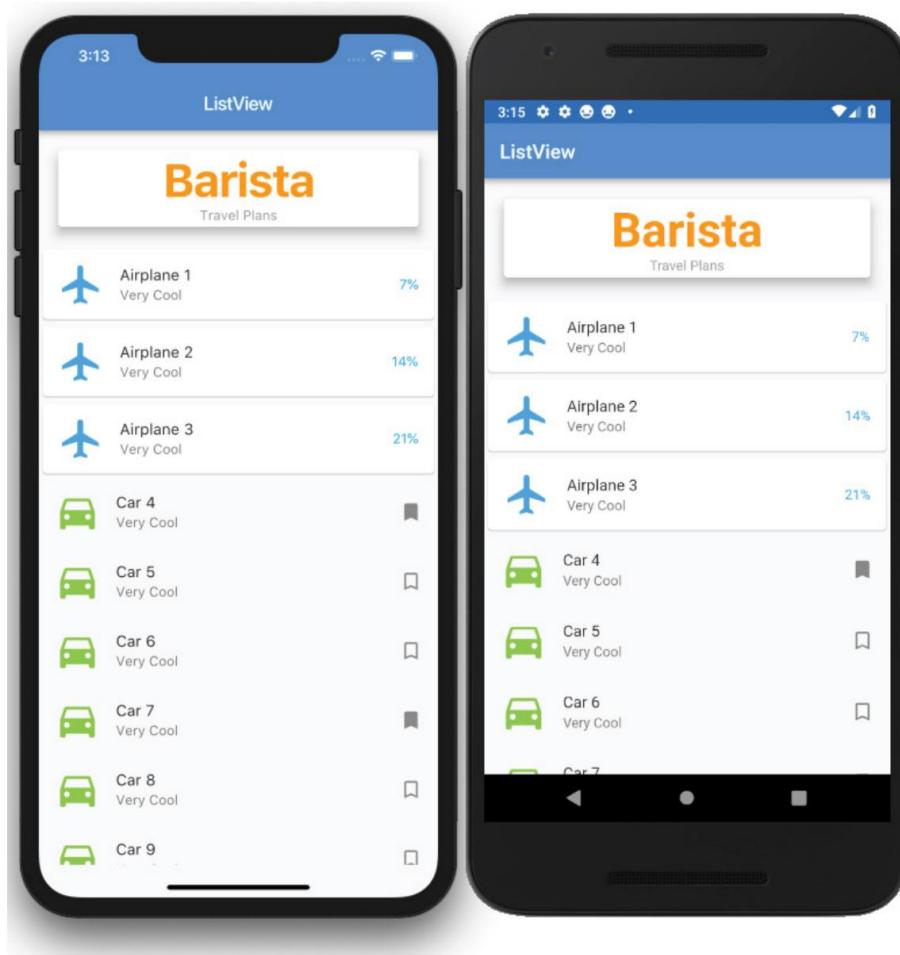
Tenga en cuenta que para cada fila se agrega un widget al árbol de widgets.

```
importar 'paquete: flutter/material.dart';
```

```
class RowWidget extiende StatelessWidget { const
  RowWidget({ clave
    clave,
    @required this.index,
  }) : super(clave: clave);

  índice int final;

  @anular
  Compilación del widget (contexto BuildContext)
  { return ListTile
    (principal: Ícono
      (Iconos.directions_car,
       tamaño: 48.0,
       color: Colors.lightGreen,), título:
      Texto ('Coche $ índice'), subtítulo:
      Texto ('Muy genial') , final: (índice %
      3).isEven
        ? Ícono(Iconos.bookmark_border)
        : Ícon(Icons.bookmark),
      seleccionado: false,
      onTap: ()
        { print('Tocado en la fila $índice'); }, );
  }
}
```



CÓMO FUNCIONA

El constructor `ListView.builder` toma un `itemCount` y usa `itemBuilder` para crear un widget para cada registro secundario. Cada widget secundario se agrega al árbol de widgets con los valores apropiados. A medida que se agrega cada widget secundario, puede personalizar las filas de `ListView` de acuerdo con las especificaciones de la aplicación.

El uso de `ListTile` hace que sea extremadamente fácil alinear widgets. El `ListTile` toma un encabezado, final, título, subtítulo, `onTap` y otras propiedades.

USO DE LA VISTA DE CUADRO

GridView (Figura 9.3) muestra mosaicos de widgets desplazables en un formato de cuadrícula . Los tres constructores en los que me centro son GridView.count, GridView.extent y GridView.constructor.

GridView.count y GridView.extent generalmente se usan con un conjunto de datos fijo o más pequeño . El uso de estos constructores significa que todos los datos, no solo los widgets visibles, se cargan en init. Si tiene un gran conjunto de datos, el usuario no ve GridView hasta que se cargan todos los datos, lo que no es una gran experiencia de usuario (UX). Por lo general, usa GridView.count cuando necesita un diseño con un número fijo de mosaicos en el eje transversal. Por ejemplo, muestra tres mosaicos en modo vertical y horizontal. Utiliza GridView.extent cuando necesita un diseño con los mosaicos que necesitan una extensión máxima de eje transversal.

Por ejemplo, caben de dos a tres mosaicos en modo vertical y de cinco a seis mosaicos en modo horizontal; en otras palabras, se ajusta a tantos mosaicos como sea posible según el tamaño de la pantalla.

El constructor GridView.builder se utiliza con un conjunto de datos de tamaño mayor, infinito o desconocido. Como nuestro ListView. builder, cuando tiene un gran conjunto de datos, se llama al builder solo para los widgets visibles, lo que es excelente para el rendimiento.

Dentro del constructor, utiliza la devolución de llamada de itemBuilder para crear la lista de widgets secundarios. Tenga en cuenta que se llama a itemBuilder solo si el argumento itemCount es mayor que cero, y se llama tantas veces como el valor de itemCount . Recuerde, la lista comienza en la fila 0, no en la 1. Si tiene 20 elementos en la lista, hace un bucle de la fila 0 a la 19. El argumento scrollDirection está predeterminado en Axis.vertical , pero se puede cambiar a Axis.horizontal (Figura 9.3).

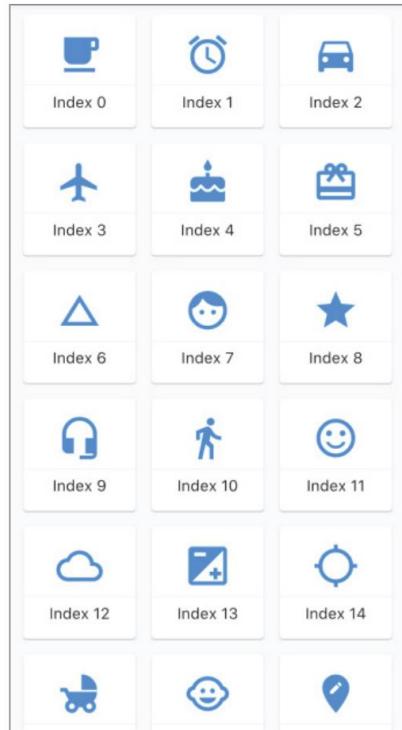


FIGURA 9.3: Diseño GridView

Uso de GridView.count

GridView.count requiere configurar crossAxisCount y el argumento de los niños . crossAxisCount establece la cantidad de mosaicos que se mostrarán (Figura 9.4) y children es una lista de widgets.

El argumento scrollDirection establece la dirección del eje principal para que se desplace la cuadrícula , ya sea Axis .vertical o Axis.horizontal, y el valor predeterminado es vertical.

Para los niños, usa List.generate para crear sus datos de muestra, una lista de valores. Dentro del argumento de los niños , agregué una declaración de impresión para mostrar que toda la lista de valores se crea al mismo tiempo, no solo las filas visibles como GridView.builder . Tenga en cuenta que para el siguiente código de ejemplo, se generan 7000 registros para mostrar que GridView.count no muestra ningún dato hasta que todos los registros se procesan primero.

```
GridView.count(crossAxisCount:  
 3, relleno: EdgeInsets.all(8.0),
```

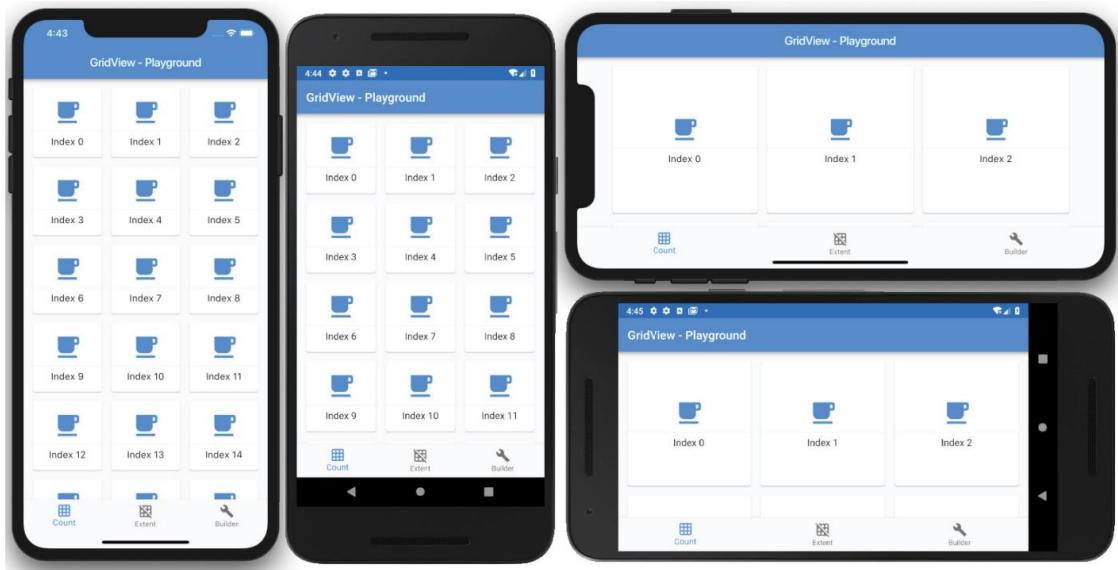


FIGURA 9.4: Recuento de GridView con tres mosaicos en modo vertical y horizontal

```

niños: List.generate(7000, (índice) { print('_buildGridView
    $índice');

tarjeta de retorno
    (margen: EdgeInsets.all (8.0), niño: InkWell
    (
        niño: Column (
            mainAxisAlignment: MainAxisAlignment.center, children:
            <Widget>[ Icon(_iconList[0],
                size:
                    48.0, color:
                        Colors.blue, ),
                Divider(), Text( 'Index $índice',
                    textAlign:
                        TextAlign.center, style :
                            Estilo de texto (tamaño de fuente: 16.0,
                                ),
                ),
            ),
        ],
    ),
    onTap: ()
    { print('Fila $índice'); },
),
);
});},
)

```

Uso de GridView.extent

GridView.extent requiere que establezca el argumento maxCrossAxisExtent y children . El argumento maxCrossAxisExtent establece el tamaño máximo de cada mosaico para el eje. Por ejemplo, en vertical, puede caber de dos a tres mosaicos, pero cuando se gira a paisaje, puede caber de cinco a seis dependiendo del tamaño de la pantalla (Figura 9.5). El argumento scrollDirection establece la dirección del eje principal para que se desplace la cuadrícula, ya sea Axis.vertical o Axis.horizontal, y el valor predeterminado es vertical.

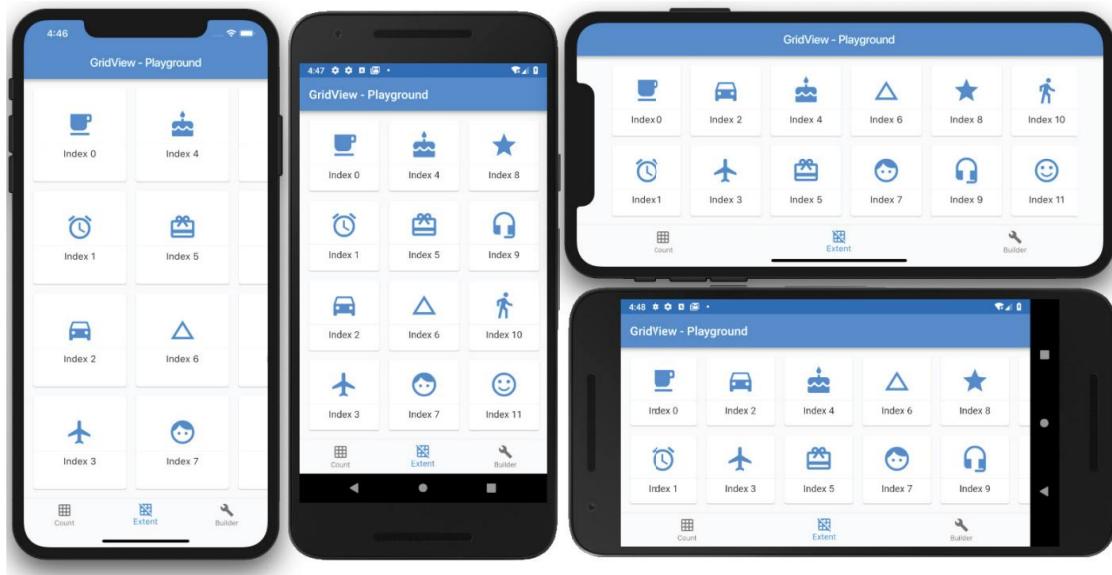


FIGURA 9.5: Extensión de GridView que muestra la cantidad máxima de mosaicos que pueden caber según el tamaño de la pantalla

Para los niños, usa List.generate para crear sus datos de muestra, que es una lista de valores.

Dentro del argumento de los niños , agregué una declaración de impresión para mostrar que toda la lista de valores se crea al mismo tiempo, no solo las filas visibles como GridView.builder .

```
GridView.extent(maxCrossAxisExtent:  
    175.0, scrollDirection: Axis.horizontal, padding:  
    EdgeInsets.all(8.0), children: List.generate(20,  
    (index) {  
        imprimir('_buildGridViewExtent $índice');  
  
        tarjeta de retorno  
        (margen: EdgeInsets.all (8.0), niño: InkWell  
        (  
            niño: Columna (  
                mainAxisAlignment: MainAxisAlignment.center, children:  
                <Widget>[ Icon( _iconList[index],  
                size:  
                    48.0, color: Colors.blue, ),
```

```
        Divider(),  
  
        Text('Índice $índice',  
              textAlign: TextAlign.center, estilo:  
              TextStyle( fontSize: 16.0, ),  
  
    )], ), onTap: ()  
    { print('Fila $índice'); }, ), ); }},  
  
)
```

Uso de GridView.builder

GridView.builder requiere que establezca los argumentos itemCount , gridDelegate y itemBuilder . El itemCount establece el número de mosaicos para construir. El gridDelegate es un delegado de SilverGrid responsable de diseñar la lista secundaria de widgets para GridView. El argumento gridDelegate no puede ser nulo; debe pasar el tamaño maxCrossAxisExtent , por ejemplo, 150,0 píxeles.

Por ejemplo, para mostrar tres mosaicos en la pantalla, especifique el argumento gridDelegate con la clase SliverGridDelegateWithFixedCrossAxisCount para crear un diseño de cuadrícula con un número fijo de mosaicos para el eje transversal. Si necesita mostrar mosaicos que tengan un ancho máximo de 150,0 píxeles, especifique el argumento gridDelegate con la clase SliverGridDelegateWithMaxCrossAxisExtent para crear un diseño de cuadrícula con mosaicos que tengan una extensión máxima de eje transversal, el ancho máximo de cada mosaico.

GridView.builder se usa cuando tiene un gran conjunto de datos porque el constructor solo se llama para mosaicos visibles, lo que es excelente para el rendimiento. El uso del constructor GridView.builder da como resultado la construcción perezosa de una lista de mosaicos visibles, y cuando el usuario se desplaza a los siguientes mosaicos visibles, se construyen perezosamente según sea necesario.

PRUÉBALO Creación de la aplicación GridView.builder

En este ejemplo, el widget GridView usa el constructor para mostrar una tarjeta que muestra cada elemento de la cuadrícula con un ícono y un texto que muestra la ubicación del índice. El onTap imprimirá el índice del elemento de cuadrícula tocado .

1. Cree un nuevo proyecto de Flutter y asigne el nombre ch9_gridview, siguiendo las instrucciones del Capítulo 4.

Para este proyecto, debe crear solo las carpetas de páginas, clases y widgets . Cree Home Class como StatelessWidget ya que los datos no requieren cambios.

2. Abra el archivo home.dart y agregue al cuerpo un SafeArea con GridViewBuilderWidget() clase de widget como un niño.

```
cuerpo: SafeArea( hijo:  
    const GridViewBuildWidget(), ),
```

3. Agregue a la parte superior del archivo la declaración de importación para la clase de `gridview_builder.dart` que creará a continuación.

```
importar 'paquete: flutter/material.dart'; importar  
'paquete:ch9_gridview/widgets/gridview_builder.dart';
```

4. Cree un nuevo archivo Dart en la carpeta de `widgets`. Haga clic derecho en la carpeta de clases y luego seleccione Nuevo Dart File, ingrese `grid_icons.dart` y haga clic en el botón Aceptar para guardar.

5. Importe la biblioteca `material.dart`, agregue una nueva línea y cree la clase `GridIcons`. El `GridIcons` Class contiene una lista de `IconData` llamada `iconList`.

6. Cree el método `getIconList()` que crea la Lista de `IconData` que se usa más adelante en el `GridView`.constructor.

```
clase GridIcons {  
    Lista<IconData> iconList = [];  
  
    List<IconData> getIconList()
```

```
{ iconList ..add(Icons.free_breakfast) ..add(Icons.access_alarms) ..add(Icons.directions_car) ..add(Icons.flight) ..add(Icons.cake) ..add(Icons)
```

```
volver lista de iconos;
```

```
}
```

7. Cree un nuevo archivo Dart en la carpeta de `widgets`. Haga clic con el botón derecho en la carpeta de `widgets` y luego seleccione Nuevo Archivo Dart, ingrese `gridview_builder.dart` y haga clic en el botón Aceptar para guardar.

8. Importe la biblioteca `material.dart`, agregue una nueva línea y luego comience a escribir `st`; se abre la ayuda de autocompletado, así que seleccione la abreviatura `stless` (`StatelessWidget`) y asígnelle el nombre `GridViewBuilderWidget`.

9. Modifique la clase de widget `GridViewBuilderWidget` para que devuelva un `GridView.builder` con el argumento `itemCount` establecido en 20. Para este ejemplo, especifica una lista de 20 filas de datos.

10. Para el argumento `gridDelegate` , utilice `SliverGridDelegateWithMaxCrossAxisExtent(maxCrossExtent: 150.0)`. Su otra opción es usar `SliverGridDelegateWithFixedCrossAxisCount` en su lugar, que funciona de la misma manera que el constructor `GridView.count` , donde pasa la cantidad de mosaicos para mostrar.
11. Para la devolución de llamada de `itemBuilder` , pasa el `BuildContext` y el índice del widget como un valor `int` .
En la primera línea de `itemBuilder` , coloque una declaración de impresión para mostrar el índice de cada elemento que se está construyendo de acuerdo con el espacio visible.
12. Devuelva una Tarjeta con el niño como un `InkWell`. `InkWell onTap` tiene una declaración de impresión para mostrar el elemento de la tarjeta tocado , con la fila seleccionada.
13. Para la propiedad secundaria `InkWell` , pase una Columna con estos elementos secundarios: Icono, Divisor y Texto
widgets Tenga en cuenta que se llama a `itemBuilder` para cada elemento de fila. Para cada fila, se crea un widget y se agrega al árbol de widgets.
El `onTap` imprimirá el índice del elemento de cuadrícula tocado .
14. Agregue en la parte superior del archivo la declaración de importación para la clase `grid_icons.dart` .

```
importar 'paquete: flutter/material.dart'; importar
'paquete:ch9_gridview/classes/grid_icons.dart';

clase GridViewBuilderWidget extiende StatelessWidget {
    const GridViewBuilderWidget({
        Clave
        clave, }): super(clave: clave);

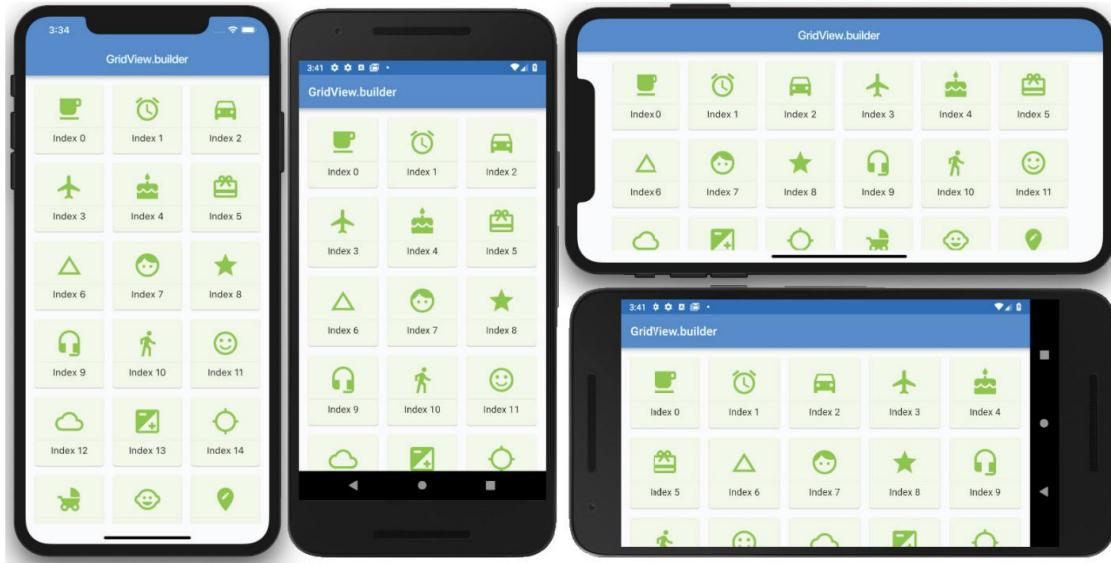
    @anular
    Compilación del widget (contexto BuildContext) {
        List<IconData> _iconList = GridIcons().getIconList();

        return GridView.builder(itemCount:
            20, padding:
            EdgeInsets.all(8.0), gridDelegate:
            SliverGridDelegateWithMaxCrossAxisExtent(maxCrossAxisExtent: 150.0), itemBuilder: (contexto BuildContext, índice
            int)
            { print('_buildGridViewBuilder $index');

                tarjeta de retorno
                (color: Colors.lightGreen.shade50, margen:
                EdgeInsets.all (8.0), niño: InkWell (
                    niño: Column (
                        mainAxisAlignment: MainAxisAlignment.center, children:
                        <Widget>[ Icon( _iconList[index],
                        size:
                        48.0, color:
                        Colors.lightGreen, ),
                        Divider(), Text(
```

```
'Índice $índice',
textAlign: TextAlign.center, estilo:
TextStyle( fontSize: 16.0, ),

) ], ), onTap: ()
{ print('Fila $índice'); }, ), ); }, );  
}}}
```



CÓMO FUNCIONA

El constructor de `GridView.builder` toma un `itemCount` y usa `itemBuilder` para crear un widget para cada registro secundario. Cada widget secundario se agrega al árbol de widgets con los valores apropiados. A medida que se agrega cada widget secundario, puede personalizar las filas de acuerdo con las especificaciones de la aplicación. En este ejemplo, usó una tarjeta para darle a cada elemento de la fila de cuadrícula una apariencia agradable y usó un `InkWell` para usar la animación de toque de Material Design y la propiedad `onTap`. El elemento secundario `InkWell` es una columna y sus elementos secundarios muestran un ícono y un texto.

USO DE LA PILA

El widget Stack se usa comúnmente para superponer, colocar y alinear widgets para crear una apariencia personalizada. Un buen ejemplo es un carrito de compras con la cantidad de artículos a comprar en el lado superior derecho. La lista de hijos de pila del widget está posicionada o no posicionada. Cuando utiliza un widget posicionado , cada widget secundario se coloca en la ubicación adecuada.

El widget Stack cambia de tamaño para adaptarse a todos los elementos secundarios no posicionados. Los elementos secundarios no colocados se colocan en la propiedad de alineación (Superior izquierda o Superior derecha , según el entorno de izquierda a derecha o de derecha a izquierda). Cada widget secundario de Stack se dibuja en orden de abajo hacia arriba, como si se apilaran hojas de papel una encima de la otra. Esto significa que el primer widget dibujado está en la parte inferior de la pila, y luego el siguiente widget se dibuja sobre el widget anterior y así sucesivamente. Cada widget secundario se coloca uno encima del otro en el orden de la lista de elementos secundarios de Stack . La clase RenderStack maneja el diseño de la pila.

Para alinear cada elemento secundario en la Pila, utilice el widget Posicionado . Al usar las propiedades superior, inferior, izquierda y derecha , alinea cada widget secundario dentro de la pila. También se pueden configurar las propiedades de alto y ancho del widget Posicionado (Figura 9.6).

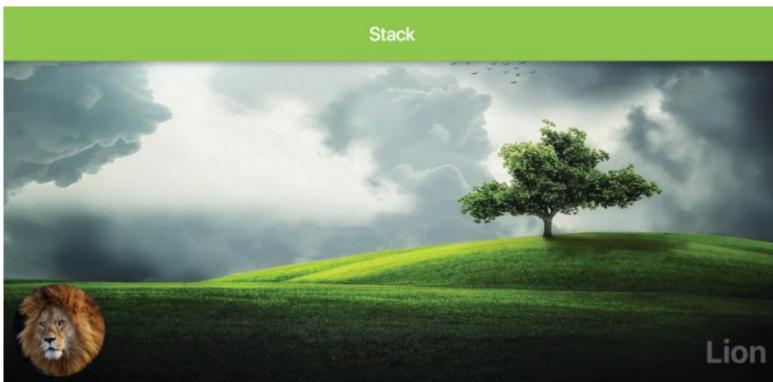


FIGURA 9.6: Diseño de pila que muestra widgets de imagen y texto apilados sobre la imagen de fondo

También aprenderá a implementar la clase FractionalTranslation para colocar un aliado de fracción de widget fuera del widget principal. Establece la propiedad de traducción con la clase Offset(dx, dy) (valor de tipo doble para los ejes x e y) que se escala al tamaño del niño, lo que da como resultado mover y colocar el widget. Por ejemplo, para mostrar un ícono favorito movido un tercio del camino hacia la parte superior derecha del widget principal, configura la propiedad de traducción con el valor Offset(0.3, -0.3) .

El siguiente ejemplo (Figura 9.7) muestra un widget de pila con una imagen de fondo y, mediante el uso de la clase FractionalTranslation , establece la propiedad de traducción en el valor Offset(0.3, -0.3) , colocando el icono de estrella un tercio a la derecha de el eje x y un tercio negativo (mover el icono hacia arriba) en el eje y.

```
Pila(  
    children: <Widget>[ Image(image:  
        AssetImage('assets/images/dawn.jpg')), Positioned(  
            child:
```

```
arriba: 0.0,  
derecha: 0.0,  
niño: FractionalTranslation(  
    traducción: Offset(0.3, -0.3), child:  
        CircleAvatar( child:  
            Icon(Icons.star), ), ), ), Positioned(/  
    *  
  
Eagle Image */), Positioned(* Bald Eagle  
*/), ] , ),
```



FIGURA 9.7: clase FractionalTranslation que muestra el ícono favorito movido hacia la esquina superior derecha

PRUÉBALO Creación de la aplicación Stack

En este ejemplo, la lista secundaria de widgets del widget Stack presenta una imagen de fondo y dos widgets posicionados con los widgets CircleAvatar y Text . Para mostrar un diseño alternativo, utilice el mismo diseño de pila anterior y agregue un widget de posición con la propiedad secundaria como una clase de traducción fraccionaria para mostrar un CircleAvatar anclado en la esquina superior derecha a mitad de camino fuera de la pila.

Se utiliza un ListView para crear la lista de muestra y cada fila muestra un widget de pila alternativo .

1. Cree un nuevo proyecto de Flutter y asígnele el nombre ch9_stack. De nuevo, siga las instrucciones del Capítulo 4. Para este proyecto, debe crear las páginas, los widgets y las carpetas de activos/ímágenes . Cree Home Class como StatelessWidget ya que sus datos no requieren cambios.
2. Abra el archivo pubspec.yaml para agregar recursos. En la sección de activos , agregue los activos/ imágenes/ carpeta.

Para agregar activos a su aplicación, agregue una sección de activos, como esta: activos:

- activos/imágenes/

3. Haga clic en el botón Guardar y, según el editor que esté utilizando, se ejecuta automáticamente el flutter packages get, y una vez que termine, mostrará un mensaje de Proceso terminado con el código de salida 0. Si no ejecuta automáticamente el comando, abra la ventana Terminal (ubicada en la parte inferior de su editor) y escriba flutter packages get .
4. Agregue los activos de la carpeta y las imágenes de la subcarpeta en la raíz del proyecto y luego copie el archivo sunrise.jpg, archivos eagle.jpg, lion.jpg y tree.jpg a la carpeta de imágenes .
5. Abra el archivo home.dart y agregue al cuerpo un SafeArea con ListView.builder() como elemento secundario.

cuerpo: SafeArea(hijo:
ListView.builder(),),

6. Agregue en la parte superior del archivo la declaración de importación para las clases de widgets stack.dart y stack_favorite.dart que creará a continuación.

```
importar 'paquete: flutter/material.dart'; importar  
'paquete:ch9_stack/widgets/stack.dart'; importar 'paquete:ch9_stack/  
widgets/stack_favorite.dart';
```

7. Agregue a ListView.builder el argumento itemCount con un valor establecido en 7. Para este ejemplo, especifica la lista de siete filas de datos. Para la devolución de llamada de itemBuilder , pasa el BuildContext y el índice del widget como un valor int .

Con cada diseño de pila , alterna entre ellos verificando si el valor del índice es par o impar y luego llama a las clases de widgets StackWidget() y StackFavoriteWidget(), respectivamente.

Quería mostrar que puede personalizar los widgets que presenta al usuario. Supongamos que tiene una aplicación que distribuye como software gratuito y cada diez registros muestra un anuncio o un consejo incrustado en la lista. Esta técnica no es tan intrusiva como una ventana emergente mientras el usuario ve los registros.

```
cuerpo: SafeArea (hijo:  
ListView.builder (  
itemCount: 7,  
itemBuilder: (contexto BuildContext, int index) { if (index.isEven) { return  
const StackWidget(); } else  
{ return const StackFavoriteWidget();
```

```
    },  
,  
,
```

8. Cree un nuevo archivo Dart en la carpeta de widgets . Haga clic con el botón derecho en la carpeta de widgets y luego seleccione Nuevo Archivo Dart, ingrese stack.dart y haga clic en el botón Aceptar para guardar.
9. Importe la biblioteca material.dart , agregue una nueva línea y luego comience a escribir st; el autocompletado Se abre la ayuda, donde puede seleccionar la abreviatura stless (StatelessWidget) y asignarle el nombre StackWidget.

10. Modifique la clase de widget StackWidget para devolver una pila. La lista de elementos secundarios de Stack de widgets consta de una Imagen con el árbol AssetImage.jpg.
11. Agregue un widget Posicionado con propiedades inferior e izquierda con un valor de 10.0. El hijo es un CircleAvatar con un radio de 48,0 y tiene la propiedad backgroundImage establecida en AssetImage lion.jpg.
12. Agregue otro widget posicionado con propiedades inferiores y derechas con un valor de 16.0. El child es un widget de texto con un valor de cadena de Lion, y tiene una propiedad de estilo con una clase TextStyle con un fontSize establecido en 32,0 pixeles, color establecido en white30 y fontWeight establecido en negrita.

```
importar 'paquete: flutter/material.dart';

clase StackWidget extiende StatelessWidget { const
  StackWidget({  
    Clave  
    clave, }) : super(clave: clave);  
  
  @anular  
  Compilación del widget (contexto BuildContext)  
  { return Stack  
    (hijos: <Widget>[ Imagen  
      (imagen: AssetImage ('activos/imágenes/árbol.jpg'), ),  
  
      Posicionado  
        (abajo: 10.0,  
        izquierda:  
          10.0, niño: CircleAvatar  
            (radio) : 48.0,  
            backgroundImage: AssetImage('assets/images/lion.jpg'), ), ),  
  
      Positioned( bottom:  
        16.0, right:  
        16.0, child:  
  
        Text( 'Lion', style:  
          TextStyle( fontSize:  
            32.0, color: Colors.white30,  
            fontWeight: FontWeight.bold, ), ), ], );  
  }  
}
```

13. Cree un nuevo archivo Dart en la carpeta de widgets . Haga clic con el botón derecho en la carpeta de widgets y luego seleccione Nuevo Archivo Dart, ingrese stack_favorite.dart y haga clic en el botón Aceptar para guardar.
14. Importe la biblioteca material.dart , agregue una nueva línea y luego comience a escribir st; Se abre la ayuda de autocompletado, así que seleccione la abreviatura stless (StatelessWidget) y asignele el nombre StackFavoriteWidget.

15. Modifique la clase de widget StackFavoriteWidget para devolver un Contenedor. Use un contenedor para establecer color a black87 y para el niño use un Padding con EdgeInsets.all (16.0). Esto creará un efecto de marco oscuro alrededor de la pila.
16. Para la lista de widgets secundarios de la pila , agregue un conjunto de imágenes a AssetImage amanecer.jpg.
17. Agregue un widget posicionado con propiedades inferiores y derechas con los valores 0.0. El hijo es una clase FractionalTranslation con una propiedad de traducción de Offset(0.3,-0.3). El niño es un CircleAvatar, y al usar el Desplazamiento, lo muestra anclado en la esquina superior derecha a la mitad fuera de la Pila.
18. Agregue un widget Posicionado con propiedades inferior y derecha con un valor de 10.0. El niño es un CircleAvatar con un radio de 48,0 y una imagen de fondo con el águila AssetImage.jpg.
19. Agregue otro widget posicionado con propiedades inferiores y derechas que tengan valores de 16.0. El elemento secundario es un widget de texto con un valor de cadena de águila calva y tiene una propiedad de estilo con TextStyle con fontSize establecido en 32,0 píxeles, color establecido en white30 y fontWeight establecido en negrita.

```
importar 'paquete: flutter/material.dart';
```

```
clase StackFavoriteWidget extiende StatelessWidget {  
  const StackFavoriteWidget({  
    Clave  
    clave, }) : super(clave: clave);  
  
  @anular  
  Compilación del widget (contexto BuildContext) { return  
    Container (color:  
      Colors.black87, child: Padding  
      (padding: const  
        EdgeInsets.all (16.0), child: Stack ( children:  
          <Widget>[ Image  
            ( imagen: AssetImage  
              ('assets/  
                images) /dawn.jpg'), ), Positioned( top: 0.0, right: 0.0, child:  
                  
```

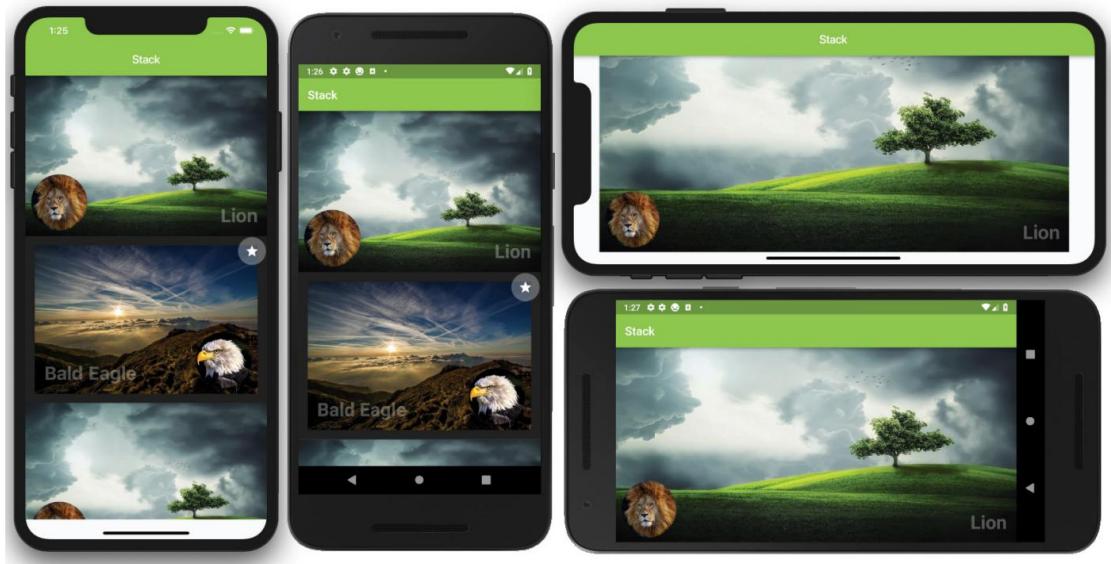
```
          FractionalTranslation(  
            traducción: Compensación (0.3, -0.3), niño:  
              CircleAvatar (radio: 24.0, color  
                de fondo:  
                  Colors.white30, niño: Icono (  
                    Iconos.estrella,  
                    tamaño: 24.0,  
                    color: Colores.blanco, ), ), ), ),  
                
```

```
    Posicionado  
      (abajo: 10.0,
```

```
derecha: 10,0,  
niño: CircleAvatar (radio: 48,0,  
    imagen de fondo:  
        Imagen de activo ('activos/ímágenes/águila.jpg'), ), ), Posicionado (abajo: 16,0,
```

```
izquierda: 16,0,  
niño: Texto ('Águila  
calva', estilo:  
TextStyle( fontSize:  
    32.0, color:  
    Colors.white30, fontWeight:  
        FontWeight.bold, ), ), ), ], ), ), );
```

```
})
```



CÓMO FUNCIONA

La pila toma una lista secundaria de widgets y se dimensiona para acomodar todos los widgets no posicionados. Cuando utiliza widgets no posicionados en la pila, se colocan automáticamente en la configuración de alineación (arriba a la izquierda o arriba a la derecha según el entorno). Cada widget secundario de Stack se dibuja en orden de abajo hacia arriba, lo que significa que cada widget secundario se coloca uno encima del otro.

El uso del widget Posicionado permite alinear cada widget secundario mediante las propiedades superior, inferior, izquierda y derecha . Aprendió a colocar el ícono favorito en la parte superior derecha del widget principal mediante el uso de la propiedad de traducción de la clase FractionalTranslation con el valor Offset(0.3,-03) .

PERSONALIZAR LA CUSTOMSCROLLVIEW CON SLIVERS

El widget CustomScrollView crea efectos de desplazamiento personalizados mediante el uso de una lista de fragmentos. Las astillas son una pequeña porción de algo más grande. Por ejemplo, las astillas se colocan dentro de un puerto de visualización como el widget CustomScrollView . En las secciones anteriores, aprendió cómo implementar los widgets ListView y GridView por separado. Pero, ¿y si necesitaras presentarlos juntos en la misma lista?

La respuesta es que puede usar un CustomScrollView con la lista de widgets de propiedad slivers establecida en los widgets SliverSafeArea, SliverAppBar, SliverList y SliverGrid (slivers). El orden en que los coloca en la propiedad de fragmentos CustomScrollView es el orden en que se representan. La tabla 9.1 muestra fragmentos de uso común y código de muestra.

TABLA 9.1: Astillas

ASTILLA	DESCRIPCIÓN	CÓDIGO
SliverSafeArea	Aggrega relleno para evitar la muesca del dispositivo que generalmente se encuentra en la parte superior de la pantalla	SliverSafeArea(sliver: SliverGrid(),)
SliverAppBar	Aggrega una barra de aplicaciones	SliverAppBar (altura expandida: 250,0, espacio flexible: barra espacial flexible (título: texto ('Parallax'),),)
AstillaLista	Crea una lista desplazable lineal de widgets	SliverList(delegado: SliverChildListDelegate(List.generate(3, (int index) { return ListTile(); }),),)
SliverGrid	Muestra mosaicos de widgets desplazables en formato de cuadrícula	SliverGrid(delegado: SliverChildBuilderDelegate((Contexto BuildContext, int índice) { devolver Tarjeta(); }, childCount: _rowCount,), gridDelegate: SliverGridDelegateWithF ixedCrossAxisCount(crossAxisCount: 3,)

Los slivers SliverList y SliverGrid usan delegados para construir la lista de elementos secundarios de forma explícita o perezosa. Una lista explícita crea primero todos los elementos y luego los muestra en la pantalla. Una lista construida con pereza solo crea los elementos visibles en la pantalla y cuando el usuario se desplaza, los siguientes elementos visibles se crean (con pereza), lo que resulta en un mejor rendimiento. SliverList tiene una propiedad de delegado y SliverGrid tiene un delegado y una propiedad gridDelegate .

La propiedad de delegado SliverList y SliverGrid puede usar SliverChildListDelegate para crear una lista explícita o usar SliverChildBuilderDelegate para crear la lista de forma diferida. SliverGrid tiene una propiedad gridDelegate adicional para especificar el tamaño y la posición de los mosaicos de la cuadrícula . Especifique la propiedad gridDelegate con la clase SliverGridDelegateWithFixed CrossAxisCount para crear un diseño de cuadrícula con un número fijo de mosaicos para el eje transversal; por ejemplo, muestre tres mosaicos de lado a lado. Especifique la propiedad gridDelegate con la clase SliverGrid DelegateWithMaxCrossAxisExtent para crear un diseño de cuadrícula con mosaicos que tengan una extensión máxima de eje transversal, el ancho máximo de cada mosaico; por ejemplo, 150,0 pixeles de ancho máximo para cada mosaico.

La Tabla 9.2 muestra los delegados SliverList y SliverGrid para ayudarlo a crear listas.

TABLA 9.2: Delegados Sliver

ASTILLA	DESCRIPCIÓN	CÓDIGO
SliverList SliverChildListDelegado	<p>construye una lista de número conocido de filas (explícito).</p> <p>SliverChildBuilderDelegate construye perezosamente una lista de un número desconocido de filas.</p>	<pre>SliverList (delegado: SliverChildListDelegate(<Widget>[ListTile(título: Texto('Uno')), ListTile(título: Texto('Dos')), ListTile(título: Texto('Tres')),]),) o SliverList(delegado: SliverChildListDelegate(List.generate(30, (int index) { return ListTile(); })),),) o SliverList(delegado: SliverChildBuilderDeleg ate((Contexto de compilación, índice int) { return ListTile(); }, childCount: _rowCount,),))</pre>

ASTILLA	DESCRIPCIÓN	CÓDIGO
SliverGrid	<p>SliverChildListDelegate crea una lista explícita.</p> <p>SliverChildBuilderDelegate construye perezosamente una lista de un número desconocido de mosaicos. La propiedad gridDelegate controla la posición y el tamaño de los widgets secundarios.</p>	<pre>SliverGrid(delegado: SliverChildListDelegate (<Widget>[Tarjeta(), Tarjeta(), Tarjeta(),]), gridDelegate: SliverGridDelegateWithFixedCrossAxis Contar (recuento de ejes cruzados: 3),) o SliverGrid(delegado: SliverChildListDelegate(List.generate(30, (int index) { return Card(); })),), gridDelegado: SliverGridDelegateWithFixedCross Contador de ejes(contador de ejes cruzados: 3),) o SliverChildBuilderDelegado((Contexto BuildContext, índice int) { devolver Tarjeta(); } childCount: _rowsCount,) gridDelegate: SliverGridDelegateWithFixedCross Contador de ejes(contador de ejes cruzados: 3),)</pre>

El widget SliverAppBar puede tener un efecto de paralaje (Figura 9.8) usando las propiedades expandedHeight y flexibleSpace . El efecto de paralaje desplaza una imagen de fondo más lentamente que el contenido en primer plano. Si necesita mostrar el CustomScrollView inicialmente desplazado en una posición particular, use un controlador y establezca la propiedad ScrollController.initialScrollOffset . Por ejemplo, establecería initialScrollOffset inicializando el controlador = ScrollController(initialScrollOffset: 10.0).

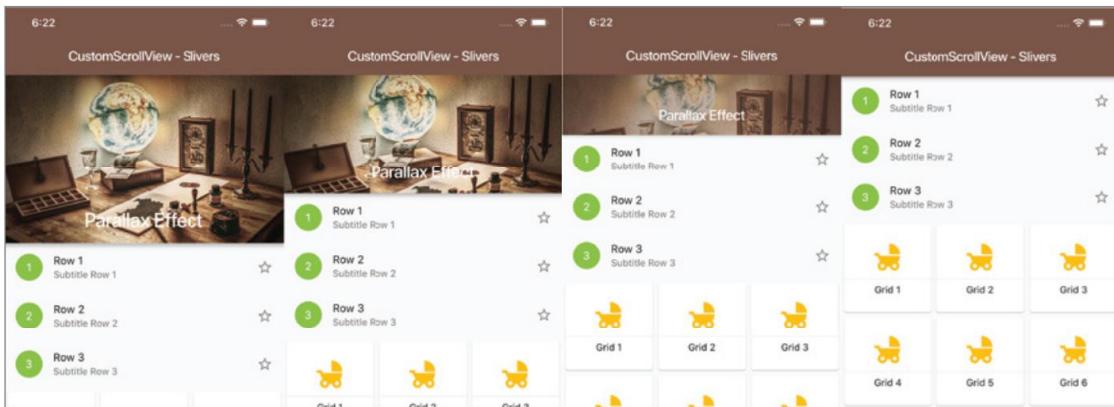


FIGURA 9.8: Efecto de paralelo de desplazamiento SliverAppBar

PRUÉBALO Creación de la aplicación CustomScrollView Slivers

En este ejemplo, la lista secundaria de widgets CustomScrollView contiene SliverAppBar, SliverList, SliverSafeArea y SliverGrid. El widget SliverAppBar usa el espacio flexible con una imagen de fondo que tiene un efecto de paralelo mientras se desplaza. SliverList genera tres elementos con el constructor List.generate . Para tener en cuenta la muesca del dispositivo, utiliza un SliverSafeArea para envolver el SliverGrid y genera 12 elementos (el valor de muestra puede ser más o menos).

1. Cree un nuevo proyecto de Flutter y asigne el nombre ch9_customscrollview_slivers; puedes seguir las instrucciones en el Capítulo 4. Para este proyecto, solo necesita crear las páginas y las carpetas de activos/índices . Cree Home Class como StatelessWidget ya que los datos no requieren cambios.
2. Abra el archivo pubspec.yaml para agregar recursos. En la sección de activos , agregue los activos/ imágenes/ carpeta.


```
# Para agregar activos a su aplicación, agregue una sección de activos, como esta: activos:
- activos/índices/
```
3. Haga clic en el botón Guardar y, según el editor que esté utilizando, se ejecutará automáticamente Flutter. Los paquetes obtienen. Una vez que haya terminado, mostrará un mensaje de Proceso finalizado con el código de salida 0. Si no ejecuta automáticamente el comando por usted, abra la ventana de Terminal (ubicada en la parte inferior de su editor) y escriba flutter packages get.
4. Agregue los activos de la carpeta y las imágenes de la subcarpeta en la raíz del proyecto y luego copie el archivo desk.jpg en la carpeta de imágenes .
5. Abra el archivo home.dart y agregue al cuerpo un CustomScrollView(). Para este proyecto, establezca la propiedad de elevación de AppBar en 0.0 porque en su lugar habilita la sombra de SliverAppBar .

```
andamio de vuelta(
```

```
barra de aplicaciones: barra de aplicaciones (
```

```
    título: Texto ('CustomScrollView - Slivers'), elevación: 0.0,
```

```
),
```

```
cuerpo: CustomScrollView(slivers:  
    <Widget>[ ], ), );
```

6. Agregue a la parte superior del archivo la declaración de importación para sliver_app_bar.dart, sliver_list.dart y sliver_grid.dart clases de widgets que creará a continuación.

```
importar 'paquete: flutter/material.dart'; importar 'paquete:  
ch9_customscrollview_slivers/widgets/sliver_app_bar.dart'; importar 'paquete: ch9_customscrollview_slivers/widgets/  
sliver_list.dart'; importar 'paquete: ch9_customscrollview_slivers/widgets/sliver_grid.dart';
```

7. Agregue llamadas a SliverAppBarWidget(), SliverListWidget() y SliverGridWidget() clases de widgets a la propiedad de fragmentos CustomScrollView() . Asegúrese de que las llamadas a las clases de widgets utilicen la palabra clave const para aprovechar el almacenamiento en caché para mejorar el rendimiento.

andamio de vuelta(

```
barra de aplicaciones: barra de aplicaciones (   
    título: Text('CustomScrollView - Slivers'), elevación: 0.0, ), cuerpo:  
    CustomScrollView( slivers:  
  
        <Widget>[ const SliverAppBarWidget(),  
            const SliverListWidget(),  
            const SliverGridWidget(), ], ), );
```

8. Cree un nuevo archivo Dart en la carpeta de widgets . Haga clic con el botón derecho en la carpeta de widgets y luego seleccione Nuevo Archivo Dart, ingrese sliver_app_bar.dart y haga clic en el botón Aceptar para guardar.

9. Importe la biblioteca material.dart , agregue una nueva línea y luego comience a escribir st; se abre la ayuda de autocompletado, así que seleccione la abreviatura stless (StatelessWidget) y asígnele el nombre SliverAppBarWidget.

10. Modifique la clase de widget SliverAppBarWidget para devolver una SliverAppBar.

11. Para mostrar una sombra en la parte inferior de la barra, establezca la propiedad forceElevated en true.

12. Para crear un efecto de paralaje mientras se desplaza, establezca la altura expandida en 250,0 píxeles y flexible.
Espacio a FlexibleSpaceBar.

13. Para la propiedad de fondo FlexibleSpaceBar , use el widget Imagen con el archivo desk.jpg y ajústelo a BoxFit.cover .

```
importar 'paquete: flutter/material.dart';  
  
clase SliverAppBarWidget extiende StatelessWidget {  
    const SliverAppBarWidget({  
        Clave  
        clave, }) : super(clave: clave);
```

```
@anular
Compilación del widget (contexto BuildContext) { return
SliverAppBar (color de fondo:
    Colors.brown, forceElevated: true, expandHeight:
    250.0, espacio flexible:
    FlexibleSpaceBar (título: Texto
    ('Efecto de paralaje'), fondo: Imagen (imagen:
        AssetImage
        ('activos/ images/desk.jpg'),
        ajuste: BoxFit.cover, ), ), );
}

})
```

14. Cree un nuevo archivo Dart en la carpeta de widgets . Haga clic derecho en la carpeta de widgets y luego seleccione Nuevo Dart File, ingrese sliver_list.dart y haga clic en el botón Aceptar para guardar.
15. Importe la biblioteca material.dart , agregue una nueva línea y luego comience a escribir st; se abre la ayuda de autocompletado, así que seleccione la abreviatura stless (StatelessWidget) y asígnele el nombre SliverListWidget.
16. Modifique la clase de widget SliverListWidget para devolver una SliverList. Para la SliverList del egate , pase SliverChildListDelegate.
17. Use el constructor List.generate para construir su lista de datos de muestra. El constructor toma dos argumentos: la longitud de la lista y el índice. Devuelve un ListTile con un CircleAvatar principal con el elemento secundario como un widget de texto con interpolación de cadenas establecida con \${index + 1}.
18. Además, configure el título, el subtítulo y las propiedades finales de ListTile .

```
importar 'paquete: flutter/material.dart';

class SliverListWidget extiende StatelessWidget {
    const SliverListWidget({
        Clave
        clave, }) : super(clave: clave);

    @anular
    Compilación del widget (contexto BuildContext) {
        volver SliverList(
            delegado: SliverChildListDelegate( List.generate(3,
                (int index) { return ListTile(liderando:
                    CircleAvatar( child:
                        Text("${index + 1}"), color de
                        fondo: Colors.lightGreen, color de primer
                        plano: Colors.white, ), título: Texto('Fila ${indice + 1}'),
                    ),
                ),
            ),
        ),
    );
}
```

```
        subtítulo: Texto('Subtítulo Fila ${índice + 1}'), final:  
        Icono(Icons.star_border), ); }, ), );  
  
    }  
}
```

19. Cree un nuevo archivo Dart en la carpeta de widgets . Haga clic derecho en la carpeta de widgets y luego seleccione Nuevo Dart File, ingrese sliver_grid.dart y haga clic en el botón Aceptar para guardar.
20. Importe la biblioteca material.dart , agregue una nueva línea y luego comience a escribir st; se abre la ayuda de autocompletado, así que seleccione la abreviatura stless (StatelessWidget) y asígnale el nombre SliverGridWidget .
21. Modifique la clase de widget SliverGridWidget para devolver un SliverSafeArea . Desde SliverGrid no maneja la muesca del dispositivo automáticamente, lo envuelve en un SliverSafeArea . La propiedad del delegado SliverGrid es un SliverChildBuilderDelegate que toma el BuildContext y el índice int .
22. Desde SliverChildBuilderDelegate , devuelva una tarjeta con el niño como columna . La lista de hijos de Columna de Widget tiene widgets de Icono , Divisor y Texto .
23. Para la propiedad childCount , pase 12 que representa cuántos elementos crea el constructor . La propiedad gridDelegate se establece en SliverGridDelegateWithFixedCrossAxisCount(cross AxisCount: 3) y muestra tres mosaicos .

```
importar 'paquete: flutter/material.dart';  
  
class SliverGridWidget extiende StatelessWidget {  
  const SliverGridWidget({  
    Clave  
    clave, }) : super(clave: clave);  
  
  @anular  
  Compilación del widget (contexto BuildContext)  
  { return SliverSafeArea (sliver:  
    SliverGrid (  
      delegado: SliverChildBuilderDelegado(  
        (Contexto BuildContext, int index) { return  
          Card( child:  
  
            Column( mainAxisAlignment: MainAxisAlignment.center,  
              children: <Widget>[  
                Icon(Icons.child_friendly, size: 48.0, color: Colors.amber,), Divider(), Text('Grid $  
                {index +  
                1}'), ], ), ); }, childCount: 12, ),  
  
    gridDelegate:
```

```
SliverGridDelegateWithFixedCrossAxisCount(crossAxisCount: 3), ), );
```

```
}
```



CÓMO FUNCIONA

Para crear un efecto de desplazamiento personalizado, CustomScrollView utiliza una lista de fragmentos. Usó SliverAppBar para crear un efecto de desplazamiento de paralelo usando FlexibleSpaceBar . También creó una SliverList y estableció la propiedad del delegado en la clase SliverChildListDelegate . Para manejar la muesca del dispositivo, SliverGrid está envuelto en un widget SliverSafeArea . La propiedad de delegado SliverGrid usa SliverChildBuilderDelegate , que toma un BuildContext y un índice int

RESUMEN

En este capítulo, aprendió a usar la Tarjeta para agrupar información con el contenedor que tiene esquinas redondeadas y una sombra. Usó ListView para crear una lista de widgets desplazables y para alinear datos agrupados con ListTile, y usó GridView para mostrar datos en mosaicos, usando Card para agrupar los datos. Incrustó una Pila en un ListView para mostrar una Imagen como fondo y apiló diferentes widgets con el widget Posicionado para superponerlos y colocarlos en las ubicaciones apropiadas usando las propiedades superior, inferior, izquierda y derecha .

En el próximo capítulo, aprenderá a crear diseños personalizados mediante SingleChildScrollView, SafeArea, Padding, Column, Row, Image, Divider, Text, Icon, SizedBox, Wrap, Chip y CircleAvatar. Aprenderá a tomar una vista de alto nivel, así como una vista detallada para separar y anidar widgets para crear una interfaz de usuario personalizada.

LO QUE APRENDISTE EN ESTE CAPÍTULO

TEMA	CONCEPTOS CLAVE
Tarjeta	Esto agrupa y organiza la información en un contenedor con esquinas redondeadas y proyecta una sombra paralela.
Vista de la lista y ListTile	Esta es una lista lineal de widgets desplazables con desplazamiento vertical u horizontal. Para formatear fácilmente cómo se muestra la lista de registros en filas, aprovechó el widget ListTile para alinear los datos agrupados con los iconos iniciales y finales.
Vista en cuadrícula	Esto muestra mosaicos de widgets desplazables en formato de cuadrícula. El desplazamiento puede ser vertical u horizontal.
Pila	Esto se usa comúnmente para superponer, colocar y alinear sus widgets secundarios para crear una apariencia personalizada.
CustomScrollView y astillas	Esto le permite crear efectos de desplazamiento personalizados mediante el uso de una lista de widgets de fragmentos como SliverSafeArea, SliverAppBar, SliverList, SliverGrid y más.

10

Diseños de construcción

LO QUE APRENDERÁS EN ESTE CAPÍTULO

Cómo crear diseños simples y complejos

Cómo combinar y anidar widgets

Cómo combinar widgets verticales y horizontales para crear un diseño personalizado

Cómo crear el diseño de la interfaz de usuario mediante widgets como SingleChildScrollView, SafeArea, Relleno, Columna, Fila, Imagen, Divisor, Texto, Icono, SizedBox, Wrap, Chip y CircleAvatar

En este capítulo, aprenderá cómo tomar widgets individuales y anidarlos para crear un diseño profesional. Este concepto se conoce como composición y es una gran parte de la creación de aplicaciones móviles de Flutter. La mayoría de las veces puede crear diseños simples o complejos utilizando widgets verticales u horizontales o una combinación de ambos.

UNA VISTA DE ALTO NIVEL DEL DISEÑO

El objetivo de este capítulo es crear una página de entrada de diario que muestre los detalles de arriba a abajo. La página del diario muestra la imagen del encabezado, el título, los detalles del diario, el clima, la dirección (ubicación del diario), las etiquetas y las imágenes del pie de página. Las secciones de clima, etiquetas e imágenes de pie de página se construyen anidando widgets para crear un diseño personalizado (Figura 10.1).

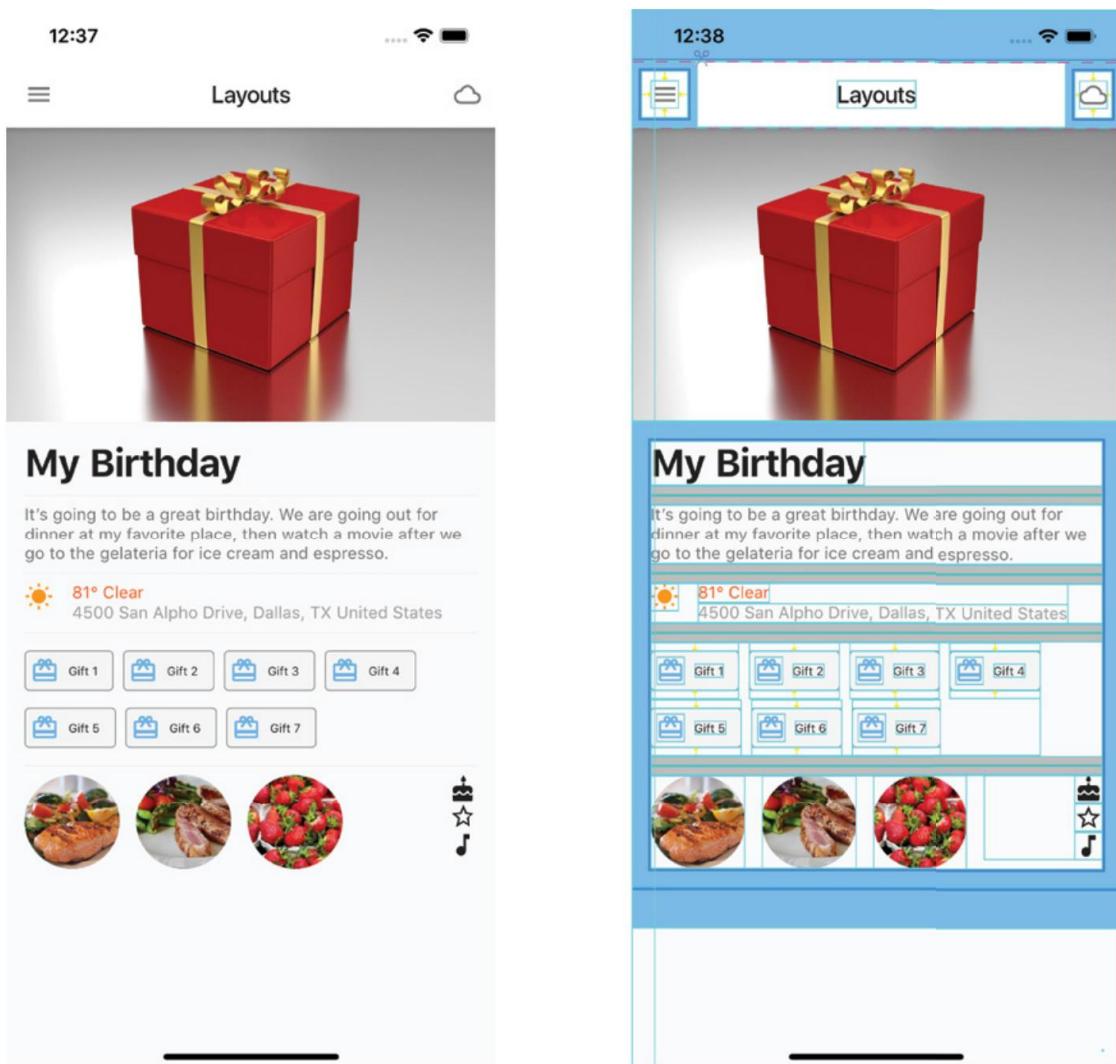


FIGURA 10.1: Diseño de la página de detalles de la revista

Comenzando con una vista de alto nivel, analicemos las partes principales del diseño que forma la base. Una excelente manera de comenzar a diseñar la entrada del diario es agregar capas desde la parte inferior hacia la parte superior, de la misma manera que apila el papel. La figura 10.2 muestra la estructura del diseño de la página de la revista.

1. Dado que los dispositivos móviles están disponibles en diferentes tamaños, el diseño comienza agregando un solo ChildScrollView para manejar automáticamente el desplazamiento de partes de la pantalla que están cortadas por dispositivos más pequeños.
2. A continuación , se usa un widget de columna para alinear los widgets verticalmente desde la parte superior hacia la parte inferior de la pantalla.

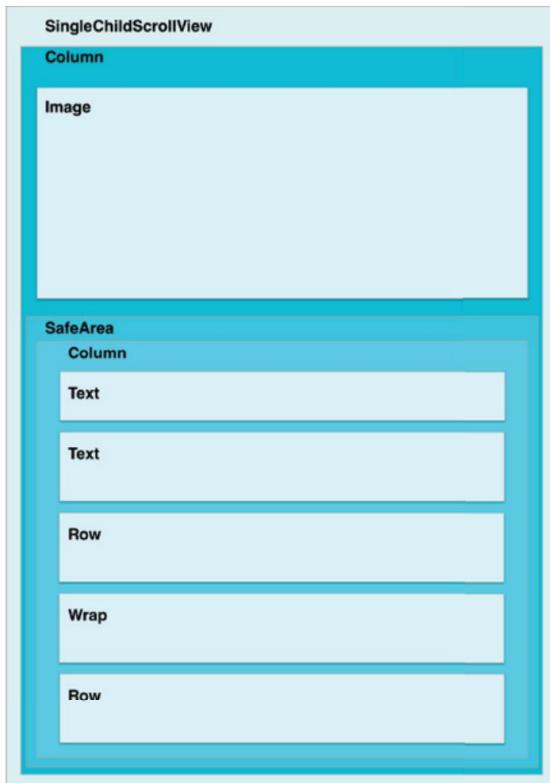


FIGURA 10.2: Vista de alto nivel

3. Para la imagen del regalo envuelto, el widget `Imagen` se agrega como el primer hijo de la primera Columna, permitiendo que la imagen ocupe todo el ancho del dispositivo.
4. El primer hijo de la columna es un widget de `SafeArea` para manejar la muesca del dispositivo para el diario. contenido de la entrada.
5. Agregue a `SafeArea child` una segunda columna con widgets secundarios compuesta por un widget de texto para el título de la entrada del diario y un widget de texto para los detalles de la entrada del diario.
6. Continúe agregando a los elementos secundarios de la segunda columna un widget de fila que contendrá el ícono del clima, la temperatura del clima y la dirección de la ubicación de la entrada del diario. En la sección "Diseño de la sección meteorológica" de este capítulo, aprenderá a agregar widgets para crear el diseño detallado.
7. Continúe agregando a los elementos secundarios de la segunda columna un widget `Wrap` que muestre los widgets `Chip`. Aprenderá a agregar widgets de diseño en la sección "Diseño de etiquetas".
8. Por último, agregará a la segunda columna un widget de fila para mostrar imágenes e íconos, y aprenderá cómo agregar widgets de diseño en la sección "Diseño de imágenes de pie de página" de este capítulo.

Diseño de la sección meteorológica

Cada entrada de diario registra el clima, la temperatura y la ubicación en el momento de la entrada para recordar los detalles en un momento posterior. Para proporcionar esa información, está incluyendo una sección meteorológica de entrada de diario. Usando una Fila, agrega dos widgets de Columna y un widget de SizedBox . La primera columna contiene un ícono para mostrar el símbolo del tiempo. La segunda columna contiene dos widgets de fila . La primera fila tiene un texto que muestra la temperatura y la descripción del clima. La segunda Fila tiene un Texto que muestra la dirección de ubicación de la entrada del diario (Figura 10.3).

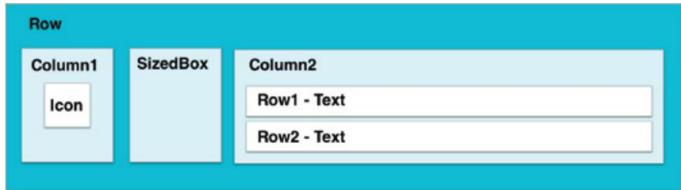


FIGURA 10.3: Sección meteorológica

Diseño de etiquetas

Para organizar cada entrada del diario y facilitar la búsqueda, utiliza etiquetas para agregar categorías a la entrada. Las etiquetas son elementos como película, familia, cumpleaños, vacaciones, etc. La sección de etiquetas utiliza un widget Wrap con una lista secundaria de widgets Chip . Cuando tiene una lista de elementos que pueden tener diferentes longitudes y un número desconocido de elementos, al anidarlos en un widget Wrap , cada elemento secundario se distribuye automáticamente de acuerdo con el espacio disponible (Figura 10.4).

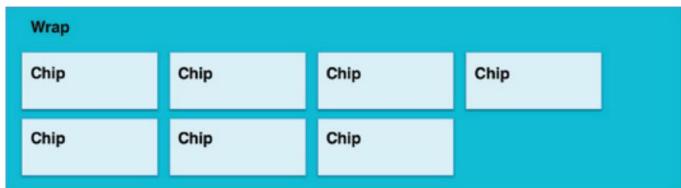


FIGURA 10.4: Sección Etiquetas

El widget Chip es una excelente manera de agrupar información y personalizar la apariencia de la presentación. Configurar la propiedad de la etiqueta es el único requisito, pero la mayoría de las veces se usa configurando la propiedad del avatar con un ícono o un widget de imagen . De forma predeterminada, el widget Chip tiene una forma de estadio gris (rectángulo con grandes semicírculos en los extremos en los lados opuestos), pero puede personalizarlo utilizando la propiedad de forma y la propiedad backgroundColor . El siguiente código de ejemplo muestra un widget de chip personalizado que muestra la etiqueta y el avatar en forma rectangular con pequeñas esquinas redondeadas. La clase RoundedRectangleBorder devuelve el borde rectangular con esquinas redondeadas.

```
Chip( etiqueta: Texto('Vacaciones'),  
      avatar: Icono(Iconos.local_aeropuerto),  
      forma:  
        RoundedRectangleBorder(borderRadius:  
          BorderRadius.circular(4.0), side: BorderSide(color:  
            Colors.grey), ), backgroundColor: Colors.  
            gris.sombra100, );
```

Diseño de imágenes de pie de página

Se dice que una imagen vale más que mil palabras, y la sección de pie de página le permite agregar fotos a cada entrada del diario para traer recuerdos. Las secciones de pie de página usan una fila con un widget CircleAvatar que muestra diferentes imágenes. Al final de la Fila, se utiliza un SizedBox para espaciar la Columna secundaria hasta el final. La Columna muestra Iconos alineados verticalmente (Figura 10.5).

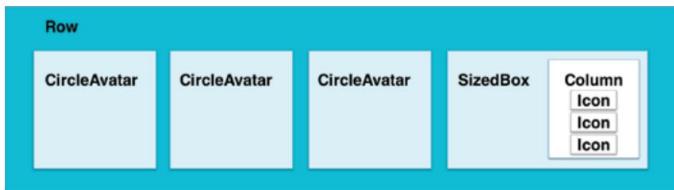


FIGURA 10.5: Sección de pie de página

Diseño final

Observó cómo diseñar cada sección de la página de detalles del diario. Al anidar widgets, crea diseños personalizados o complejos conocidos como composición. El poder de anidar widgets para crear hermosas interfaces de usuario está limitado únicamente por su imaginación. La Figura 10.6 muestra la página de detalles de la revista y las tres secciones principales personalizadas para el clima, las etiquetas y las imágenes de pie de página.

CREANDO EL DISEÑO

Al crear el diseño, es bueno comenzar desde una vista de alto nivel y luego avanzar hacia cada sección detallada. Al tomar cada sección de la página, comienza a analizar los requisitos y el formato según sea necesario. Por ejemplo, si una sección en particular dispone los elementos horizontalmente, comienza con una Fila; si el diseño de la sección es vertical, comienza con una Columna. Luego observa los requisitos de visualización y comienza a desglosar los datos en sus propias secciones anidando widgets.

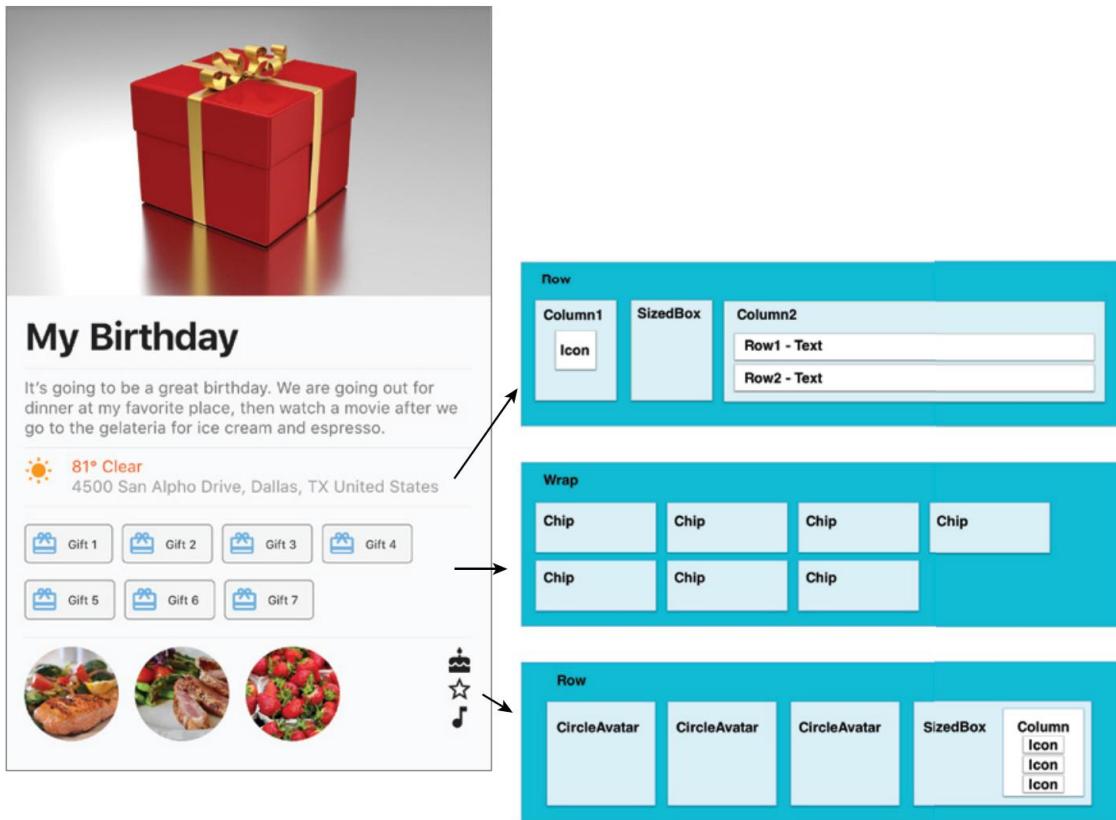


FIGURA 10.6: Diseño final

PRUÉBALO Creación de la aplicación de diseño

En nuestro ejemplo, el cuerpo principal contiene un `SingleChildScrollView` con el elemento secundario como Columna. La lista Columna de widgets contiene una imagen de encabezado seguida de un área segura con relleno como elemento secundario. La imagen se ajusta a todo el ancho del dispositivo, pero las entradas del diario están contenidas en `SafeArea` con un relleno para formatear la entrada.

El elemento secundario `Padding` es una columna con una lista de widgets que desglosan cada sección de la entrada del diario separada por un widget de divisor. Los métodos `_buildJournalHeaderImage()`, `_buildJournalEntry()`, `_buildJournalWeather()`, `_buildJournalTags()` y `_buildJournalFooterImages()` llaman a cada sección por separado.

Está creando una página de entrada de diario que muestra los detalles de arriba a abajo. La página del diario muestra la imagen del encabezado, el título, los detalles del diario, el clima, la dirección, las etiquetas y las imágenes del pie de página. Las secciones de clima, etiquetas e imágenes de pie de página se construyen anidando widgets para crear un diseño personalizado.

En este ejemplo, para mantener el árbol de widgets poco profundo, utilizará métodos en lugar de clases de widgets. Este es un gran ejemplo del uso de la técnica adecuada para cada situación. El propósito de la entrada de diario es ver los detalles y no requiere cambios, razón por la cual utiliza métodos para mantener el widget.

árbol poco profundo. Pero si esta página requiere actualizar partes de la pantalla en función de cambios de datos externos, entonces usar clases de widget podría ser la mejor solución, porque solo se reconstruye la parte de la pantalla que cambia.

1. Cree un nuevo proyecto de Flutter y asígnele el nombre ch10_layouts. Puede seguir las instrucciones en Capítulo 4, "Creación de una plantilla de proyecto inicial". Para este proyecto, debe crear solo las páginas y las carpetas de activos/ímágenes . Cree Home Class como StatelessWidget ya que los datos no requieren cambios.

2. Abra el archivo pubspec.yaml para agregar recursos. En la sección de activos , agregue los activos/ imágenes/ carpeta.

Para agregar activos a su aplicación, agregue una sección de activos, como esta: activos:

- activos/ímágenes/

3. Haz clic en el botón Guardar y, según el Editor que estés usando, se ejecutará automáticamente Flutter. los paquetes se obtienen, y una vez terminado, mostrará un mensaje de Proceso terminado con el código de salida 0. Si no ejecuta automáticamente el comando por ti, abre la ventana Terminal (ubicada en la parte inferior de tu editor) y escribe flutter packages get.

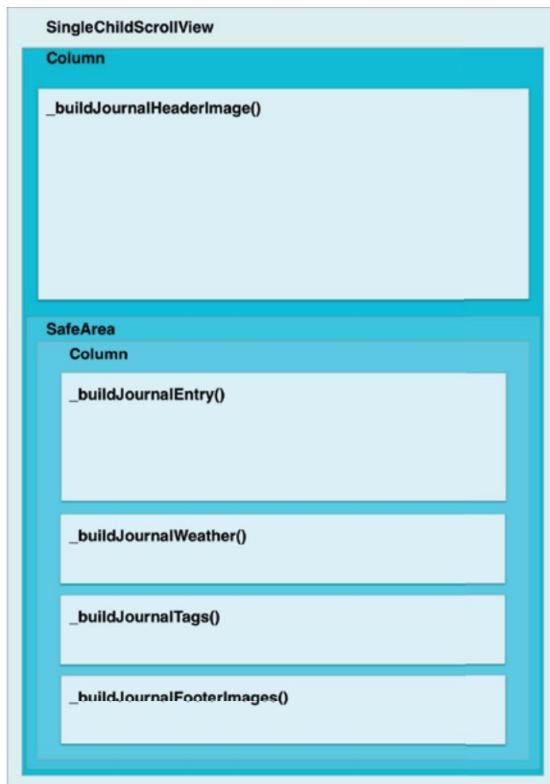
4. Agregue los recursos de la carpeta y las imágenes de la subcarpeta en la raíz del proyecto y luego copie los archivos presente.jpg, salmón.jpg, espárragos.jpg y fresas.jpg en la carpeta de imágenes . Dado que este ejemplo se centra en cómo diseñar una pantalla, las imágenes se incluyen en la carpeta de activos/ímágenes , pero en una aplicación real, el usuario elegiría las imágenes y, en su lugar, se guardarían en el dispositivo.

5. Abra el archivo home.dart y agregue al cuerpo el método _buildBody() . Para este proyecto, personalice el widget AppBar cambiando las propiedades backgroundColor, iconTheme, brillo, interlineado y acciones , como se muestra en el siguiente código. Hacer los cambios de AppBar es puramente cosmético para que la aplicación se vea genial. En el Capítulo 6, "Uso de widgets comunes", aprendió que el widget Icon se dibuja con un glifo de una fuente descrita en IconData. Tienes disponible una lista completa de íconos de la fuente MaterialIcons .

```
Creación de widgets (contexto BuildContext)
{ return Scaffold (
    appBar: AppBar( title:
        Text('Layouts',
            style:
                TextStyle(color: Colors.black87), ), backgroundColor:
        Colors.white, iconTheme:
        IconThemeData(color: Colors.black54), brillo: Brillo.luz, principal:
        IconButton(icono: Icon(Icons.menu),
            onPressed: () {}), acciones: <Widget>[ IconButton(icon: Icon(Icons.cloud_queue),
            onPressed: () {}],

        ),
    cuerpo: _buildBody(), );
}
```

6. Agregue el método de widget `_buildBody()` después de Widget build (BuildContext context) {...}.
7. Devuelva un `SingleChildScrollView` con el elemento secundario como Columna. La lista de elementos secundarios de Columna está llamando a todos los métodos para crear cada sección de la página. Tenga en cuenta que el primer método, `_buildJournalHeaderImage()`, se coloca antes de SafeArea y Padding, lo que permite que la imagen ocupe todo el ancho del dispositivo.
8. Agregue una columna como elemento secundario de Padding y luego llame a los métodos `_buildJournalEntry()`, `_buildJournalWeather()`, `_buildJournalTags()` y `_buildJournalFooterImages()`.



```
Widget _buildBody()
{ devuelve SingleChildScrollView(
    niño: Column( niños:
        <Widget>[ _buildJournalHeaderImage(),
        SafeArea( niño: relleno(
            relleno: EdgeInsets.all(16.0), hijo:
            Column(
```

```
crossAxisAlignment: CrossAxisAlignment.start, niños: <Widget>[  
    _construirEntradaDiario(),  
    Divisor(),  
    _construirJournalWeather(),  
    Divisor(),  
    _buildJournalTags(),  
    Divider(),  
    _buildJournalFooterImages(), ], ), ), ), ], );  
}  
}
```

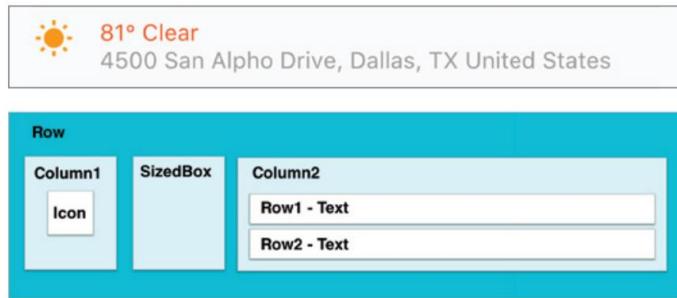
9. Cree el método `_buildJournalHeaderImage()` , que devuelve una imagen. La propiedad de imagen usa `AssetImage present.jpg` con el ajuste establecido en `BoxFit.cover`, lo que permite que la imagen ocupe todo el ancho del dispositivo.

```
Image _buildJournalHeaderImage() { return  
    Image( image:  
        AssetImage('assets/images/present.jpg'), fit: BoxFit.cover, );  
  
}
```

10. Cree el método `_buildJournalEntry()` , que devuelve una columna. La lista de elementos secundarios de columna de widgets contiene dos widgets de texto y un widget de divisor () .

```
Columna _buildJournalEntry() {  
    return  
        Column( crossAxisAlignment: CrossAxisAlignment.start,  
            children: <Widget>[ Text( 'Mi  
cumpleaños', estilo:  
                TextStyle( fontSize:  
                    32.0, fontWeight:  
                        FontWeight.bold, ), ), Divider(),  
  
                Text( 'Va a  
que  
sea un gran cumpleaños. Saldremos a cenar a mi lugar favorito, luego veremos una película  
después de ir a la heladería a tomar un helado y un espresso.',  
estilo: TextStyle(color: Colors.black54), ], );  
  
    }  
}
```

11. Cree el método `_buildJournalWeather()` , que devuelve una Fila. La lista de elementos secundarios de la fila contiene una columna, un cuadro de tamaño y otra columna. La lista de widgets de la segunda columna contiene dos widgets de fila .



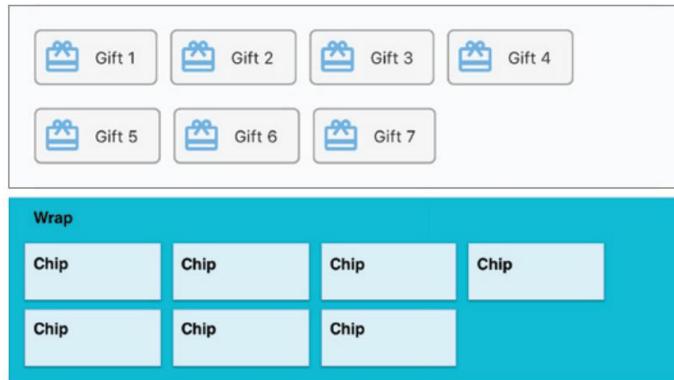
```

Fila _buildJournalWeather() { return Fila(
  mainAxisAlignment: MainAxisAlignment.start, children:
<Widget>[ Column( mainAxisAlignment: MainAxisAlignment.start,
  children:
<Widget>[ Icon( Icons.wb_sunny,
  color: Colors.orange, ),
],
),
SizedBox(ancho: 16.0,),

Columna( mainAxisAlignment: MainAxisAlignment.start, children:
<Widget>[ Row( children:
<Widget>[ Text( '81° Clear',
  style:
  TextStyle(color:
  Colors.deepOrange), ),
],
),
Row( children:
<Widget>[ Text( '4500 San Alpho Drive, Dallas, TX Estados Unidos',
  estilo: TextStyle(color: Colors.grey), ),
],
),
],
),
);
}
}

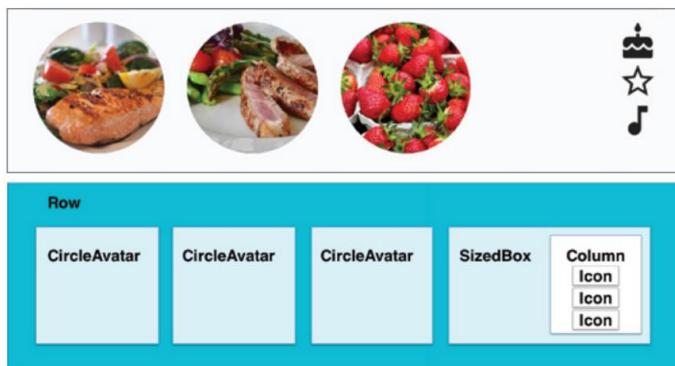
```

12. Cree el método `_buildJournalTags()`, que devuelve un Wrap. Los hijos Wrap usan el constructor `List.generate` para construir la lista de datos de muestra para mostrar siete valores de etiqueta de muestra. El constructor toma dos argumentos, la longitud de la lista y el índice.

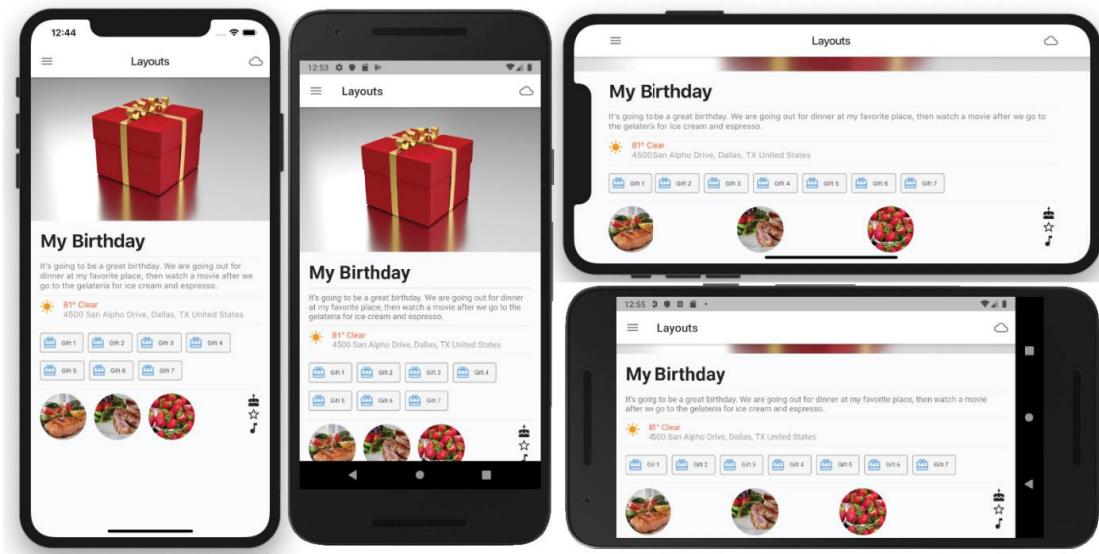


```
Envolver _buildJournalTags() {
    return
        Wrap(espaciado:
            8.0, hijos: List.generate(7, (índice int) { return Chip(
                etiqueta:
                    Texto('Regalo ${índice
                        + 1}', estilo: TextStyle(fontSize: 10.0), ),
                avatar: Icono(
                    Icons.card_giftcard,
                    color: Colors.blue.shade300, ),
                forma:
                    RoundedRectangleBorder( borderRadius:
                        BorderRadius.circular(4.0), side:
                    BorderSide(color: Colors.grey), ),
                backgroundColor: Colors.grey.shade100, ); })), );}
```

13. Cree el método `_buildJournalFooterImages()`, que devuelve una Fila. La lista de elementos secundarios de la fila contiene tres CircleAvatars y un SizedBox. El elemento secundario SizedBox es una columna con una lista de elementos secundarios de Widget de tres iconos. El objetivo principal de SizedBox es agregar espacio adicional entre CircleAvatar y los iconos colocados verticalmente.



```
Fila _buildJournalFooterImages() { devolver Fila(  
  
    mainAxisAlignment: MainAxisAlignment.spaceBetween, crossAxisAlignment:  
    CrossAxisAlignment.start, children: <Widget>[ CircleAvatar(  
  
        backgroundImage: AssetImage('assets/images/salmon.jpg'), radius: 40.0,  
  
>,  
        CírculoAvatar(  
            backgroundImage: AssetImage('assets/images/asparagus.jpg'), radius: 40.0,  
  
>,  
        CírculoAvatar(  
            backgroundImage: AssetImage('assets/images/strawberries.jpg'), radius: 40.0,  
  
>,  
        SizedBox(ancho: 100.0,  
            hijo: Columna(  
                crossAxisAlignment: CrossAxisAlignment.end, children:  
                <Widget>[ Icon(Icons.cake),  
                    Icon(Icons.star_border),  
                    Icon(Icons.music_note), //  
                    Icon(Icons.movie), ],  
  
>,  
>),  
    ], );  
}
```



CÓMO FUNCIONA

Primero creó una vista de alto nivel desglosando las secciones principales de la página. En la base, usó un SingleChildScrollView y construyó consecutivamente en la parte superior una columna, una imagen de encabezado , un área segura, un relleno y una columna con una lista secundaria de widgets que construyen cada sección por separado. Los dividió en cuatro secciones separadas y las creó llamando a los métodos _buildJournalEntry(), _buildJournal Weather(), _buildJournalTags() y _buildJournalFooterImages() . Cada uno de estos métodos crea un diseño personalizado anidando widgets.

RESUMEN

En este capítulo, aprendió cómo visualizar un diseño personalizado de alto nivel y dividirlo en sus secciones principales. Luego tomó cada sección principal y creó el diseño necesario anidando widgets.

En el próximo capítulo, aprenderá a agregar interactividad mediante los widgets GestureDetector, Draggable, DragTarget, InkWell y Dismissible .

LO QUE APRENDISTE EN ESTE CAPÍTULO

TEMA	CONCEPTOS CLAVE
Obtener una vista de alto nivel	Divide la página en secciones principales.
Creación de diseños simples y complejos.	Widgets separados y anidados.
Creación de un diseño personalizado	Diseñe y use widgets como SingleChildScrollView, SafeArea, Padding, Column, Row, Image, Divider, Text, Icon, SizedBox, Wrap, Chip, y CircleAvatar.

11

Aplicación de interactividad

LO QUE APRENDERÁS EN ESTE CAPÍTULO

Cómo usar GestureDetector, que reconoce gestos como tocar, tocar dos veces, pulsación larga, panorámica, arrastre vertical, arrastre horizontal y escala.

Cómo usar el widget Arrastrable que se arrastra a un DragTarget.

Cómo usar el widget DragTarget que recibe datos de un Draggable.

Cómo usar los widgets InkWell e InkResponse . Aprenderá que InkWell es un área rectangular que responde al tacto y recorta las salpicaduras dentro de su área.

Aprenderá que InkResponse responde al tacto y que las salpicaduras se expanden fuera de su área.

Cómo usar el widget Descartable que se descarta arrastrando.

En este capítulo, aprenderá a agregar interactividad a una aplicación mediante gestos. En una aplicación móvil, los gestos son el corazón de escuchar la interacción del usuario. Hacer uso de gestos puede definir una aplicación con una gran UX. El uso excesivo de gestos cuando no agregan valor o transmiten una acción crea una experiencia de usuario deficiente. Echará un vistazo más de cerca a cómo encontrar un equilibrio usando el gesto correcto para la tarea en cuestión.

CONFIGURACIÓN DEL DETECTOR DE GESTOS: CONCEPTOS BÁSICOS

El widget GestureDetector detecta gestos como toque, doble toque, pulsación larga, movimiento panorámico, arrastre vertical, arrastre horizontal y escala. Tiene una propiedad secundaria opcional y, si se especifica un widget secundario , los gestos se aplican solo al widget secundario . Si se omite el widget secundario , GestureDetector llena todo el elemento principal en su lugar. Si necesita atrapar el arrastre vertical y el arrastre horizontal al mismo tiempo, use el gesto panorámico. Si necesita atrapar un arrastre de un solo eje, use el gesto de arrastre vertical u horizontal.

Si intenta usar los gestos de arrastre vertical, arrastre horizontal y desplazamiento panorámico al mismo tiempo, recibirá un error de Argumentos de detector de gestos incorrectos . Sin embargo, si usa el arrastre vertical u horizontal con un gesto panorámico, no recibirá ningún error. La razón por la que recibe el error es que al tener simultáneamente un gesto de arrastre vertical y horizontal y un gesto panorámico, el gesto panorámico se ignora, ya que los otros dos (arrastre vertical y horizontal) capturarán primero todos los arrastres (Figura 11.1).

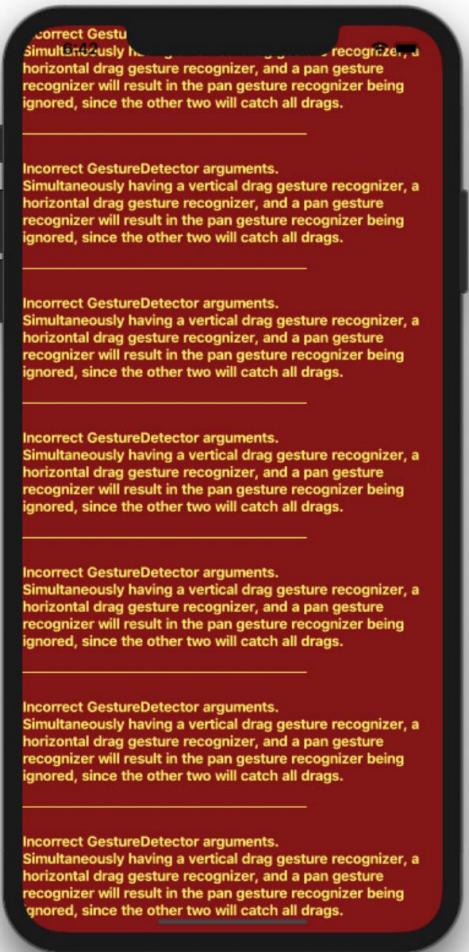


FIGURA 11.1: Error de gestos verticales, horizontales y panorámicos

Obtendrá el mismo tipo de error si intenta utilizar los gestos de arrastre vertical, arrastre horizontal y escala al mismo tiempo. Sin embargo, si usa el arrastre vertical u horizontal con un gesto de escala, no recibirá ningún error.

Cada propiedad de desplazamiento, arrastre vertical, arrastre horizontal y escala tiene una devolución de llamada para cada arrastre de inicio, actualización y finalización. (Consulte la Tabla 11.1.) Cada devolución de llamada tiene acceso al objeto de detalles que contiene valores sobre el gesto, que es rico en información y proporciona la posición táctil.

TABLA 11.1: Devoluciones de llamada de GestureDetector

PROPIEDAD/DEVOLUCIÓN DE LLAMADA	OBJETO DE DETALLES PARA DEVOLUCIÓN DE LLAMADA
onPanStart	ArrastrarInicioDetalles
onVerticalDragStart	ArrastrarInicioDetalles
onHorizontalDragStart	ArrastrarInicioDetalles
onScaleStart	EscalaInicioDetalles
onPanActualizar	ArrastrarActualizarDetalles
onVerticalDragUpdate	ArrastrarActualizarDetalles
onHorizontalDragUpdate	ArrastrarActualizarDetalles
onScaleUpdate	ScaleUpdateDetails
onPanEnd	DragEndDetails
onVerticalDragEnd	DragEndDetails
onHorizontalDragEnd	DragEndDetails
onScaleEnd	ScaleEndDetails

Por ejemplo, para verificar si un usuario arrastró en la pantalla desde la izquierda o la derecha, use la devolución de llamada onHorizontalDragEnd que tiene acceso al objeto de detalles DragEndDetails . Usas el valor details.primaryVelocity para verificar si es negativo, 'Arrastrado de derecha a izquierda', o si es positivo, 'Arrastrado de izquierda a derecha'.

```
onHorizontalDragEnd: (DragEndDetails detalles)
  { print('onHorizontalDragEnd: $detalles');

    if (detalles.primaryVelocity < 0)
      { print('Arrastrado de derecha a izquierda: ${detalles.primaryVelocity}'); } else
    if (detalles.primaryVelocity > 0) { print('Arrastrado de
      izquierda a derecha: ${detalles.primaryVelocity}'); } },

// resultados de la instrucción de
impresión flutter: onHorizontalDragEnd: DragEndDetails(Velocity(-2313.4, -110.3)) flutter:
Arrastrado de derecha a izquierda: -2313.4407865184226 flutter:
onHorizontalDragEnd: DragEndDetails(Velocity(3561.4, 123.2)) flutter: Arrastrado de
izquierda a derecha.34 258615
```

Los siguientes son los gestos de GestureDetector que puede escuchar y tomar las medidas adecuadas:

Pulse

al pulsar hacia abajo

onTapUp

en el toque

onTapCancelar

Doble toque

onDoubleTap

Pulsación larga

onLongPress

Sartén

onPanStart

onPanUpdate

en PanEnd

Arrastre vertical

onVerticalDragStart

onVerticalDragUpdate

en final de arrastre vertical

Arrastre horizontal

onHorizontalDragStart

onHorizontalDragUpdate

onHorizontalDragEnd

Escala

onScaleStart

onScaleUpdate

onScaleEnd

PRUÉBALO Creación de la aplicación de gestos, arrastrar y soltar

En esta sección, creará el área de gestos que captura los eventos de arrastre. Para que el área de gestos sea visible, utilizará un color verde claro y colocará un ícono de reloj de alarma únicamente con fines visuales. En la siguiente sección, agregará otra área a esta aplicación para manejar las capacidades de arrastre.



En este ejemplo, el widget Columna mostrará verticalmente un GestureDetector escuchando los gestos onTap, onDoubleTap, onLongPress y onPanUpdate . También mostrará un widget Arrastrable y un widget DragTarget . El ícono arrastrable pasará un color al objetivo de arrastre que muestra un widget de texto que cambia al color pasado. En este ejemplo, para mantener el árbol de widgets poco profundo, utilizará métodos en lugar de clases de widgets.

1. Cree un nuevo proyecto de Flutter y asígnele el nombre ch11_gestures_drag_drop. Puede seguir las instrucciones del Capítulo 4, “Creación de una plantilla de proyecto inicial”. Para este proyecto, solo necesita crear la carpeta de páginas .

2. Abra el archivo main.dart . Cambie la propiedad primarySwatch de azul a verde claro.

muestra primaria: Colors.lightGreen,

3. Abra el archivo home.dart y agregue al cuerpo un SafeArea con SingleChildScrollView como niño. La razón para usar SingleChildScrollView es manejar la rotación del dispositivo y poder desplazarse automáticamente para ver el contenido oculto. Agregue una columna como elemento secundario de SingleChildScrollView. Para la lista de elementos secundarios de Columna de Widget, agregue una llamada de método a _buildGestureDetector(), _buildDraggable() y _buildDragTarget() con widgets Divider entre ellos.

Tenga en cuenta que implementará _buildDraggable() y _buildDragTarget() en la siguiente sección. Puede comentar estos métodos si desea probar el proyecto solo con el Detector de gestos.

```
cuerpo: SafeArea(  
    hijo: SingleChildScrollView(  
        child:  
            Column( children:  
                <Widget>[ _buildGestureDetector(),  
  
                Divider( color: Colors.black,  
                    height: 10.0,),  
                _buildDraggable(),  
                _buildDragTarget(),  
            ],  
        ),  
    ),  
);
```

```
altura: 44,0, ),  
  
_buildDraggable(),  
  
Divider( altura:  
  
40,0, ), _buildDragTarget(),  
  
Divider( color:  
  
Colors.black, ), ], ), ), ),  
,
```

4. Agregue el método `_buildGestureDetector()` GestureDetector después del Widget
compilar (contexto de contexto de compilación) {...}.
5. Devuelva un GestureDetector escuchando onTap, onDoubleTap, onLongPress y
Gestos onPanUpdate .
6. Para ver los gestos capturados, agregue un Contenedor como elemento secundario del GestureDetector. El
contenedor secundario es una columna que muestra un ícono y un widget de texto que muestra
el gesto detectado y la ubicación del puntero en la pantalla. También agregué los gestos
(propiedades) onVerticalDragUpdate y onHorizontalDragUpdate , pero los comenté para que experimente.

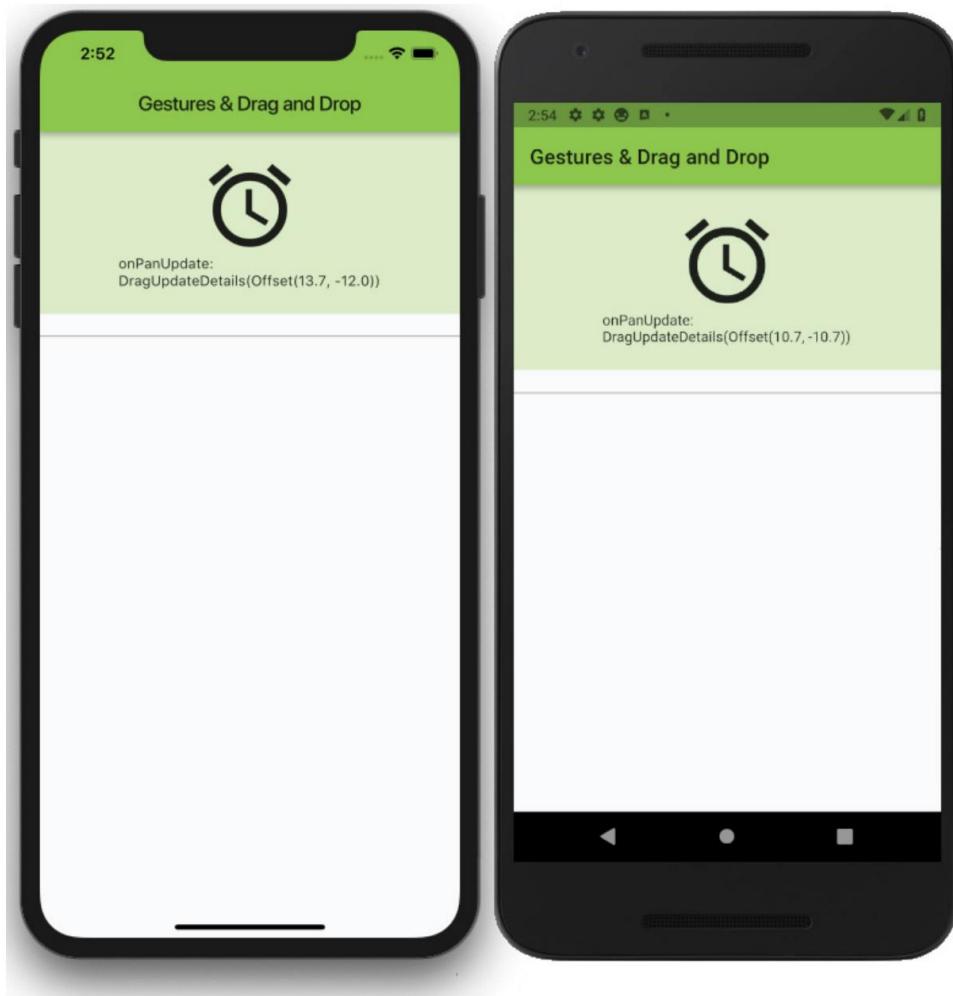
Recuerde que cuando use el gesto Pan puede escuchar solo en HorizontalDragUpdate o
enVerticalDragUpdate, no en ambos, o recibirá un error (consulte la Figura 11.1).

7. Para actualizar la pantalla con la ubicación del puntero y reutilizar el código, cree el
Método `_displayGestureDetected(gesto de cadena)` . Cada gesto pasa la representación String del gesto. Los
gestos (propiedades) onPanUpdate, onVerticalDragUpdate y onHorizontalDragUpdate escuchan
DragUpdateDetails.

En este ejemplo, estoy usando onPanUpdate, pero he dejado los gestos (propiedades)
onVerticalDragUpdate, onHorizontalDragUpdate y onHorizontalDragEnd comentados para que
experimente.

```
GestureDetector _buildGestureDetector() {  
  return GestureDetector(  
    onTap:  
    ()  
    { print('onTap');  
      _displayGestureDetected('onTap'); },  
  
    onDoubleTap: ()  
    { print('onDoubleTap'); },  
  );  
}  
  
void _displayGestureDetected(String gesture) {  
  print(gesture);  
}
```

```
_displayGestureDetected('onDoubleTap'); },  
  
onLongPress: ()  
{ print('onLongPress');  
_displayGestureDetected('onLongPress'); },  
  
onPanUpdate: (DragUpdateDetails detalles)  
{ print('onPanUpdate: $detalles');  
_displayGestureDetected('onPanUpdate:\n$detalles'); }, //  
  
onVerticalDragUpdate: ((DragUpdateDetails detalles) { //  
print('onVerticalDragUpdate: $detalles'); //  
_displayGestureDetected('onVerticalDragUpdate:\n$detalles'); //}, //  
  
onHorizontalDragUpdate: (DragUpdateDetails detalles ) { //  
print('onHorizontalDragUpdate: $detalles'); //  
_displayGestureDetected('onHorizontalDragUpdate:\n$detalles'); //}, //onHorizontalDragEnd:  
  
(DragEndDetails detalles) { // print('onHorizontalDragEnd:  
$detalles'); // if (detalles.primaryVelocity < 0)  
{ print('Arrastrando de derecha a izquierda: $  
{detalles.velocidad}'); // // } else if (detalles.primaryVelocity > 0)  
{ print('Arrastrando de izquierda a derecha: $  
{detalles.velocidad}'); // // } //, hijo: Contenedor(  
  
color: Colors.lightGreen.shade100, ancho:  
doble.infinito, relleno:  
EdgeInsets.all (24.0), niño: Columna  
(niños:  
<Widget>[ Icono  
  
(Iconos.access_alarm,  
tamaño: 98.0,),  
  
Texto ('$_gestureDetected '), ], ), ), );  
  
}  
  
void _displayGestureDetected(String gesto) { setState()  
{ _gestureDetected  
= gesto; }};
```



CÓMO FUNCIONA

GestureDetector escucha onTap, onDoubleTap, onLongPress y onPanUpdate y, para la aplicación, los gestos opcionales onVerticalDragUpdate y onHorizontalDragUpdate (propiedades). A medida que el usuario toca y arrastra sobre la pantalla, GestureDetector actualiza dónde comienza y termina el puntero, y mientras se mueve. Para limitar el área de detección de los gestos, pasó un contenedor como elemento secundario del Detector de gestos.

IMPLEMENTACIÓN DE LOS WIDGETS ARRASTRABLES Y DRAGTARGET

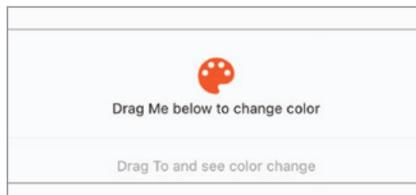
Para implementar una función de arrastrar y soltar, arrastre el widget Draggable a un widget DragTarget . Usa la propiedad de datos para pasar cualquier dato personalizado y usa la propiedad secundaria para mostrar un widget como un ícono y permanece visible mientras no se arrastra, siempre que la propiedad childWhenDragging sea nula. Configure la propiedad childWhenDragging para mostrar un widget mientras arrastra. Utilice la propiedad de comentarios para mostrar un widget que muestre los comentarios visuales del usuario sobre dónde se arrastra el widget. Una vez que el usuario levanta el dedo sobre DragTarget , el objetivo puede aceptar los datos. Para rechazar la aceptación de los datos, el usuario se aleja del DragTarget sin soltar el toque. Si necesita restringir el arrastre vertical u horizontalmente, puede configurar opcionalmente la propiedad Eje arrastrable . Para atrapar un arrastre de un solo eje, establezca la propiedad del eje en Axis.vertical o Axis.horizontal.

El widget DragTarget escucha un widget arrastrable y recibe datos si se suelta. La propiedad del constructor DragTarget acepta tres parámetros: BuildContext, List<dynamic> acceptData (candidateData) y List<dynamic> of addedData. Los datos aceptados son los datos pasados desde el widget arrastrable y espera que sea una lista de valores. Los datos rechazados contienen la Lista de datos que no serán aceptados.

PRUÉBALO Gestos: agregar arrastrar y soltar

En esta sección, agregará a la aplicación anterior un área de arrastre adicional que captura los eventos de arrastre. Creará dos widgets: un widget de ícono de paleta que se puede arrastrar por la pantalla y un widget de texto que recibe datos al aceptar un gesto de arrastre. Cuando el widget de texto recibe datos, cambiará el color del texto de un gris claro a un color naranja intenso, pero solo si se suelta el arrastre encima.

Para mantener limpio el ejemplo, la propiedad de datos arrastrables pasa el color naranja profundo como un valor entero. El DragTarget que acepta un valor entero comprueba si un Draggable está sobre él. De lo contrario, una etiqueta predeterminada muestra el mensaje "Arrastrar a y ver el cambio de color". Si Arrastrable está sobre él y los datos tienen un valor, verá una etiqueta con los datos pasados. El operador ternario (declaración condicional) se utiliza para comprobar si se pasan los datos .



Continuando con el proyecto de gestos anterior, agreguemos los métodos Draggable y DragTarget .

1. Cree el método _buildDraggable() , que devuelve un entero arrastrable. el arrastrable child es una columna con la lista de elementos secundarios de Widget que consta de un ícono y un widget de texto . La propiedad de retroalimentación es un ícono y la propiedad de datos pasa el color como un valor entero. El

La propiedad de datos puede ser cualquier dato personalizado necesario, pero para mantenerlo simple, está pasando el valor entero de Color.

```
Arrastrable<int> _buildDraggable() {
    return Arrastrable( child:
        Column( children:
            <Widget>[ Icon( Icons.palette,
                color:
                    Colors.deepOrange,
                    size: 48.0, ), Text('Arrástrame
                        abajo para
                            cambiar el color', ), ], childWhenDragging:
                    Icon( Icons.palette, color:
                        Colors.grey, size:
                            48.0, ), feedback:
                                Icon( Icons.brush,
                                    color:
                                        Colors.deepOrange, size: 80.0, ),
                                    data:
                                        Colors.deepOrange.value, );
    }
}
```

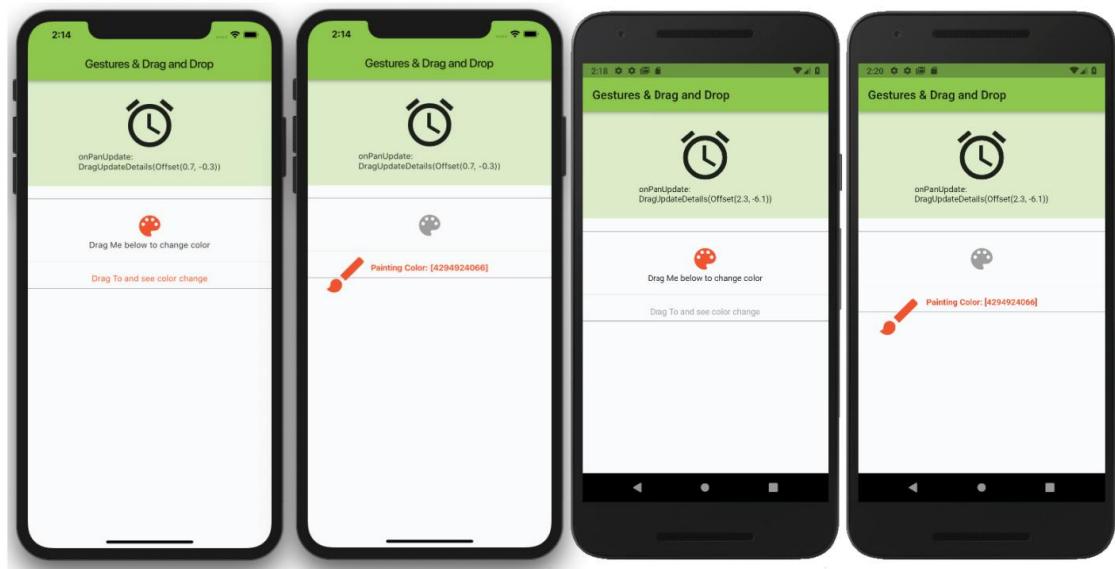
2. Cree el método `_buildDragTarget()` , que devuelve un entero DragTarget. Para aceptar datos, establezca el valor de la propiedad DragTarget `onAccept` en `colorValue` y establezca la variable `_paintedColor` en `Color(colorValue)`. `Color (colorValue)` construye (convierte) el valor entero en un color.

3. Establezca la propiedad del constructor para aceptar tres parámetros: `BuildContext`, `List<dynamic> acceptData`, y `List<dynamic>` de `addedData`. Tenga en cuenta que los datos son una Lista, y para obtener el valor entero de Color, lea el valor de la primera fila utilizando `acceptData[0]`.

Utilice la sintaxis de la flecha con el operador ternario para verificar `acceptData.isEmpty`. Si está vacío, significa que no se debe arrastrar sobre DragTarget y muestra un widget de texto que indica al usuario que "arrastre hacia y vea el cambio de color". De lo contrario, si se arrastra, DragTarget muestra el widget de texto que muestra el 'Color de pintura: \$acceptedData'. Para el mismo widget de texto , establezca la propiedad de estilo de color en `Color(acceptedData[0])` para proporcionar al usuario información visual sobre el color.

```
ArrastrarObjetivo<int> _buildDragTarget() {
    devuelve DragTarget<int>(
        al aceptar: (colorValor) {
```

```
_colorColor = Color(valorColor); }, constructor:  
  
(contexto BuildContext, Lista<dinámica> datos aceptados, Lista<dinámica>  
datosrechazados) => datosaceptados.isEmpty ?  
  
    Text('Arrastrar y ver el cambio de color', estilo:  
    TextStyle(color: _paintedColor),  
)  
:  
    Text('Color de la pintura: $datosaceptados',  
estilo: EstiloTexto(color:  
    Color(datosaceptados[0]), fontWeight:  
    FontWeight.bold, ), ), );  
  
}  
}
```



CÓMO FUNCIONA

El widget DragTarget escucha un widget que se puede arrastrar para soltar. Si el usuario suelta sobre DragTarget, se llama `onAccept`, siempre que los datos sean aceptables. La propiedad del constructor acepta tres parámetros: `BuildContext`, `List<dynamic> acceptData` (`candidateData`) y `List<dynamic>` de datos rechazados. Usando la sintaxis de flecha con un operador ternario, verifica `acceptData.isEmpty`. Si está vacío, no se pasan datos y un widget de texto muestra instrucciones. Si los datos son válidos y aceptados, se muestra un widget de texto con el valor de los datos y utiliza la propiedad de estilo para establecer el color adecuado.

USO DEL DETECTOR DE GESTOS PARA MOVER Y ESCALAR

Ahora se basará en lo que aprendió en la sección "Configuración de GestureDetector: Conceptos básicos" y, al profundizar más, aprenderá a escalar widgets mediante gestos individuales o multitáctiles. El objetivo es aprender a implementar el escalado multitáctil de una imagen acercando o alejando el zoom, tocando dos veces para aumentar el zoom y manteniendo presionado para restablecer la imagen al tamaño original. El GestureDetector le brinda la capacidad de lograr la escala mediante el uso de `onScaleStart` y `onScaleUpdate`. Use `onDoubleTap` para aumentar el zoom y use `onLongPress` para restablecer el zoom al tamaño predeterminado original.

Cuando el usuario toca la imagen, la imagen se puede arrastrar para cambiar de posición o escalar acercándola o alejándola. Para cumplir con ambos requisitos, utilizará el widget Transformar . Use el constructor `Transform.scale` para cambiar el tamaño de la imagen y use el constructor `Transform.translate` para mover la imagen (Figura 11.2).

El widget Transformar aplica una transformación antes de que se pinte el elemento secundario . Usando el constructor predeterminado `Transform` , el argumento `transform` se establece usando la clase `Matrix4` (4D Matrix), y esta matriz de transformación se aplica al niño durante la pintura. Los beneficios de usar el constructor predeterminado son usar `Matrix4` para ejecutar múltiples transformaciones en cascada `(..scale()..translate())` . Los puntos dobles (...) se utilizan para varias transformaciones en cascada. En el Capítulo 3, "Aprendizaje de los conceptos básicos de Dart", aprendió que la notación en cascada le permite realizar una secuencia de operaciones en el mismo objeto. El siguiente código de muestra muestra cómo usar la clase `Matrix4` con la notación en cascada para aplicar una escala y una transformación de traducción al mismo objeto:

```
Matrix4.identidad()  
..escala(1.0,  
1.0) ..traducir(30, 30);
```

El widget Transform tiene cuatro constructores diferentes.

`Transform`: constructor predeterminado que toma `Matrix4` como argumento de transformación.

`Transform.rotate`: Constructor para rotar un widget secundario alrededor del centro usando un ángulo. El argumento del ángulo gira en el sentido de las agujas del reloj en radianes. Para girar en sentido contrario a las agujas del reloj, pase un radián negativo.

`Transform.scale`: constructor para escalar uniformemente un widget secundario en el eje x y el eje y. El widget se escala por su alineación central. El valor del argumento de escala de 1.0 es el tamaño original del widget. Cualquier valor por encima de 1,0 escala el widget más grande, y los valores por debajo de 1,0 escalan el widget más pequeño. Un valor de 0,0 hace que el widget sea invisible.

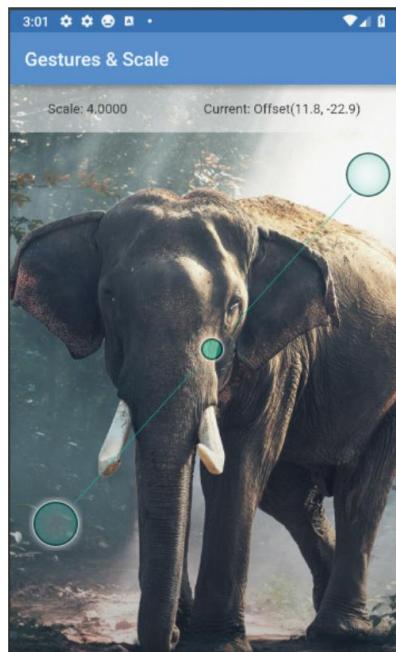
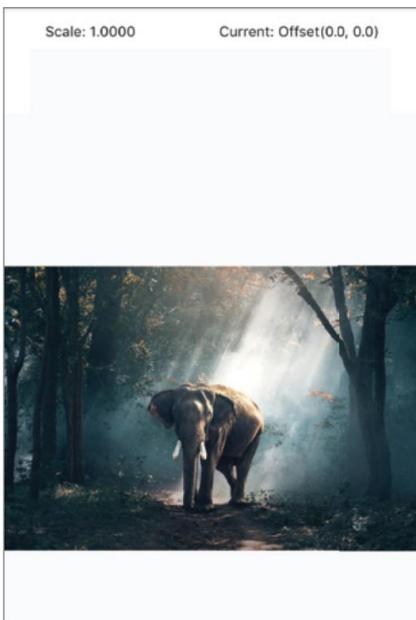


FIGURA 11.2: Mover y escalar una imagen

`Transform.translate`: constructor para mover/posicionar un widget secundario mediante una traducción, un desplazamiento. El argumento de desplazamiento toma la clase `Offset(doble dx, doble dy)` colocando el widget en el eje x y el eje y.

PRUÉBALO Creación de una aplicación con gestos para mover y escalar

En este ejemplo, la aplicación presenta una página que muestra una imagen que tiene el ancho total del dispositivo. Imagíñese si esta es una aplicación de diario y el usuario navegó a esta página para ver la imagen seleccionada con la capacidad de hacer zoom para obtener más detalles. La imagen se mueve con un solo toque de arrastre y se puede acercar/alejar (pellizcar) usando multitoque. Tocar dos veces permite que la imagen se acerque en la ubicación tocada, y una sola pulsación prolongada restablece la ubicación de la imagen y el nivel de zoom a los valores predeterminados.



Este ejemplo es la primera parte de la aplicación en la que diseña los widgets que manejan los gestos que mueven y escalan la imagen. En el próximo ejercicio, se concentrará en agregar la lógica para manejar los cálculos necesarios para realizar un seguimiento de la ubicación y la escala de la imagen.

`GestureDetector` es el widget base del cuerpo (propiedad) y escucha los gestos (propiedades) `onScaleStart`, `onScaleUpdate`, `onDoubleTap` y `onLongPress`. El elemento secundario `GestureDetector` es una pila que muestra la imagen y una barra de estado de gestos. Para aplicar el movimiento y la escala de la imagen, use el widget `Transform`. Para mostrar diferentes formas de aplicar cambios a un widget, utiliza tres constructores de transformación diferentes: predeterminado, escalar y traducir.

Echará un vistazo a dos técnicas para lograr los mismos resultados de movimiento y escala. La primera técnica (que comienza con el paso 13) consiste en anidar los constructores de escala y traducción. La segunda técnica (a partir del paso 16) utiliza el constructor predeterminado con `Matrix4` para aplicar transformaciones.

Tenga en cuenta que en este ejemplo, para mantener el árbol de widgets poco profundo, utilizará métodos en lugar de clases de widgets.

1. Cree un nuevo proyecto de Flutter y asignele el nombre ch11_gestures_scale. Como de costumbre, puede seguir las instrucciones del Capítulo 4. Para este proyecto, necesita crear solo las páginas y las carpetas de activos/ímágenes .

2. Cree la clase de inicio como StatelessWidget ya que los datos (estado) requieren cambios.

3. Abra el archivo pubspec.yaml para agregar recursos. En la sección de activos , agregue los activos/ imágenes/ carpeta.

Para agregar activos a su aplicación, agregue una sección de activos, como esta: activos:

- activos/ímágenes/

4. Haga clic en el botón Guardar y, según el Editor que esté utilizando, se ejecuta automáticamente el paquetes de aleteo obtener; una vez terminado, mostrará un mensaje de Proceso finalizado con el código de salida 0. Si no ejecuta automáticamente el comando por usted, abra la ventana de Terminal (ubicada en la parte inferior de su editor) y escriba flutter packages get.

5. Agregue los activos de la carpeta y las imágenes de la subcarpeta en la raíz del proyecto y luego copie el elefante. jpg archivo a la carpeta de imágenes .

6. Abra el archivo home.dart y agregue al cuerpo una llamada al método _buildBody(context) . El Método _buildBody() recibe el contexto como un parámetro para que MediaQuery del método obtenga el ancho del dispositivo.

cuerpo: _buildBody(contexto),

7. Debajo de la clase _HomeState se extiende State<Home> y arriba @override, agregue las variables para _startLastOffset, _lastOffset, _currentOffset, _lastScale y _currentScale.

Las variables _startLastOffset, _lastOffset y _currentOffset son del tipo Offset inicializadas con un valor de Offset.zero. Offset.zero es lo mismo que Offset(0.0, 0.0), es decir , la posición predeterminada de la imagen. Estas variables se utilizan para realizar un seguimiento de la posición de la imagen mientras se arrastra.

Las variables _lastScale y _currentScale son de tipo doble. Se inicializan con un valor de 1,0, que es el tamaño de zoom normal. Estas variables se utilizan para realizar un seguimiento de la imagen mientras se escala. Los valores superiores a 1,0 escalan la imagen más grande y los valores inferiores a 1,0 escalan la imagen más pequeña.

```
class _HomeState extiende State<Home> {
    Desplazamiento _startLastOffset =
        Desplazamiento.cero; Desplazamiento
    _lastOffset = Desplazamiento.cero; Desplazamiento
    _currentOffset =
        Desplazamiento.cero; doble _lastScale = 1.0; double _currentScale = 1.0;

    @anular
    Compilación del widget (contexto BuildContext) {
```

8. Agregue el método Widget _buildBody(BuildContext context) después del Widget compile (contexto de contexto de compilación) {...}. Devuelve un GestureDetector con el niño como una pila. Tenga en cuenta que GestureDetector está en la raíz de la propiedad del cuerpo para interceptar todos los gestos en cualquier lugar de la pantalla.

Para mostrar la imagen y la visualización de la barra de estado de gestos superior, utilizará el widget Apilar .

9. Establezca la propiedad Stack fit en StackFit.expand para expandirse al tamaño más grande permitido.
10. Para la lista de elementos secundarios Stack de Widget, agregue tres métodos y asígneles el nombre _transformScale AndTranslate(), _transformMatrix4() y _positionedStatusBar(context). El método _positionedStatusBar pasa el contexto de MediaQuery para obtener el ancho completo del dispositivo.
11. Comente el método _transformMatrix4() ya que estará probando con el _transformScaleAndTranslate() primero.

12. Agregue al GestureDetector onScaleStart , onScaleUpdate, onDoubleTap y onLongPress gestos (propiedades) para escuchar cada gesto. Pase respectivamente los métodos _onScaleStart, _onScaleUpdate, _onDoubleTap y _onLongPress .

```
Widget _buildBody (contexto BuildContext) {  
    return GestureDetector( child:  
        Stack( fit:  
            StackFit.expand, children:  
  
                <Widget>[ _transformScaleAndTranslate(), //  
                    _transformMatrix4(),  
                    _positionedStatusBar(context), ],  
  
                ),  
                onScaleStart: _onScaleStart,  
                onScaleUpdate: _onScaleUpdate,  
                onDoubleTap: _onDoubleTap,  
                onLongPress: _onLongPress, );  
  
}
```

13. En este paso, implementa la primera técnica (mover y escalar) anidando la escala y traducir constructores. Agregue el método de transformación _transformScaleAndTranslate() después de la creación del widget (contexto BuildContext) {...}. Devuelve un Transform anidando la escala y traducir constructores.

14. Para el argumento de escala del constructor Transform.scale , ingrese la variable _currentScale y establezca el argumento secundario en el constructor Transform.translate .

15. Para el argumento de compensación del constructor Transform.translate , ingrese la variable _current_offset y establezca el argumento secundario en una imagen. Este widget secundario es la imagen que se arrastra y escala al anidar los dos widgets de transformación .

```
Transformar _transformScaleAndTranslate() {
    return Transform.scale( escala:
        _currentScale, child:
            Transform.translate( offset: _currentOffset,
                child: Image(
                    imagen: AssetImage('assets/images/elephant.jpg'), ), ), );
}

}
```

16. En este paso, implementará la segunda técnica (mover y escalar) utilizando el valor predeterminado constructor. Agregue el método de transformación `_transformMatrix4()`. Devuelve un `Transform` utilizando el constructor predeterminado .

17. Para el argumento `transform` del constructor `Transform` , use `Matrix4`. Uso de `Matrix4`. `Identity()` crea la matriz desde cero y establece los valores predeterminados. En realidad, está haciendo una llamada a `Matrix4.zero().setIdentity()`. Desde el constructor de identidad , use los puntos dobles para escalar en cascada y traducir transformaciones. Con esta técnica, no hay necesidad de usar varios widgets de transformación : solo usa uno y ejecuta varias transformaciones.

18. Para el método de escala , pase `_currentScale` para los ejes x e y. El eje x es mandatory, pero el eje y es opcional. Dado que la imagen se escala proporcionalmente, se utilizan los valores del eje x y del eje y.

19. Para el método de traducción , pase `_currentOffset.dx` para el eje x y `_currentOffset.dy` para el eje y. El eje x es obligatorio, pero el eje y es opcional. En esta aplicación, la imagen se arrastra (se mueve) sin restricciones y se utilizan los valores del eje x y del eje y.

20. Para mantener la imagen alineada en el centro durante el escalado, configure la propiedad de alineación para usar el `FractionalOffset.center`. Si no se utiliza la alineación central mientras se escala la imagen, el `translate` `_currentOffset` mueve la imagen. Al mantener la imagen en la misma ubicación durante el escalado, crea una gran experiencia de usuario. Si la imagen se aleja de la ubicación actual durante el escalado, no sería una buena experiencia de usuario.

Establezca la propiedad secundaria en un widget de imagen . La propiedad de imagen utiliza el archivo `AssetImage elephant.jpg`.

```
Transformar _transformMatrix4() {
    return Transform(transformar:
        Matrix4.identity() ..scale(_currentScale,
            _currentScale)..translate(_currentOffset.dx,
                _currentOffset.dy,),,
        alineación: FractionalOffset.center, child:
            Image( image:
                AssetImage('assets/images/elephant.jpg'), ),
    );
}
```

21. Agregue el método Positioned _positionedStatusBar(BuildContext context) . devolver un Posicionado mediante el uso del constructor predeterminado . El propósito de este widget Posicionado es mostrar una barra de estado de gestos en la parte superior de la pantalla con la escala y la posición actuales.

22. Establezca la propiedad top en 0.0 para colocarla en la parte superior de la pantalla en el widget Stack . Selecciona el width mediante el uso de MediaQuery width para expandir todo el ancho del dispositivo. El elemento secundario es un contenedor con la propiedad de color establecida en un tono de Colors.white54. Establezca la propiedad Altura del contenedor en 50,0. Establezca el contenedor secundario en una fila con mainAxisAlignment de MainAxisSizeAlignment.spaceAround. Para la lista de filas secundarias de Widget, use dos widgets de texto . El primer widget de texto muestra la escala actual mediante la variable _currentScale . Para mostrar solo una precisión de hasta cuatro puntos decimales, use _currentScale.toStringAsFixed(4). El segundo widget de texto muestra la ubicación actual mediante la variable _currentOffset .

```
Posicionado _positionedStatusBar (contexto BuildContext) {
    return Positioned( top:
        0.0, width:
        MediaQuery.of(context).size.width, child:
        Container( color:
            Colors.white54, height:
            50.0, child:
            Row( mainAxisAlignment: MainAxisSizeAlignment.spaceAround,
                children:
                    <Widget>[ Text( 'Escala: ${_currentScale.toStringAsFixed(4)}', ),
                    Text( 'Actual: ${_currentOffset}', ), ], ), ), ); }

}
```

CÓMO FUNCIONA

GestureDetector escucha los gestos onScaleStart, onScaleUpdate, onDoubleTap y onLongPress (propiedades). A medida que el usuario toca y arrastra sobre la pantalla, GestureDetector actualiza dónde comienza y termina el puntero, además de actualizar su ubicación mientras se mueve. Para maximizar el área de detección de gestos, GestureDetector llena toda la pantalla configurando la propiedad de ajuste del widget Stack secundario en StackFit.expand para expandirse al tamaño más grande permitido. Tenga en cuenta que GestureDetector llena toda la pantalla solo si no se usa ningún widget secundario .

Las variables _startLastOffset, _lastOffset y _currentOffset son del tipo Offset y se utilizan para realizar un seguimiento de la posición de la imagen. Las variables _lastScale y _currentScale son de tipo doble y se utilizan para realizar un seguimiento de la escala de la imagen.

El uso del widget Apilar le permite colocar la Imagen y un widget Posicionado para que se superpongan entre sí.

El widget Posicionado se coloca como el último widget en la pila para permitir que la barra de estado de gestos permanezca sobre la imagen.

El widget Transform se utiliza para mover y escalar la imagen mediante la implementación de dos técnicas diferentes. La primera técnica utiliza el anidamiento de los constructores Transform.scale y Transform.translate . La segunda técnica usa el constructor predeterminado Transform usando Matrix4 para aplicar transformaciones. Ambos métodos logran los mismos resultados.

PRUÉBALO Adición de lógica y cálculos al proyecto de movimiento y escalado

Continúe con el proyecto de movimiento y escalado anterior y comience a agregar la lógica y los cálculos para manejar los gestos de GestureDetector . onScaleStart y onScaleUpdate son responsables de manejar los gestos de movimiento y escala. onDoubleTap es responsable de manejar el gesto de doble toque para aumentar el zoom. onLongPress es responsable de manejar un gesto de presión prolongada para restablecer el zoom al valor predeterminado original.

1. Cree el método _onScaleStart(ScaleStartDetails detalles) . Este gesto se llama una vez cuando el usuario comienza a mover o escalar la imagen. Este método rellena las variables _startLastOffset, _lastOffset y _lastScale , y los cálculos de posición y escala de la imagen se basan en estos valores.

```
void _onScaleStart(ScaleStartDetails detalles)
{ print('ScaleStartDetails: $detalles');

  _startLastOffset = detalles.puntofocal; _lastOffset =
  = _currentOffset; _lastScale =
  _currentScale;
}
```

2. Cree el método _onScaleUpdate(ScaleUpdateDetails detalles) . Este gesto se llama cuando el usuario está moviendo o escalando la imagen.

Al usar el objeto de detalles (ScaleUpdateDetails from callback), se pueden verificar diferentes valores como la escala, la rotación y el punto focal (Desplazamiento de la posición del contacto en la pantalla del dispositivo). El objetivo es verificar si el usuario está moviendo o escalando la imagen al verificar el valor de detalles.escala .

```
void _onScaleUpdate(ScaleUpdateDetails detalles)
{ print('ScaleUpdateDetails: $detalles - Escala: ${detalles.escala}');
```

3. Configure una declaración if para verificar si el usuario está escalando la imagen evaluando los detalles. escala != 1.0. Si la escala de detalles es mayor o menor que 1.0, significa que la imagen se está escalando. Con valores superiores a 1,0, la imagen se escala más y con valores inferiores a 1,0, la imagen se escala más pequeña. void

```
_onScaleUpdate(ScaleUpdateDetails detalles)
{ print('ScaleUpdateDetails: $detalles - Escala: ${detalles.escala}');

  if (detalles.escala!= 1.0) { //
    Escalado
}}
```

4. Para calcular la escala actual, cree una variable local llamada `escalaActual` de tipo doble y calcule el valor tomando `_lastScale * detalles.escala`. `_lastScale` se calculó previamente a partir del método `_onScaleStart()`, y `detalles.scale` es la escala actual a medida que el usuario continúa haciendo zoom en la imagen .

```
double currentScale = _lastScale * detalles.escala;
```

5. Para restringir la escala de la imagen a más de la mitad de su tamaño, compruebe si la escala actual es inferior a 0,5 (la mitad del tamaño original) y restablezca el valor a 0,5. if

```
(currentScale < 0.5)
{ currentScale = 0.5;
}
```

6. Para que el widget Imagen actualice el zoom actual, agregue el método `setState()` y complete el valor `_currentScale` con la variable local `currentScale` . Recuerda que el método `setState()` le dice al marco Flutter que el widget debe volver a dibujarse porque el estado ha cambiado. Se recomienda colocar los cálculos que no necesitan cambios de estado fuera del método `setState()` . Esta mejor práctica es la razón por la que creó una variable local llamada `escalaActual`.

```
setState()
{ _currentScale = currentScale; };
```

El siguiente es el método `_onScaleUpdate()` actual :

```
void _onScaleUpdate(ScaleUpdateDetails detalles)
{ print('ScaleUpdateDetails: $detalles - Escala: ${detalles.escala}');
if (detalles.escala!= 1.0) { //
  Escalado
  doble escalaActual = _últimaEscala * detalles.escala; if
  (currentScale < 0.5)
  { currentScale = 0.5;

  } setState()
  { _currentScale = currentScale; });

  print('_scale: ${_currentScale} - _lastScale: ${_lastScale}');
}
}
```

7. Continúe agregando al método `_onScaleUpdate()` la lógica para mover la imagen por la pantalla.

8. Agregue una instrucción else if para verificar si la escala de `detalles` es igual a 1.0. Si la escala es 1.0, significa que la imagen se está moviendo, no escalando.

```
} else if (detalles.escala == 1.0) {
```

Antes de poder mover la imagen, debe tener en cuenta la escala actual, ya que afecta el Desplazamiento (posición).

9. Cree la variable local `offsetAdjustedForScale` del tipo `Offset`. Esta variable contiene el último desplazamiento teniendo en cuenta el factor de escala. Tome el `_startLastOffset` y reste el `_lastOffset`; luego divida el resultado por `_lastScale`.

```
// Calcular el desplazamiento dependiendo de la escala de la imagen actual.  
Offset offsetAdjustedForScale = (_startLastOffset - _lastOffset) / _lastScale;
```

10. Cree la variable `currentOffset` local de tipo `Offset`. Esta variable contiene el desplazamiento actual (posición) de la imagen. El `currentOffset` se calcula tomando los `detalles.focalPoint` menos el `offsetAdjustedForScale` multiplicado por `_currentScale`.

```
Offset currentOffset = detalles.focalPoint - (offsetAdjustedForScale * _currentScale);
```

11. Para que el widget `Imagen` actualice la posición actual, agregue el método `setState()` y complete el valor `_currentOffset` con la variable local `currentOffset` .

```
setState()  
{ _currentOffset = currentOffset; };
```

El siguiente es el método `_onScaleUpdate()` completo:

```
void _onScaleUpdate(ScaleUpdateDetails detalles)  
{ print('ScaleUpdateDetails: $detalles - Escala: ${detalles.escala}');  
  
if (detalles.escala == 1.0) { // Escalado  
    doble  
    escalaActual = _últimaEscala * detalles.escala; if (currentScale <  
    0.5) { currentScale = 0.5;  
  
} setState()  
{ _currentScale = currentScale; };  
  
print('_scale: ${_currentScale} - _lastScale: ${_lastScale}'); } else if (detalles.escala  
== 1.0) { // No estamos escalando sino  
    arrastrando por la pantalla // Calcula el desplazamiento  
    dependiendo de la escala de la imagen actual.  
    Offset offsetAdjustedForScale = (_startLastOffset - _lastOffset) / _lastScale; Offset  
    currentOffset  
    = detalles.focalPoint - (offsetAdjustedForScale * _currentScale); setState(){ _currentOffset =  
    currentOffset; };  
  
    print('offsetAdjustedForScale:  
    $offsetAdjustedForScale - _currentOffset: ${_currentOffset}');  
  
}
```

12. Cree el método `_onDoubleTap()`. Este gesto se llama cuando el usuario toca dos veces la pantalla. Cuando se detecta un doble toque, la imagen se escala el doble de grande.
13. Cree la variable local `currentScale` de tipo doble. La escala actual se calcula multiplicando `_lastScale` por 2,0 (el doble del tamaño).

```
double currentScale = _lastScale * 2.0;
```

14. Agregue una declaración `if` que verifique si `currentScale` es mayor que 16.0 (16 veces el tamaño original); luego restablezca `currentScale` a 1.0. Agregue una llamada al método `_resetToDefaultValues()` (creado en el paso 19) que restablece todas las variables a sus valores predeterminados. El resultado es que la imagen se vuelve a centrar y se escala al tamaño original.

15. Después de la instrucción `if`, establezca la variable `_lastScale` en la variable local `currentScale`.

16. Para que el widget `Imagen` actualice la escala actual, agregue el método `setState()` y complete el valor `_currentScale` con la variable local `currentScale`.

```
vacío _onDoubleTap () {
    imprimir('enDobleToque');

    // Calcule la escala actual y complete _lastScale con currentScale // si currentScale es mayor que 16
    // veces la imagen original, restablezca la escala
    por defecto, 1.0
    double currentScale = _lastScale * 2.0; if (currentScale
        > 16.0) { currentScale = 1.0;
        _resetToDefaultValues(); }
    _lastScale = currentScale;

    setState()
    { _currentScale = currentScale; });
}
```

17. Cree el método `_onLongPress()`. Este gesto se llama cuando el usuario está presionando y manteniendo presionada la pantalla. Cuando se detecta una pulsación prolongada, la imagen se restablece a su posición y escala originales.

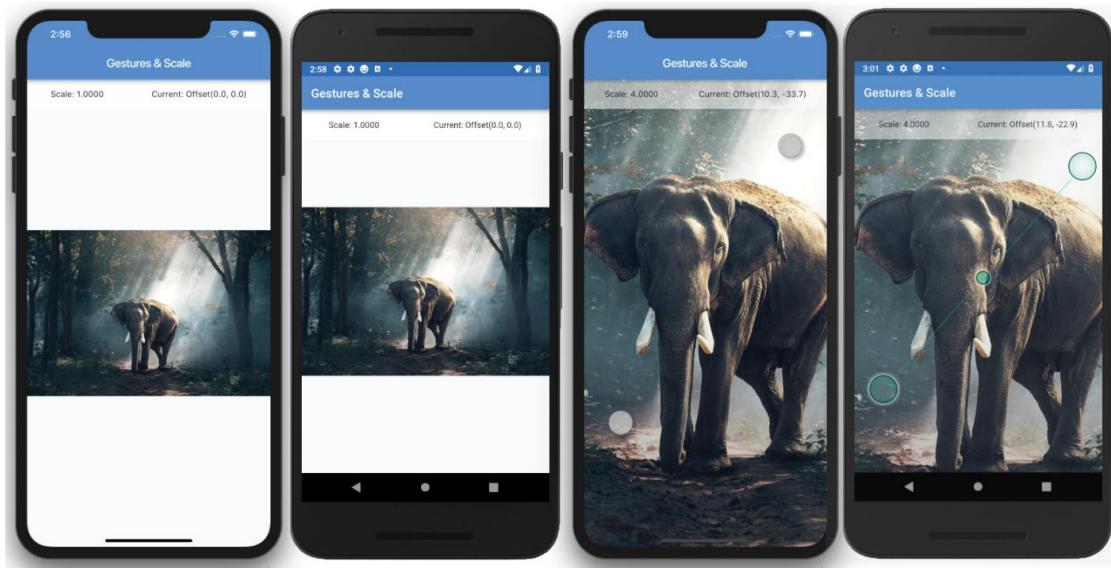
18. Para que el widget `Imagen` se actualice a la posición y escala predeterminadas, agregue el método `setState()` y llame al método `_resetToDefaultValues()`.

```
vacío _onLongPress () {
    imprimir('enPresionLarga');

    setState()
    { _resetToDefaultValues(); });
}
```

19. Cree el método `_resetToDefaultValues()` . El propósito de este método es restablecer todos los valores a los valores predeterminados con el widget Imagen centrado en la pantalla y reducido al tamaño original. La creación de este método es una excelente manera de compartir la reutilización de código desde cualquier lugar de la página.

```
void _resetToDefaultValues() {  
    _startLastOffset = Desplazamiento.cero;  
    _lastOffset = Desplazamiento.cero;  
    _currentOffset = Offset.cero;  
    _últimaEscala = 1.0;  
    _escalaActual = 1.0; }
```



CÓMO FUNCIONA

Se llama al método `_onScaleStart()` cuando un toque comienza a mover o escalar la imagen por primera vez. Las variables `_startLastOffset`, `_lastOffset` y `_lastScale` se completan con los valores necesarios para colocar y escalar correctamente la imagen mediante el método `_onScaleUpdate()` .

Se llama al método `_onScaleUpdate()` cuando el usuario está moviendo o escalando la imagen. El objeto de detalles (`ScaleUpdateDetails`) contiene valores como la escala, la rotación y el punto focal. Al usar el valor de `detalles.escala` , verifica si la imagen se está moviendo o escalando. Si la escala de detalles es mayor que 1,0, la imagen se escala más grande, y si la escala de detalles es menor que 1,0, la imagen se escala más pequeña. Si la escala de detalles es igual a 1,0, la imagen se está moviendo.

El método `_onDoubleTap()` se llama cuando el usuario toca dos veces la pantalla. Con cada doble toque, la imagen se escala el doble de grande y, una vez que alcanza 16 veces el tamaño original, la escala se restablece a 1,0, el tamaño original.

El método `_onLongPress()` se llama cuando el usuario presiona y mantiene presionada la pantalla. Cuando se detecta este gesto, la imagen se restablece a su posición y escala originales llamando al método `_resetToDefaultValues()`.

El método `_resetToDefaultValues()` es responsable de restablecer todos los valores a los valores predeterminados. El widget `Imagen` se mueve de vuelta al centro de la pantalla y se escala al tamaño original.

USO DE LOS GESTOS INKWELL Y INKRESPONSE

Tanto los widgets `InkWell` como `InkResponse` son componentes materiales que responden a gestos táctiles. La clase `InkWell` extiende (subclase) la clase `InkResponse`. La clase `InkResponse` extiende una clase `StatefulWidget`.

Para `InkWell`, el área que responde al tacto tiene forma rectangular y muestra un efecto de "salpicadura", aunque en realidad parece una onda. El efecto de salpicadura se recorta en el área rectangular del widget (para no salirse de él). Si necesita expandir el efecto de salpicadura fuera del área rectangular, `InkResponse` tiene una forma configurable. De forma predeterminada, `InkResponse` muestra un efecto de salpicadura circular que puede expandirse fuera de su forma (Figura 11.3).

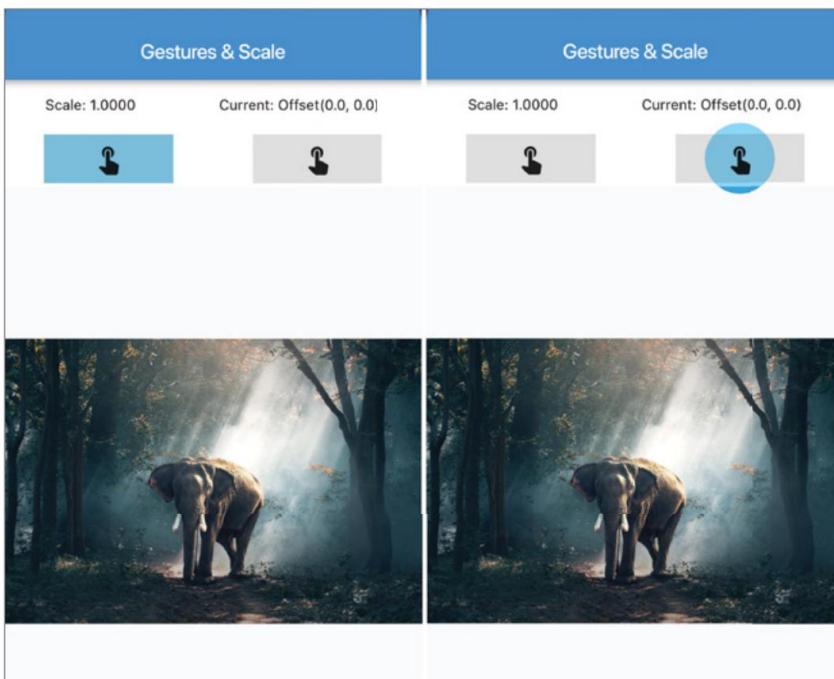


FIGURA 11.3: Splash de `InkWell` y `InkResponse`

Tocar InkWell muestra cómo aparece el efecto de salpicadura de forma incremental (Figura 11.4). En este escenario, el usuario tocó el botón izquierdo y se muestra el efecto de salpicadura cambiando gradualmente el color del botón de gris a azul. El color de la salpicadura permanece dentro del área rectangular del botón.

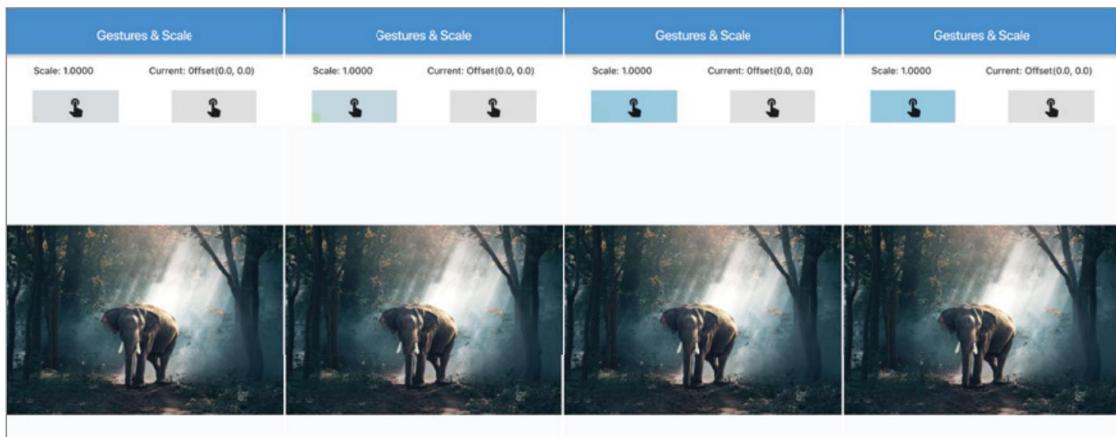


FIGURA 11.4: Salpicadura gradual de InkWell dentro del área rectangular

Tocar InkResponse muestra cómo aparece el efecto de salpicadura de forma incremental (Figura 11.5). En este escenario, el usuario tocó el botón derecho y muestra el efecto de salpicadura circular que cambia gradualmente el color del botón de gris a azul. El color de salpicadura se expande circularmente fuera del área rectangular del botón.

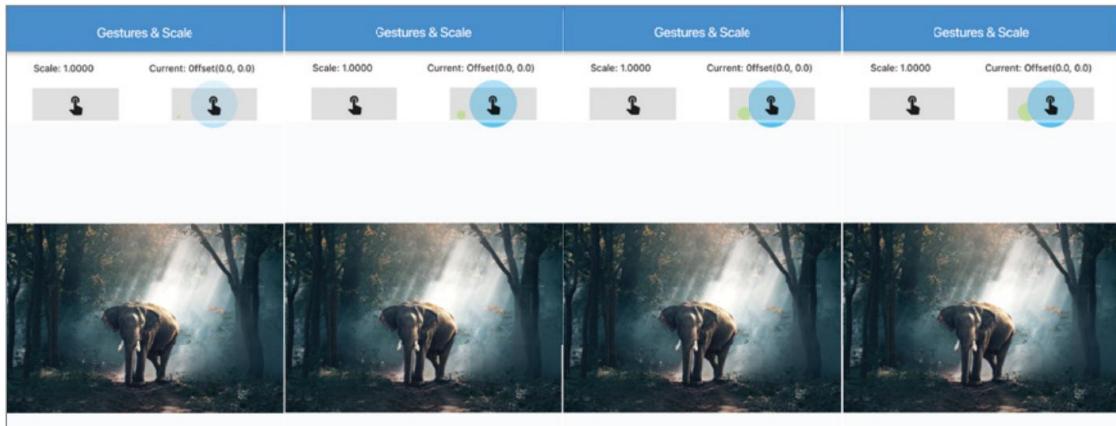


FIGURA 11.5: Salpicadura gradual de InkResponse fuera del área rectangular

Los siguientes son los gestos de InkWell e InkResponse que puede escuchar y tomar las medidas adecuadas. Los gestos capturados son toques en la pantalla, excepto la propiedad `onHighlightChanged`, que se activa cuando parte del material comienza o deja de resaltarse.

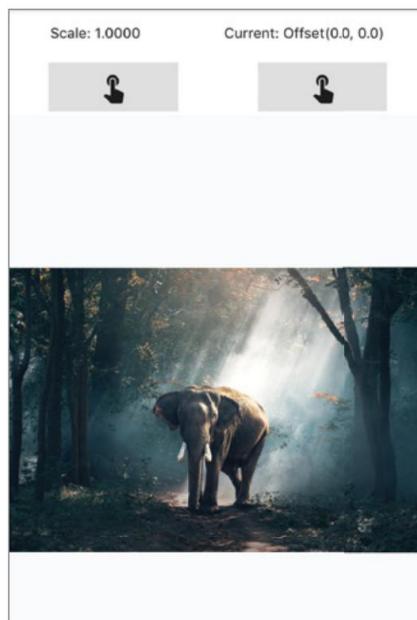
Pulse

en el toque

al pulsar hacia abajo
onTapCancelar
Doble toque
onDoubleTap
Pulsación larga
onLongPress
Destacado cambiado
onHighlightChanged

PRUÉBELO Adición de InkWell e InkResponse a la barra de estado de gestos

Continuando con el proyecto de gestos anterior para comparar el rendimiento de cada widget, agregará InkWell e InkResponse a la visualización de la barra de estado de gestos actual.



1. En el método `_buildBody(BuildContext context)` , agregue una llamada al Método `_positionedInkWellAndInkResponse(contexto)` . Coloque esta llamada después del método `_positionedStatusBar(context)` .

```
Widget _buildBody (contexto BuildContext) {  
    return GestureDetector( child:  
        Stack( fit:  
            StackFit.expand, children:  
                <Widget>[
```

```
//_transformScaleAndTranslate(),
_transformMatrix4(),
_positionedStatusBar(contexto),
_positionedInkWellAndInkResponse(contexto), ], ), onScaleStart:

_onScaleStart, onScaleUpdate:
_onScaleUpdate, onDoubleTap:
_onDoubleTap, onLongPress:
_onLongPress, );

}
```

2. Cree el método Positioned `_positionedInkWellAndInkResponse(BuildContext context)` después del método `_positionedStatusBar(BuildContext context)`. Devuelve un Positioned usando el constructor por defecto . El propósito de este widget posicionado es agregar a la barra de estado de gestos actual los widgets InkWell e InkResponse .
3. Establezca la propiedad superior en 50.0 para colocarla debajo del widget Posicionado anterior (estado de gesto barra de visualización) en el widget Pila .
4. Establezca la propiedad de ancho utilizando el ancho de MediaQuery para expandir el ancho completo del dispositivo. El elemento secundario es un contenedor con la propiedad de color establecida en un tono de Colors.white54.
5. Establezca la propiedad Altura del contenedor en 56,0. Establezca el contenedor secundario en una fila con mainAxisAlignment de MainAxisAlignment.spaceAround.

```
Posicionado _positionedInkWellAndInkResponse (contexto BuildContext) {
return Positioned( top:
50.0, width:
MediaQuery.of(context).size.width, child: Container( color:
Colors.white54, height:
56.0, child: Row( mainAxisAlignment:
MainAxisAlignment.spaceAround, children: <Widget>[

], ), ), );
}
```

6. Agregue a la lista de filas secundarias de Widget un InkWell y InkResponse. Dado que ambos widgets tienen las mismas propiedades, siga las mismas instrucciones para InkWell e InkResponse.
7. Establezca la propiedad secundaria en un contenedor con la propiedad de altura establecida en 48,0, la propiedad de ancho establecida en 128,0 y la propiedad de color establecida en un tono claro de Colors.black12. Establezca la propiedad secundaria en Icons.touch_app con la propiedad de tamaño de 32,0.
8. Para personalizar el color de la salpicadura, establezca la propiedad splashColor en Colors.lightGreenAccent y la propiedad HighlightColor a Colors.lightBlueAccent. El splashColor se muestra donde el puntero tocó la pantalla por primera vez, y el highlightColor es el efecto de salpicadura (ondulación).
9. Agregue a InkWell e InkResponse onTap , onDoubleTap y onLongPress para escuchar cada gesto. Pase respectivamente los métodos _setScaleSmall, _setScaleBig y _onLongPress .

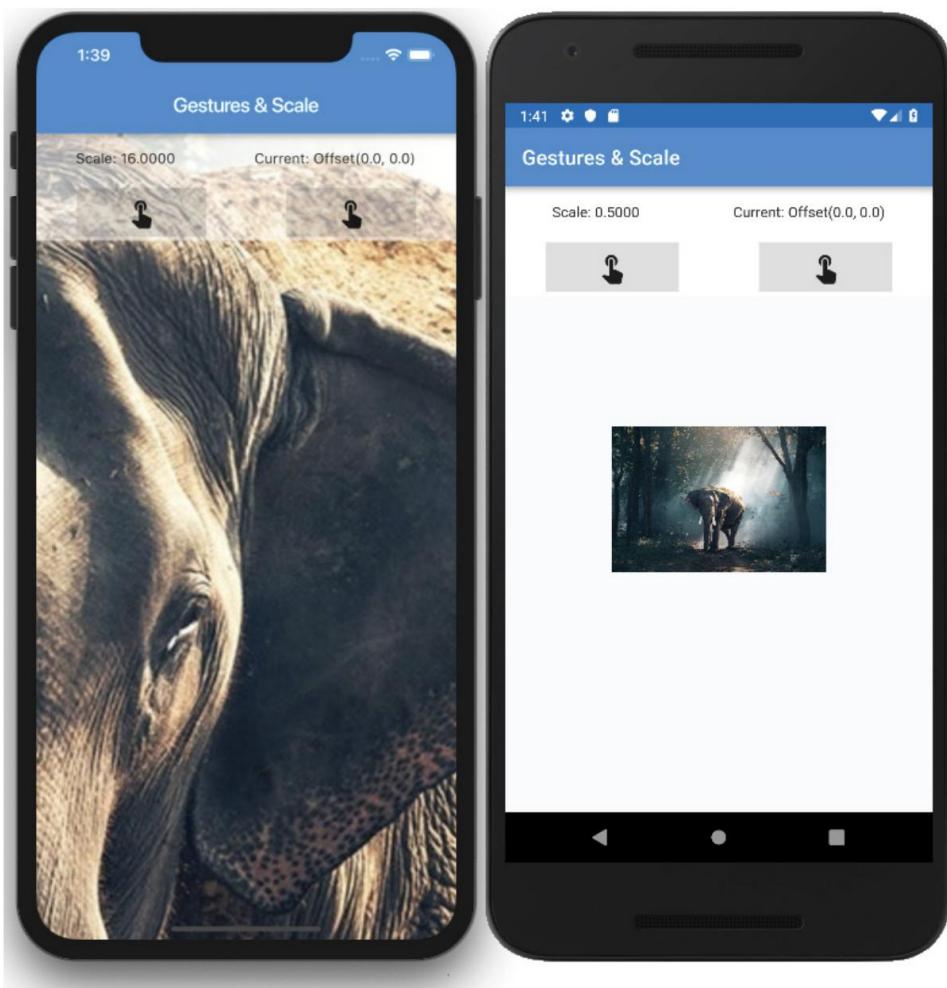
```
Posicionado _positionedInkWellAndInkResponse (contexto BuildContext) {  
    return Positioned( top:  
        50.0, width:  
        MediaQuery.of(context).size.width, child:  
        Container( color:  
            Colors.white54, height: 56.0,  
            child:  
            Row( mainAxisAlignment: MainAxisAlignment.spaceAround,  
                children:  
  
                <Widget>[ InkWell( child:  
                    Container( height:  
                        48.0, width:  
                        128.0, color: Colors.black12,  
                        child:  
                        Icon( Icons.touch_app,  
                            size: 32.0, ), ),  
  
                        splashColor: Colors.lightGreenAccent,  
                        HighlightColor: Colors.lightBlueAccent, onTap:  
                        _setScaleSmall,  
                        onDoubleTap: _setScaleBig,  
                        onLongPress: _onLongPress, ),  
  
                        InkResponse( hijo:  
                        Contenedor(
```

```
alto: 48.0, ancho:  
128.0, color:  
Colors.black12, child:  
  
Icon( Icons.touch_app,  
size: 32.0, ), ),  
  
splashColor: Colors.lightGreenAccent,  
HighlightColor: Colors.lightBlueAccent, onTap:  
_setScaleSmall,  
onDoubleTap: _setScaleBig,  
onLongPress:  
  
_onLongPress, ), ], ), ), );  
}
```

10. Cree los métodos `_setScaleSmall()` y `_setScaleBig()` después del método `_resetToDefaultValues()`. Para el método `_setScaleSmall()`, agregue un `setState()` para modificar la variable `_currentScale` a 0.5. Cuando se captura el gesto `onTap`, se reducirá el tamaño de la imagen a la mitad del tamaño original.

Para el método `_setScaleBig()`, agregue un `setState()` para modificar la variable `_currentScale` a 16.0. Cuando se captura el gesto `onDoubleTap`, aumentará el tamaño de la imagen a 16 veces el tamaño original.

```
void _setScaleSmall()  
{ setState()  
{ _currentScale = 0.5; };  
  
}  
  
void _setScaleBig()  
{ setState()  
{ _currentScale = 16.0; };  
  
}
```



CÓMO FUNCIONA

Tanto los widgets InkWell como InkResponse escuchan las mismas devoluciones de llamada de gestos. Los widgets capturan los gestos onTap, onDoubleTap y onLongPress (propiedades).

Cuando se captura un solo toque, onTap llama al método `_setScaleSmall()` para escalar la imagen a la mitad del tamaño original.

Cuando se captura un doble toque, onDoubleTap llama al método _setScaleBig() para escalar la imagen a 16 veces el tamaño original.

Cuando se captura una pulsación prolongada, onLongPress llama al método _onLongPress() para restablecer todos los valores a las posiciones y tamaños originales.

Los principales beneficios de usar InkWell e InkResponse son capturar toques en la pantalla y tener un hermoso toque. Este tipo de reacción genera una buena UX, ya que correlaciona una animación con la acción de un usuario.

USO DEL WIDGET DESECHABLE

El widget Descartable se descarta con un gesto de arrastre. La dirección del arrastre se puede cambiar usando DismissDirection para la propiedad de dirección . (Consulte la Tabla 11.2 para ver las opciones de DismissDirection). El widget secundario Descartable se desliza fuera de la vista y anima automáticamente la altura o el ancho (dependiendo de la dirección de descarte) hasta cero. Esta animación ocurre en dos pasos; primero, el elemento secundario Descartable se desliza fuera de la vista y, en segundo lugar, el tamaño se reduce a cero. Una vez que se descarta Dismissible , puede usar la devolución de llamada onDismissed para realizar las acciones necesarias, como eliminar un registro de datos de la base de datos o marcar una tarea pendiente como completa (Figura 11.6).

Si no maneja la devolución de llamada onDismissed , recibirá el error "Un widget descartado descartado sigue siendo parte del árbol". Por ejemplo, si usa una Lista de elementos, una vez que se elimine el Desechable , debe eliminar el elemento de la Lista implementando la devolución de llamada onDismissed .

Echará un vistazo detallado a cómo manejar esto en el paso 9 del siguiente ejercicio.

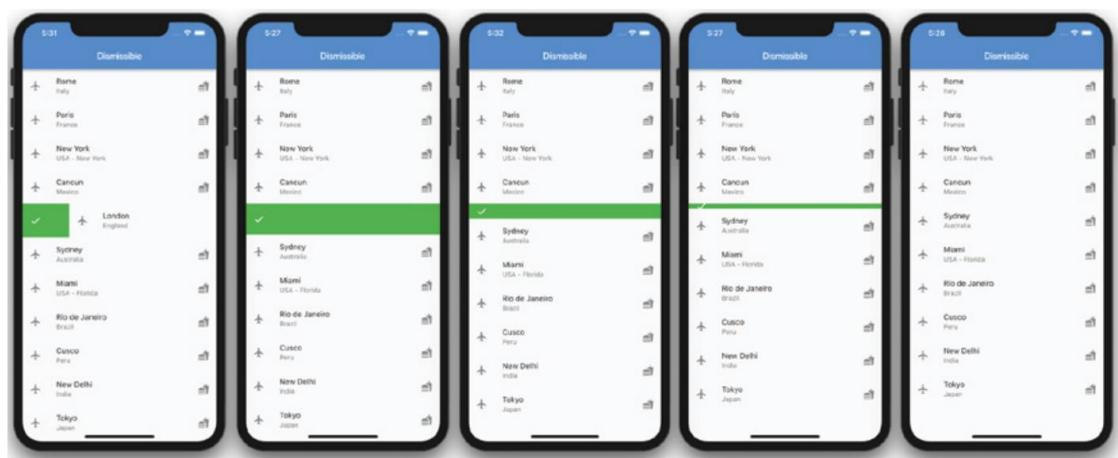


FIGURA 11.6: Widget desecharable que muestra la animación desecharada de la fila deslizada para completar el elemento

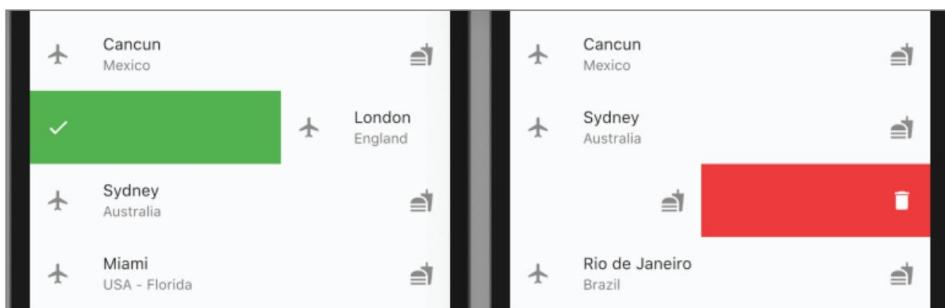
TABLA 11.2: Opciones de descartar DismissDirection

DIRECCIÓN	DESPEDIDO CUANDO . . .
de inicio a fin	Arrastrando de izquierda a derecha.*
fin a inicio	Arrastrando de derecha a izquierda.*
horizontal	Arrastrando hacia la izquierda o hacia la derecha.
arriba	Arrastrando hacia arriba.
abajo	Arrastrando.
vertical	Arrastrando hacia arriba o hacia abajo.

* Suponiendo que la dirección de lectura es de izquierda a derecha; cuando la dirección de lectura es de derecha a izquierda, funcionan de manera opuesta.

PRUÉBALO Creación de la aplicación descartable

En este ejemplo, creará una lista de viajes de vacaciones y, cuando arraste de izquierda a derecha, Descartar muestra un ícono de casilla de verificación con un fondo verde para marcar el viaje como completado. Al deslizar el dedo de derecha a izquierda, Descartable muestra un ícono de eliminación con fondo rojo para eliminar el viaje.



El Dismissible maneja todas las animaciones, como deslizar, cambiar el tamaño para eliminar la fila seleccionada y deslizar el siguiente elemento de la lista hacia arriba.

Creará una clase de viaje para guardar los detalles de las vacaciones con las variables id, tripName y tripLocation . Carga una lista para realizar un seguimiento de las vacaciones agregando detalles de viaje individuales . La propiedad del cuerpo usa un ListView.builder que devuelve un widget descartable . Establece la propiedad secundaria Descartable en ListTile, y las propiedades de fondo y fondo secundario devuelven un Contenedor con el elemento secundario como Fila con una lista de elementos secundarios de Widget con un ícono.

Tenga en cuenta que en este ejemplo, para mantener el árbol de widgets poco profundo, utilizará métodos en lugar de clases de widgets.

1. Cree un nuevo proyecto de Flutter y asigne el nombre ch11_dismissible. Nuevamente, puede seguir las instrucciones del Capítulo 4. Para este proyecto, solo necesita crear las carpetas de páginas y clases . Cree Home Class como StatefulWidget ya que los datos (estado) requieren cambios.

2. Abra el archivo home.dart y agregue al cuerpo un ListView.builder().

cuerpo: ListView.builder(),

3. Agregue en la parte superior del archivo el paquete import trip.dart que creará a continuación.

```
importar 'paquete: flutter/material.dart'; importar  
'paquete: ch11_dismissible/classes/trip.dart';
```

4. Cree un nuevo archivo Dart en la carpeta de clases . Haga clic derecho en la carpeta de clases , seleccione Nuevo Dart Archivo, ingrese trip.dart y haga clic en el botón Aceptar para guardar.

5. Cree la clase de viaje. La clase de viaje contiene los detalles de las vacaciones con una identificación, un nombre de viaje y tripLocation Variables de cadena. Cree el constructor de viaje con parámetros con nombre ingresando los nombres de variable this.id, this.tripName y this.tripLocation dentro de las llaves {}.

```
excursión {  
    ID de cadena;  
    String viajeNombre;  
    String ubicación de viaje;  
  
    Viaje({this.id, this.tripName, this.tripLocation}); }
```

6. Edite el archivo home.dart y después de que la clase _HomeState extienda State<Home> y antes de @override, agregue la variable List _trips inicializada por una lista de viajes vacía.

```
Lista _viajes = Lista<Viaje>();
```

7. Anule initState() para inicializar la lista _trips. Vas a agregar 11 elementos a la lista de _trips. Por lo general, estos datos se leen desde una base de datos local o un servidor web.

```
@anular  
void initState() {  
    super.initState();  
    _viajes..add(Viaje(id: '0', nombre del viaje: 'Roma', ubicación del viaje: 'Italia')) ..add(Viaje(id:  
        '1', nombre del viaje: 'París', ubicación del viaje: 'Francia') ) ..add(Viaje(id: '2', nombre  
        del viaje: 'Nueva York', ubicación del viaje: 'EE.UU. - Nueva York')) ..add(Viaje(id: '3', nombre del viaje:  
        'Cancún', ubicación del viaje: 'México')) ..add(Viaje(id: '4', nombre del viaje: 'Londres',  
        ubicación del viaje: 'Inglaterra')) ..add(Viaje(id: '5', nombre del viaje: 'Sidney', ubicación del  
        viaje: 'Australia')) ..add(Trip(id: '6', tripName: 'Miami', tripLocation: 'USA - Florida')) ..add(Trip(id:  
        '7', tripName: 'Río de Janeiro ', TripLocation: 'Brasil')) ..add(Trip(id: '8', tripName: 'Cusco',  
        tripLocation: 'Peru')) ..add(Trip(id: '9', tripName: 'New Delhi', tripLocation: 'India')) ..add(Trip(id: '10',  
        tripName: 'Tokyo', tripLocation: 'Japan'));  
}
```

8. Cree dos métodos que simulen marcar un elemento de viaje como completado o eliminado en la base de datos.

Cree los métodos `_markTripCompleted()` y `_deleteTrip()` que actúan como marcadores de posición para escribir en una base de datos.

```
void _marcarViajeCompleto() {  
    // Marcar viaje completado en base de datos o servicio web  
}  
  
void _borrarViaje() {  
    // Eliminar viaje de la base de datos o servicio web  
}
```

9. Establezca el constructor `ListView.builder` con el argumento `itemCount` establecido en `_trips.length`, que es el número de filas en la lista `_trips`. Para el argumento `itemBuilder`, toma el `BuildContext` y el índice del widget como un valor `int`.

`itemCount: _trips.length,`

El `itemBuilder` devuelve un `Dismissible` con la propiedad clave como `Key(_trips[index].id)`.

La clave es el identificador de cada widget y debe ser único, por lo que utiliza el elemento de identificación `_trips`. La propiedad secundaria se establece en el método `_buildListTile(index)`, que pasa el índice del widget actual.

`clave: Clave(_viajes[índice].id),`

10. `Dismissible` tiene propiedades de fondo (arrastrar de izquierda a derecha) y de fondo secundario (arrastrar de izquierda a derecha). Establezca la propiedad de fondo en el método `_buildCompleteTrip()` y configure el secundario en el método `_buildRemoveTrip()`. Tenga en cuenta que `Dismissible` tiene una propiedad de dirección opcional que puede establecer las restricciones sobre qué dirección usar.

```
hijo: _buildListTile(index), fondo:  
    _buildCompleteTrip(), secundarioBackground:  
    _buildRemoveTrip(),
```

La devolución de llamada `onDismissed` (propiedad) se llama cuando se descarta el widget, lo que proporciona una función para ejecutar el código eliminando el elemento del widget descartado de la lista `_trips`. En un escenario del mundo real, también actualizaría la base de datos.

Es importante que una vez que se descarte el artículo, se elimine de la lista de `_trips` o se eliminará. causar un error.

```
// Un widget descartable sigue siendo parte del árbol.  
// Asegúrese de implementar el controlador onDismissed y de eliminar inmediatamente el  
descartable  
// widget de la aplicación una vez que el controlador se haya activado.
```

Esto tiene sentido ya que el artículo ha sido descartado y eliminado. Todo esto es posible mediante el uso de la propiedad de clave única .

`onDismissed` pasa `DismissDirection` donde verifica con un operador ternario si la dirección es `startToEnd` y llama al método `_markTripCompleted()` o llama al método

método `_deleteTrip()`. El siguiente paso es usar `setState` para eliminar el artículo descartado de la lista `_trips` usando `_trips.removeAt(index)`.

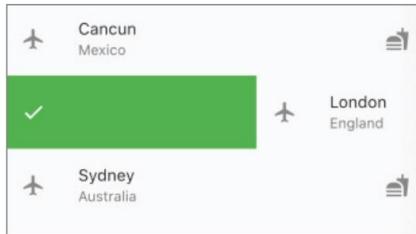
```
cuerpo: ListView.builder(  
    itemCount: _trips.length,  
    itemBuilder: (contexto BuildContext, int index) { return Desechable(  
  
        clave: Clave(_viajes[indice].id), hijo:  
        _buildListTile(indice), fondo:  
        _buildCompleteTrip(), secundario:  
        _buildRemoveTrip(), onDismissed: (dirección  
DismissDirection) {  
  
    dirección == DismissDirection.startToEnd ? _markTripCompleted() : _deleteTrip();  
    // Eliminar elemento de la lista  
    setState(()  
        { _trips.removeAt(index);});}, );  
  
}, ),
```

11. Agregue el método Widget `_buildListTile(int index)` después de Widget `build(BuildContext context) {...}`. Devuelve un `ListTile` y establece las propiedades de título, subtítulo, inicial y final .
12. Configure la propiedad del título como un widget de texto que muestre el nombre del viaje y configure el subtítulo en `TripLocation`. Establezca las propiedades iniciales y finales como íconos.

```
ListTile _buildListTile(int index) { return ListTile( title:  
    Text('${  
        _trips[index].tripName}'), subtítulo: Text(_trips[index].tripLocation),  
    inicial: Icon(Icons.flight), final: Icon(Icons.comida rápida), );  
  
}
```

13. Agregue el método Widget `_buildCompleteTrip()` para devolver un Contenedor con el color verde y la propiedad secundaria como Relleno. El elemento secundario de relleno es una fila con la alineación configurada para comenzar (en el lado izquierdo para los idiomas de izquierda a derecha) con una lista de elementos secundarios de Widget de un ícono.

La propiedad de fondo se revela cuando el usuario arrastra el elemento y es importante transmitir qué acción se llevará a cabo. En este caso, usted está completando un viaje y muestra un ícono hecho (casilla de verificación) con un fondo verde que convoca a la acción.



```
Contenedor _buildCompleteTrip() {  
    return Container( color:  
        Colors.green, child:  
        Padding( padding:  
            const EdgeInsets.all(16.0), child: Row(  
  
                mainAxisAlignment: MainAxisAlignment.start, children:  
                <Widget>[ Icon( Icons.done,  
                    color:  
  
                        Colors.white, ), ], ), );  
  
    }  
}
```

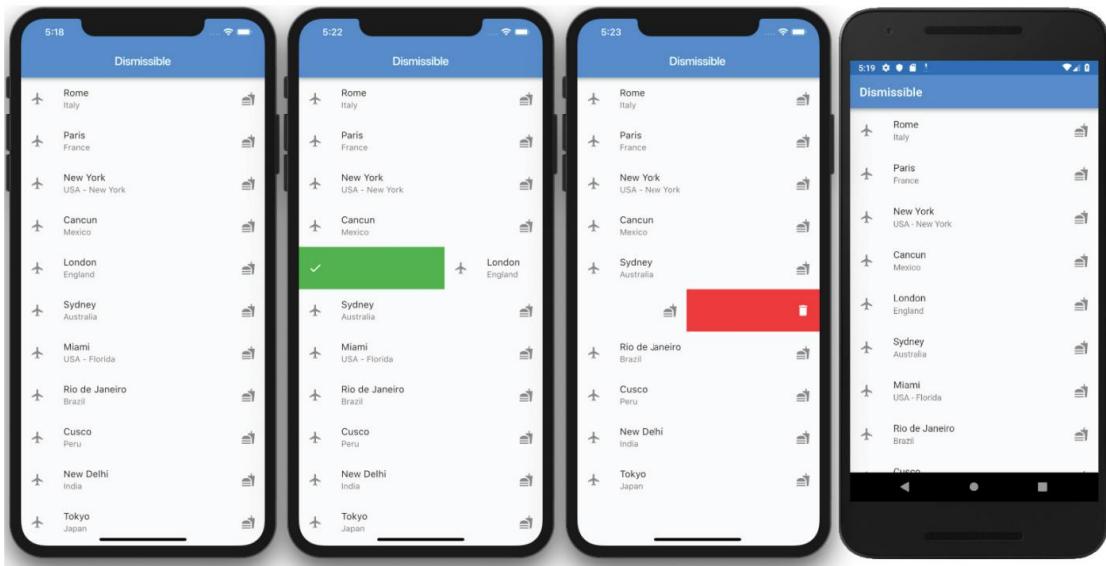
14. Agregue el método de widget _buildRemoveTrip() para devolver un Contenedor con el color rojo y la propiedad secundaria como Relleno. El elemento secundario de relleno es una fila con la alineación configurada para finalizar (en el lado derecho para los idiomas de izquierda a derecha) con una lista de elementos secundarios de Widget de un ícono.

La propiedad secondBackground se revela cuando el usuario arrastra el elemento y es importante transmitir qué acción se llevará a cabo. En este caso, estás eliminando un viaje, y te muestra un ícono de eliminar (papelera) con fondo rojo convocando a la acción.



```
Contenedor _buildRemoveTrip() { return  
    Container( color:  
        Colors.red,
```

```
child: Padding( padding:  
    const EdgeInsets.all(16.0), child: Row( mainAxisAlignment:  
        MainAxisAlignment.end, children: <Widget>[ Icon( Icons.delete,  
            color: Colors.white, ), ], ), );  
  
}  
}
```



CÓMO FUNCIONA

Usó un ListView para crear una lista de detalles del viaje . ListView itemBuilder devuelve un Dismissible con una propiedad de clave establecida mediante el uso de la clase Key como un identificador único para cada widget. Usar la propiedad clave es extremadamente importante porque Dismissible la usa cuando reemplaza un widget con otro en el árbol de widgets.

Para transmitir la acción adecuada cuando el usuario está arrastrando (de izquierda a derecha), personalizó la propiedad de fondo para mostrar un fondo verde con un ícono de hecho. Cuando el usuario está arrastrando (de derecha a izquierda), usted personalizó la propiedad secondBackground para mostrar un fondo rojo con un ícono de eliminación.

Usó la devolución de llamada `onDismissed` (propiedad) para verificar `DismissDirection` y tomar las medidas apropiadas. Al utilizar el operador ternario, verificó si la dirección era de inicio a fin y llamó al método `_markTripCompleted()` ; de lo contrario, llamaste al método `_deleteTrip()` .

A continuación, usó `setState` para eliminar el elemento actual de la lista `_trips`.

RESUMEN

En este capítulo, aprendió a usar `GestureDetector` para manejar los gestos `onTap`, `onDoubleTap`, `onLongPress` y `onPanUpdate` . `onPanUpdate` es adecuado para usar cuando necesita rastrear el arrastre en cualquier dirección . Echó un vistazo en profundidad al uso de `GestureDetector` para moverse, escalar acercando/alejando, toque dos veces para aumentar el zoom y mantenga presionado para restablecer la imagen del elefante al tamaño original. Por ejemplo, estas técnicas se aplicarían a una aplicación de diario cuando un usuario selecciona una imagen y quiere ver más de cerca los detalles. Para lograr este objetivo, utilizó `onScaleStart` y `onScaleUpdate` para escalar la imagen. Use `onDoubleTap` para aumentar el zoom y `onLongPress` para restablecer el zoom al tamaño predeterminado original.

Aprendiste dos técnicas diferentes para escalar y mover la imagen cuando se detecta un gesto. Con la primera técnica, usó el widget `Transform` al anidar el constructor `Transform.scale` para cambiar el tamaño de la imagen y el constructor `Transform.translate` para mover la imagen. Para la segunda técnica, usó el constructor predeterminado `Transform` usando `Matrix4` para aplicar las transformaciones. Al usar `Matrix4`, ejecutó múltiples transformaciones en cascada (... escalar (...) traducir ()) sin la necesidad de anidar múltiples widgets de transformación .

Usó `InkWell` e `InkResponse` para responder a gestos táctiles como tocar, tocar dos veces y mantener presionado. Ambos widgets son componentes de materiales (widgets de diseño de materiales Flutter) que muestran un efecto de salpicadura cuando se tocan.

Implementó la característica de arrastrar y soltar mediante los widgets `Draggable` y `DragTarget` . Estos widgets se usan en conjunto. El widget `Draggable` tiene una propiedad de datos que pasa información al widget `DragTarget` . El widget `DragTarget` puede aceptar o rechazar los datos, lo que le permite verificar el formato de datos correcto. En este ejemplo, arrastró el ícono de la paleta de pintura (`Arrastrable`) sobre el widget de Texto (`DragTarget`) , y una vez que lo soltó, el color del Texto cambia a rojo.

El widget `Draggable` escucha los gestos de arrastre verticales y horizontales. Al usar `DismissDirection` para la propiedad de dirección, puede limitar los gestos de arrastre que escucha, como la restricción a gestos solo horizontales. En este ejemplo, creó una lista de elementos de viaje que se muestran con `ListView.builder`. Cuando el usuario arrastra un elemento de la lista de izquierda a derecha, se revela un fondo verde con un ícono de casilla de verificación para transmitir la acción que está a punto de realizar, completando el viaje. Pero si el usuario arrastra el elemento de la lista de derecha a izquierda, se revela un fondo rojo con un ícono de papelera para transmitir la acción de que se va a eliminar el viaje. Cómo

el Desechable sabe qué elemento eliminar? Al usar la propiedad de clave Dismissible , pasó un identificador único para cada elemento de la lista y, una vez que se llamó a la propiedad onDismissed (devolución de llamada), verificó la dirección del arrastre y tomó la acción adecuada. Luego usó setState para asegurarse de que el elemento descartado se elimine de la lista _trips . Es importante manejar la devolución de llamada onDismissed o recibirá el error "Un widget descartable descartado todavía es parte del árbol".

En el próximo capítulo, aprenderá a escribir código específico de la plataforma iOS y Android. Usarás Swift para iOS y Kotlin para Android. Estos canales de plataforma le brindan la posibilidad de utilizar funciones nativas, como acceder a la ubicación GPS del dispositivo, notificaciones locales, sistema de archivos local, uso compartido y muchas más.

LO QUE APRENDISTE EN ESTE CAPÍTULO

TEMA	CONCEPTOS CLAVE
Implementando GestureDetector	Este widget reconoce gestos como tocar, tocar dos veces, presionar prolongadamente, desplazarse, arrastrar verticalmente, arrastrar horizontalmente y escalar.
Implementando Arrastrable	Este widget se arrastra a un DragTarget.
Implementando DragTarget	Este widget recibe datos de un Draggable.
Implementación de InkWell y Respuesta de tinta	El InkWell es un área rectangular que responde al tacto y recorta las salpicaduras dentro de su área. InkResponse responde al tacto y las salpicaduras se expanden fuera de su área .
Implementación descartable	Este widget se descarta arrastrándolo.

12

Escritura de código nativo de la plataforma

LO QUE APRENDERÁS EN ESTE CAPÍTULO

Cómo usar los canales de la plataforma para enviar y recibir mensajes desde la aplicación Flutter a iOS y Android para acceder a funciones API específicas

Cómo escribir código de plataforma nativa en iOS Swift y Android Kotlin para acceder al dispositivo información

Cómo usar MethodChannel para enviar mensajes desde la aplicación Flutter (en la lado del cliente)

Cómo usar FlutterMethodChannel en iOS y MethodChannel en Android para recibir llamadas y enviar resultados (en el lado del host)

Los canales de la plataforma le brindan la capacidad de usar funciones nativas, como acceder a la información del dispositivo, la ubicación del GPS, las notificaciones locales, el sistema de archivos local, compartir y muchas más. En la sección "Paquetes externos" del Capítulo 2, "Creación de una aplicación Hello World", aprendió a usar paquetes de terceros para agregar funcionalidad a sus aplicaciones. En este capítulo, en lugar de depender de paquetes de terceros, aprenderá a agregar funciones personalizadas a sus aplicaciones mediante el uso de canales de plataforma y escribiendo el código API usted mismo. Creará una aplicación que solicite a las plataformas iOS y Android que devuelvan la información del dispositivo.

ENTENDER LOS CANALES DE LA PLATAFORMA

Cuando necesita acceder a las API específicas de la plataforma para iOS y Android, utiliza los canales de la plataforma para enviar y recibir mensajes. La aplicación Flutter es el cliente y el código nativo de la plataforma para iOS y Android es el host. Si es necesario, también es posible tener el código nativo de la plataforma para que actúe como un cliente para llamar a los métodos escritos en el código dart de la aplicación Flutter.

Los mensajes entre el cliente y el host son asíncronos, lo que garantiza que la interfaz de usuario siga respondiendo y no se bloquee. En el Capítulo 3, "Aprendizaje de los conceptos básicos de Dart", aprendió que las funciones asíncronas realizan operaciones que consumen mucho tiempo sin esperar a que se completen.

Para el lado del cliente (aplicación Flutter), utiliza MethodChannel desde un método asíncrono para enviar mensajes que contienen la llamada al método que ejecutará el lado del host (iOS y Android). Una vez que el host devuelve la respuesta, puede actualizar la interfaz de usuario para mostrar la información recibida.

Para el lado del host, usa FlutterMethodChannel en iOS y MethodChannel en Android.

Una vez que el host recibe la llamada del cliente, el código de la plataforma nativa ejecuta el método llamado y luego devuelve el resultado (Figura 12.1).

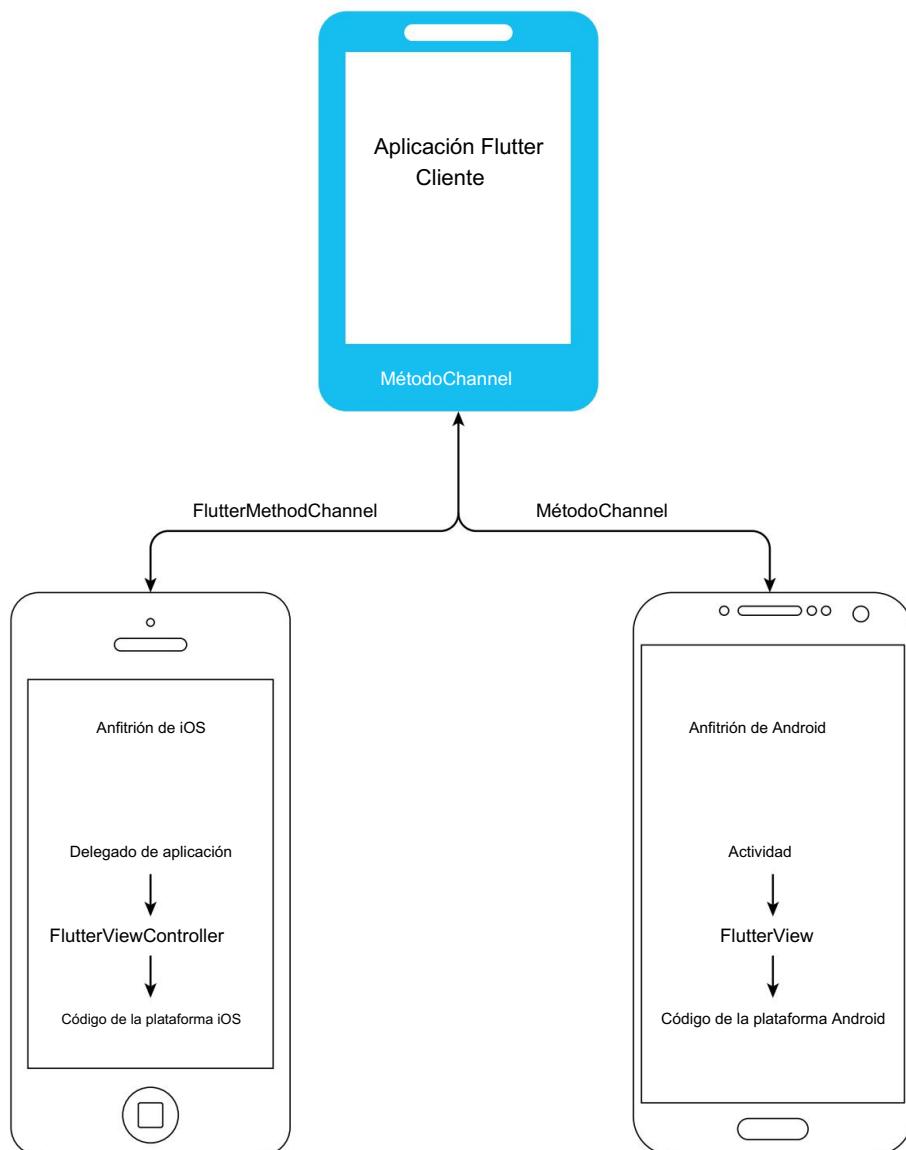


FIGURA 12.1: Mensajes del canal de la plataforma

IMPLEMENTACIÓN DE LA APLICACIÓN DEL CANAL DE LA PLATAFORMA DEL CLIENTE

Para iniciar la comunicación desde la aplicación cliente de Flutter a las plataformas iOS y Android, utilice MethodChannel . Un MethodChannel usa llamadas a métodos asíncronos y el canal requiere un nombre único. El nombre del canal debe ser el mismo para el cliente que para el host de iOS y Android. Le sugiero que cuando esté creando un nombre único para el canal, use el nombre de la aplicación, un prefijo de dominio y un nombre descriptivo para la tarea, como canalplataforma.nombrededeempresa.com/información del dispositivo.

```
// Plantilla de nombre
appname.domain.com/taskname //
Nombre de canal
canalplataforma.companyname.com/deviceinfo
```

Al principio, parece que te estás pasando de la raya al nombrar el canal, entonces, ¿por qué es importante que el nombre sea único? Si tiene varios canales con nombre y comparten el mismo nombre, provocarán conflictos con los mensajes de los demás.

Para implementar un canal, crea MethodChannel a través del constructor predeterminado pasando el nombre de canal único. El constructor predeterminado toma dos argumentos: el primero es el nombre del canal y el segundo (que es opcional) declara el MethodCodec predeterminado. El MethodCodec es el StandardMethodCodec , que utiliza la codificación binaria estándar de Flutter; esto significa que la serialización de los datos enviados entre el cliente y el host se maneja automáticamente. Dado que conoce el nombre del canal en el momento de la compilación y no cambiará, cree MethodChannel en una variable const estática . Asegúrese de usar la palabra clave constante , o recibirá el error "Solo los campos estáticos pueden declararse como constantes".

```
static const plataforma = const
MethodChannel('platformchannel.companyname.com/deviceinfo');
```

La Tabla 12.1 muestra los tipos de valores admitidos para Dart, iOS y Android.

TABLA 12.1: Tipos de valores admitidos por StandardMessageCodec

DARDO	iOS	ANDROIDE
nulo	nulo	nulo
bool	NSNumber númeroConBool:	java.lang.booleano
En t	NSNumber númeroConInt:	java.lang.Integer
int (mayor de 32 bits)	NSNumber númeroConLargo:	java.lang.Long
doble	NSNumber númeroConDoble:	java.lang.Double
Cadena	NSCadena	java.lang.String
Uint8List	FlutterStandardTypedData typedDataWithBytes:	byte[]

continúa

TABLA 12.1: (continuación)

DARDO	iOS	ANDROIDE
Int32Lista	FlutterStandardTypedData typedDataWithInt32:	En t[]
Int64Lista	FlutterStandardTypedData typedDataWithInt64:	largo[]
Lista flotante64	FlutterStandardTypedData typedDataWithFloat64:	doble[]
Lista	NSArreglo	Java. util.ArrayList
Mapa	NSDiccionario	java.util.HashMap

Para llamar y especificar qué método ejecutar en el host de iOS y Android, utilice el constructor del método de invocación para pasar el nombre del método como una cadena. El método de llamada se llama desde dentro de un método Future ya que la llamada es asíncrona.

```
String deviceInfo = espera plataform.invokeMethod('getDeviceInfo');
```

Una vez que se implementan el cliente y los canales de la plataforma iOS y Android, el lado del cliente de Flutter de la aplicación mostrará la información del dispositivo correspondiente según el dispositivo (Figura 12.2).

PRUÉBELO Creación de la aplicación de canal de plataforma de cliente

En este ejemplo, desea mostrar la información del dispositivo en ejecución, como el fabricante, el modelo del dispositivo, el nombre, el sistema operativo y algunos otros detalles. El cliente de la aplicación Flutter está escrito en Dart (como de costumbre) e implementa MethodChannel para iniciar una llamada al host de iOS y Android.

El host de iOS está escrito en Swift para acceder a la llamada API de la plataforma al UIDevice para consultar la información del dispositivo y usa FlutterMethodChannel para recibir y devolver la información solicitada.

El host de Android está escrito en Kotlin para acceder a la llamada API de la plataforma a Build para consultar la información del dispositivo y utiliza MethodChannel para recibir y devolver la información solicitada.

Esta aplicación se divide en tres ejercicios diferentes de "Pruébalo". En este primero, se concentrará en crear la solicitud del lado del cliente. En el segundo, "Creación del canal de la plataforma de host de iOS", creará el canal de la plataforma de host de iOS y, en el tercer ejercicio, "Creación del canal de la plataforma de host de Android", creará el canal de la plataforma de host de Android. Si ejecuta la aplicación después de completar esta primera sección, recibirá el error "Error al obtener la información del dispositivo" ya que aún no ha escrito el código de host de iOS y Android. Tenga en cuenta que los proyectos de iOS y Android son independientes entre sí, y puede apuntar a ambos o solo a uno, y recibirá el error en la plataforma que ejecuta.

1. Cree un nuevo proyecto de Flutter y asígnele el nombre ch12_platform_channel; como siempre, puede seguir las instrucciones del Capítulo 4, "Creación de una plantilla de proyecto inicial". Para este proyecto, solo necesita crear la carpeta de páginas .

2. Abra el archivo home.dart y agregue al cuerpo un SafeArea con el niño como ListTile.

cuerpo: SafeArea(hijo:
ListTile(),),

3. Después de que la clase _HomeState extienda State<Home> y antes de @override, agregue el static const variable _methodChannel inicializado por MethodChannel con el nombre canalplataforma.nombreempresa.com/deviceinfo.

static const _methodChannel = const MethodChannel('platformchannel.companyname. com/deviceinfo');

4. Declare la variable de cadena _deviceInfo que recibirá la información del dispositivo de la llamada de host de iOS y Android.

// Obtener información del dispositivo
Cadena _info del dispositivo = ";

5. Cree la llamada asíncrona _getDeviceInfo() que usa _methodChannel.invokeMethod() para iniciar la llamada al host de iOS y Android. Este método se declara como Future<void> y se marca como asíncrono.

Future<void> _getDeviceInfo() asíncrono { }

6. Cree la variable DeviceInfo String local que recibe la información del dispositivo.

Cadena de información del dispositivo;

Es una buena práctica usar el manejo de excepciones try-catch al llamar al métodoChannel .invokeMethod('getDeviceInfo') en caso de que la llamada falle. El método de llamada toma el nombre del método getDeviceInfo que debe ser el mismo que declaras en el código de host de iOS y Android.

pruebe
{ deviceInfo = await methodChannel.invokeMethod('getDeviceInfo'); } en PlatformException catch
(e) { deviceInfo = "Error al obtener la información del dispositivo: '\${e.message}'.";
}

7. Agregue el método setState() que completa la variable _deviceInfo (disponible en toda la clase) desde el valor de deviceInfo local. En el Capítulo 3, "Aprendizaje de los conceptos básicos de Dart", en la sección "Programación asíncrona", aprendió a usar el objeto Future .

Futuro<vacío> _getDeviceInfo() asíncrono {
Cadena de información del dispositivo;
pruebe { deviceInfo = await _methodChannel.invokeMethod('getDeviceInfo'); } en PlatformException catch (e) { deviceInfo = "Error al obtener la información del dispositivo: '\${e.message}'.";
}

```
setState()  
{ _deviceInfo = deviceInfo; };
```

8. Anule initState() para llamar al método _getDeviceInfo(). Una vez que se inicia la aplicación, hace que la llamada _getDeviceInfo() comience a recuperar información del dispositivo.

```
@anular  
void initState() {  
    super.initState();  
    _getDeviceInfo(); }
```

9. Volvamos a la propiedad del cuerpo y terminemos el widget ListTile para mostrar la información del dispositivo.

Para la propiedad del título , agregue un widget de texto para mostrar el encabezado Información del dispositivo y configure TextStyle en fontSize 24.0 y FontWeight.bold.

10. Para la propiedad de subtítulos , agregue un widget de texto con la variable _deviceInfo que muestra la información real del dispositivo y establezca TextStyle en fontSize 18.0 y FontWeight.bold.

11. Agregue una propiedad contentPadding a ListTile y establezca EdgeInsets.all() en 16.0 para agregar un buen relleno alrededor de la información.

```
cuerpo: SafeArea(  
    hijo: ListTile (título:  
        Texto  
            ('Información del  
            dispositivo:', estilo:  
                Estilo de texto  
                    (tamaño de fuente: 24.0, peso de
```

```
fuente: Peso de  
        fuente.bold,),  
    subtítulo: Texto ( _info  
        de dispositivo,  
        estilo: Estilo de texto (Tamaño de
```

fuente: 18.0, peso de fuente: Peso de fuente.

negrita,),), relleno de contenido: EdgeInsets.all(16.0),),),

CÓMO FUNCIONA

En esta sección de cliente de la aplicación, creó MethodChannel con un nombre único para una variable estática const _methodChannel . El methodChannel se usa para comunicarse entre el cliente y el host mediante la llamada de método asíncrono . Con el método _getDeviceInfo() Future<void> , _methodChannel.invokeMethod('getDeviceInfo') llama al host para ejecutar el método getDeviceInfo en las plataformas iOS y Android. Una vez que se devuelven los datos, el método setState() completa la variable _deviceInfo y el subtítulo ListTile se actualiza con la información del dispositivo.

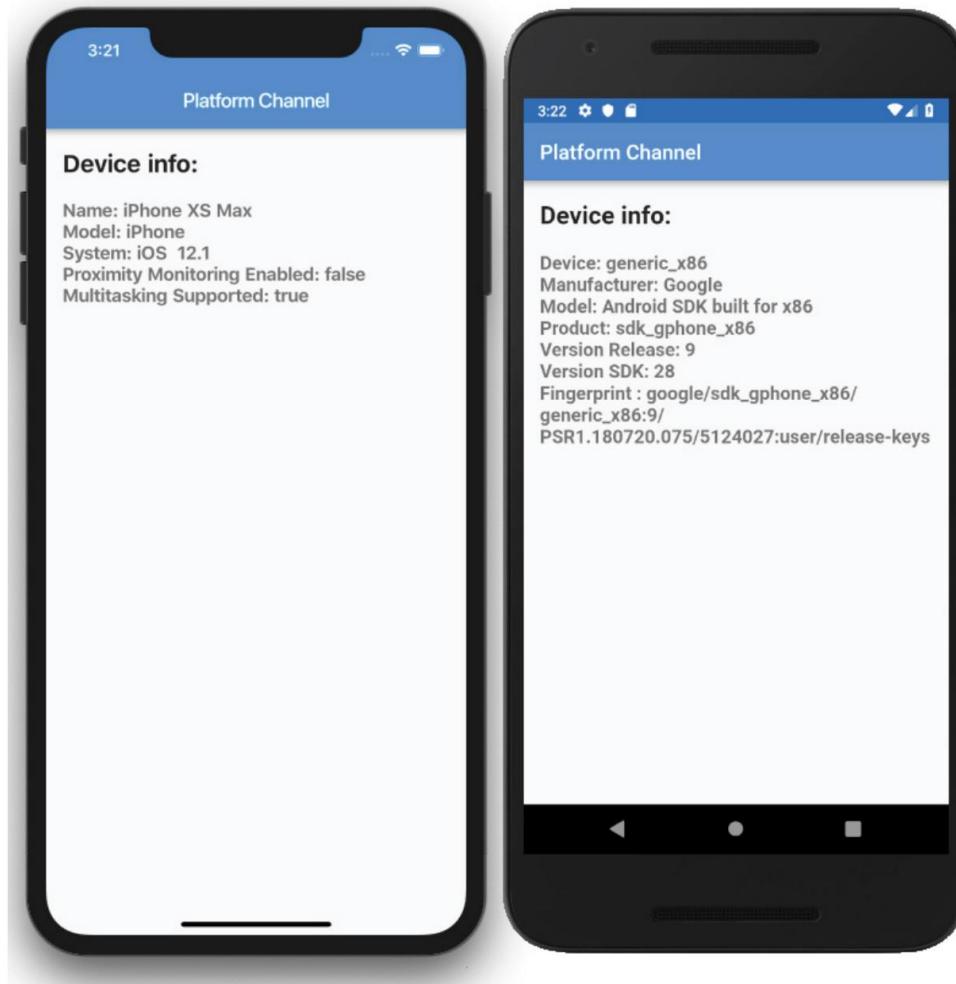


FIGURA 12.2: Información del dispositivo iOS y Android

IMPLEMENTACIÓN DEL CANAL DE LA PLATAFORMA HOST DE iOS

El anfitrión es responsable de escuchar los mensajes entrantes del cliente. Una vez que se recibe un mensaje, el canal busca un nombre de método coincidente, ejecuta el método de llamada y devuelve el resultado apropiado. En iOS, usa FlutterMethodChannel para escuchar los mensajes entrantes que toman dos parámetros. El primer parámetro es el mismo nombre del canal de plataforma: 'platformchannel. companyname.com/deviceinfo'—como el cliente. El segundo es FlutterViewController, que es el rootViewController principal de una aplicación de iOS. rootViewController es el controlador de vista raíz para la ventana de la aplicación iOS que proporciona la vista de contenido de la ventana.

```
let flutterViewController: FlutterViewController = window?.rootViewController as!
FlutterViewController let
deviceInfoChannel = FlutterMethodChannel(nombre: "platformchannel.companyname. com/deviceinfo", binaryMessenger:
controlador)
```

Luego, usa `setMethodCallHandler` (controlador futuro) para configurar una devolución de llamada para un nombre de método coincidente que ejecuta el código de la plataforma nativa de iOS. Una vez completado, devuelve el resultado al cliente.

```
deviceInfoChannel.setMethodCallHandler({ (llamada:
FlutterMethodCall, resultado: FlutterResult) -> Void in // Comprobar el nombre de la
llamada del método entrante y devolver un resultado
})
```

Tanto `FlutterMethodChannel` como `setMethodCallHandler` se colocarán en el método `didFinishLaunchingWithOptions` del archivo `AppDelegate.swift` de la aplicación iOS. `didFinishLaunchingWithOptions` es responsable de notificar al delegado de la aplicación que el proceso de inicio de la aplicación casi ha terminado.

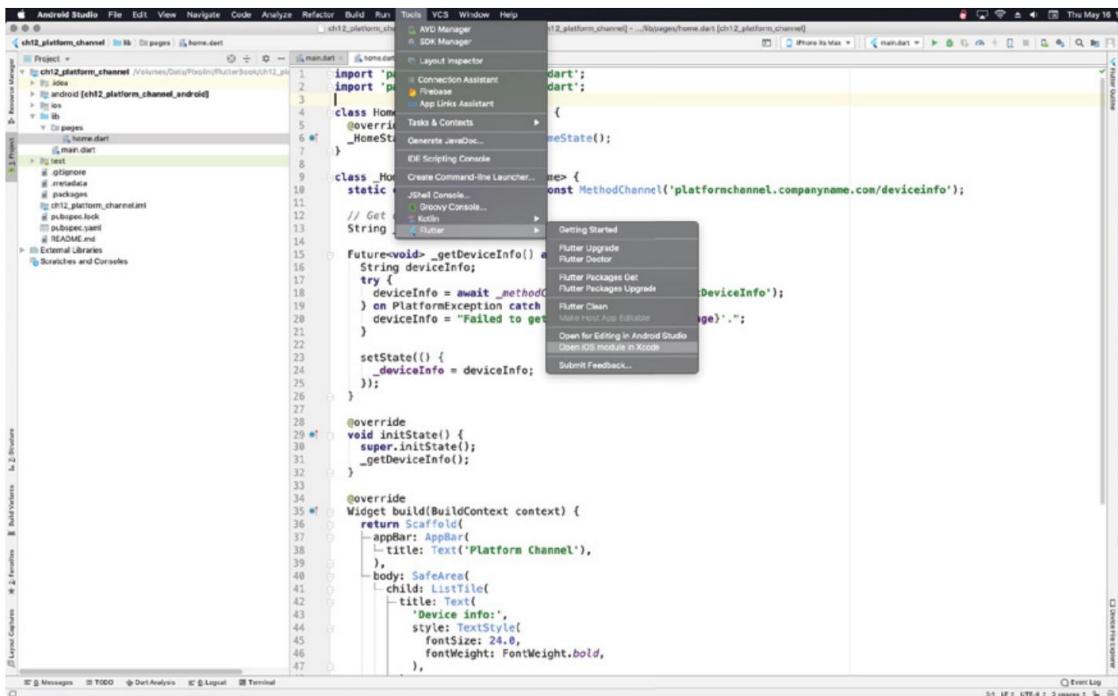
```
anular la aplicación de función (
    _ aplicación: UIApplication,
    didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Cualquiera]?
) -> Bool { //
    Código
}
```

PRUÉBELO Creación del canal de plataforma de host de iOS

En este ejemplo, desea recuperar la información del dispositivo en ejecución, como el fabricante, el modelo del dispositivo, el nombre, el sistema operativo y algunos otros detalles. El host de iOS está escrito en Swift para acceder a la llamada API de la plataforma al objeto `UIDevice` para consultar la información del dispositivo y usa `FlutterMethodChannel` para recibir la comunicación del cliente. Una vez que se recibe el mensaje, `setMethodCallHandler` maneja la solicitud y devuelve el resultado.

En esta sección, abrirá Xcode y editará el código Swift nativo de iOS.

1. Si cerró el proyecto Flutter ch12_platform_channel, vuelva a abrirlo. Haz clic en la barra de menú de Android Studio Tools y selecciona Flutter Abrir módulo iOS en Xcode. Tenga en cuenta que la selección de este elemento de menú abre la aplicación Xcode con el proyecto iOS, pero también puede abrir el proyecto iOS manualmente haciendo doble clic en el archivo `Runner.xcworkspace` ubicado en la carpeta `ios`. Se requiere una computadora Mac con Xcode instalado para editar el proyecto host de iOS.



2. En el área del navegador (lado izquierdo), expanda la carpeta Runner haciendo clic en la flecha y luego seleccione el archivo AppDelegate.swift .

```

import UIKit
import Flutter

@UIApplicationMain
@objc class AppDelegate: FlutterAppDelegate {
    override func application(
        _ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?
    ) -> Bool {
        // Platform Channel
        let flutterViewController = FlutterViewController()
        let deviceInfoChannel = FlutterMethodChannel(name: "platformchannel.companyname.com/deviceinfo", binaryMessenger: flutterViewController)
        deviceInfoChannel.setMethodCallHandler({
            call: FlutterMethodCall, result: FlutterResult -> Void in
            if (call.method == "getDeviceInfo") {
                self.getDeviceInfo(result: result)
            } else {
                result(FlutterMethodNotImplemented)
            }
        })
        GeneratedPluginRegistrant.register(with: self)
        return super.application(application, didFinishLaunchingWithOptions: launchOptions)
    }

    private func getDeviceInfo(result: FlutterResult) {
        let device = UIDevice.current
        var deviceInfo: String = ""
        deviceInfo += "\nName: \(device.name)"
        deviceInfo += "\nSystem Version: \(device.systemVersion)"
        deviceInfo += "\nProximity Monitoring Enabled: \(device.isProximityMonitoringEnabled)"
        deviceInfo += "\nMultitasking supported: \(device.isMultitaskingSupported)"
        result(deviceInfo)
    }
}

```

3. Edite el método didFinishLaunchingWithOptions ; agregará código antes de la llamada de línea GeneratedPluginRegistrant . Declare la variable flutterViewController como FlutterViewController e iníciela con la ventana?.rootViewController as!

FlutterViewController.

```
let flutterViewController: FlutterViewController = ventana?.rootViewController  
¡como! FlutterViewController
```

4. En la línea siguiente, declare la variable deviceInfoChannel iniciándola con el FlutterMethodChannel.

5. Para el primer parámetro de FlutterMethodChannel , nombre, pase el nombre del canal Flutter, el mismo declarado en el cliente. El segundo parámetro, binaryMessenger, toma la variable flutterViewController .

```
let deviceInfoChannel = FlutterMethodChannel(nombre: "platformchannel  
.companyname.com/deviceinfo", binaryMessenger: flutterViewController)
```

6. Agregue deviceInfoChannel.setMethodCallHandler para configurar una devolución de llamada que coincide con el nombre del método entrante de getDeviceInfo mediante una instrucción if-else . Si call.method coincide (==) con getDeviceInfo , entonces llama al método self.getDeviceInfo(result: result) (que crea en el siguiente paso) que recupera la información del dispositivo.

Si call.method no coincide con el nombre del método entrante, la instrucción else devuelve el resultado (FlutterMethodNotImplemented). FlutterMethodNotImplemented es una constante que responde a la llamada de que el método es desconocido o no está implementado .

```
dispositivoInfoChannel.setMethodCallHandler({  
    (llamada: FlutterMethodCall, resultado: FlutterResult) -> Void in if (llamada.método  
    == "getDeviceInfo") { self.getDeviceInfo(resultado:  
    resultado)  
  
    } más  
    { resultado (FlutterMethodNotImplemented)  
    }  
})
```

7. Después del método didFinishLaunchingWithOptions , agregue getDeviceInfo(resultado:

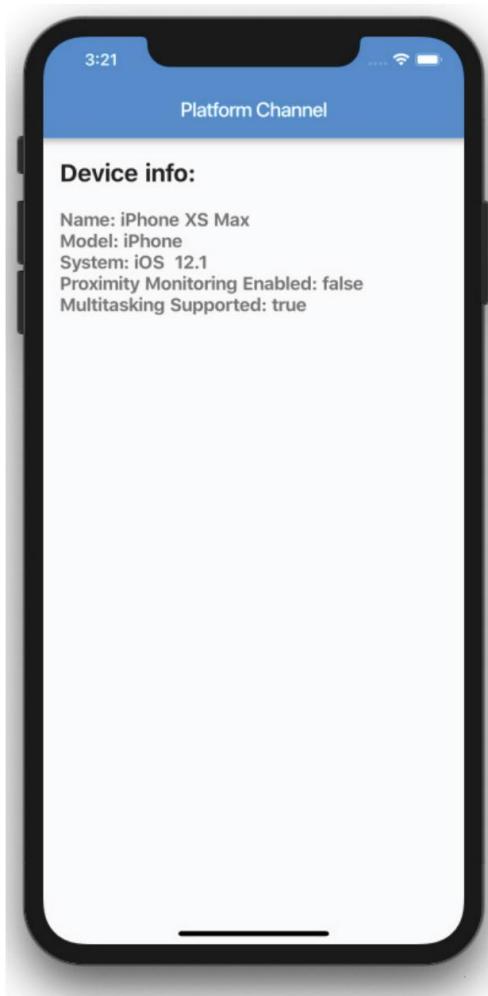
FlutterResult) que utiliza iOS Swift UIDevice.current para consultar y recuperar la información del dispositivo actual.

8. Declare la variable de dispositivo let inicializándola con UIDevice.current. La palabra clave let es similar a la palabra clave final de Dart , y le dice al compilador que el valor no cambiará. Declare la variable DeviceInfo String e inicialícela en una cadena vacía usando comillas dobles ("").

Para dar formato al resultado a la variable deviceInfo , utilice los caracteres \n para comenzar una nueva línea para cada pieza de información. Al usar la concatenación de cadenas, usa el signo += para agregar cada línea a la variable deviceInfo . En Swift, dentro de una cadena, utiliza la combinación de caracteres \\() para extraer el valor de la expresión que contiene.

```
función privada getDeviceInfo (resultado: FlutterResult) { let dispositivo  
    = UIDevice.current var deviceInfo: String  
    = deviceInfo = "\nName: \"  
    (device.name)"
```

```
DeviceInfo += "\nModelo: \\" + dispositivo.modelo + "\"  
DeviceInfo += "\nSistema: \\" + dispositivo.nombre del sistema + "\\" + dispositivo.version del sistema + "\""DeviceInfo += "\nSupervisión de  
proximidad habilitada: \\" + dispositivo.isProximityMonitoringEnabled + "\""DeviceInfo +=  
"\nMultitarea compatible: \\" +  
(device.isMultitaskingSupported) + "\"" result(DeviceInfo)  
}
```



CÓMO FUNCIONA

En la sección de host de iOS de la aplicación, está escuchando los mensajes entrantes utilizando FlutterMethodChannel. FlutterMethodChannel espera dos parámetros, el nombre y binaryMessenger . El parámetro de nombre es el nombre del canal de Flutter declarado en la plataforma de la aplicación del cliente.companyname.com/deviceinfo. El parámetro binaryMessenger es el FlutterViewController iniciado por la ventana de la aplicación iOS ?.rootViewController.

Configura una devolución de llamada con setMethodCallHandler para que coincida con el nombre del método entrante de getDeviceInfo mediante una instrucción if-else . Si call.method coincide con el nombre del método, llame al método getDeviceInfo para recuperar la información del dispositivo y devolver un resultado. La información del dispositivo se obtiene consultando el objeto UIDevice.current . Si call.method no coincide con el nombre del método entrante, la instrucción else devuelve FlutterMethodNotImplemented.

IMPLEMENTACIÓN DEL CANAL DE LA PLATAFORMA HOST DE ANDROID

El host es responsable de escuchar los mensajes entrantes del cliente. Una vez que se recibe un mensaje, el canal busca un nombre de método coincidente, ejecuta el método de llamada y devuelve el resultado apropiado. En Android, usa MethodChannel para escuchar los mensajes entrantes que toman dos parámetros. El primer parámetro es FlutterView, que amplía la actividad de la pantalla de una aplicación de Android y, al usar la variable flutterView como parámetro, es lo mismo que llamar al método getFlutterView() (FlutterView) desde la clase FlutterActivity . El segundo parámetro es el mismo nombre de canal de plataforma plataforma.nombredeempresa.com/infodedispositivo que el cliente.

```
privado val DEVICE_INFO_CHANNEL = "platformchannel.companyname.com/deviceinfo" val methodChannel =  
    MethodChannel(flutterView, DEVICE_INFO_CHANNEL)
```

Luego, usa setMethodCallHandler (controlador futuro) para configurar una devolución de llamada para un nombre de método coincidente que ejecuta el código de la plataforma nativa de Android. Una vez completado, envía el resultado al cliente.

```
methodChannel.setMethodCallHandler { llamada, resultado ->  
    // Comprobar el nombre de la llamada del método entrante y devolver un resultado  
}
```

Tanto MethodChannel como setMethodCallHandler se colocan en el método onCreate del archivo MainActivity.kt de la aplicación de Android . Se llama a onCreate cuando se crea la actividad por primera vez.

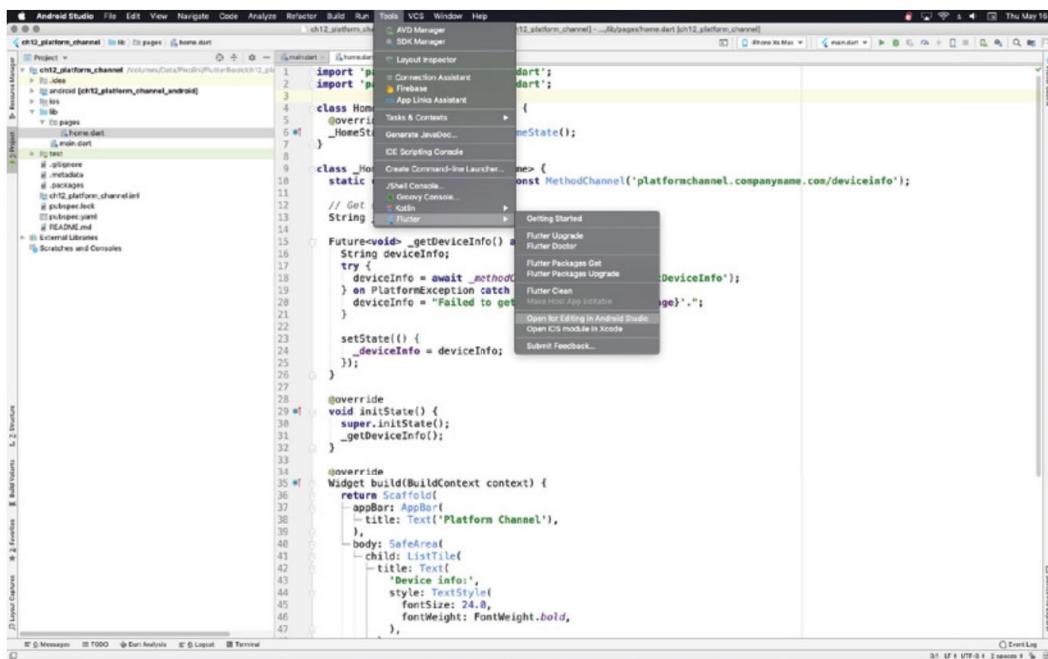
```
anula la diversión onCreate (savedInstanceState: ¿Paquete?) {  
    // código  
}
```

PRUÉBELO Creación del canal de plataforma de host de Android

En este ejemplo, desea recuperar la información del dispositivo en ejecución, como el fabricante, el modelo del dispositivo, el nombre, el sistema operativo y algunos otros detalles. El host de Android está escrito en Kotlin para acceder a la llamada API de la plataforma a la clase Build para consultar la información del dispositivo y utiliza MethodChannel para recibir la comunicación del cliente. Una vez que se recibe el mensaje, setMethodCallHandler maneja la solicitud y devuelve el resultado.

En esta sección, abrirá otra instancia de Android Studio y editarán el código Kotlin nativo de Android.

1. Si cerró el proyecto Flutter ch12_platform_channel, vuelva a abrirlo. Haga clic en la barra de menú Herramientas de Android Studio y seleccione Flutter Abrir para editar en Android Studio.



2. En el área de la ventana de la herramienta (en el lado izquierdo), expanda la carpeta de la aplicación haciendo clic en la flecha, abra la carpeta java , abra la carpeta com.domainname.ch12platformchannel y luego seleccione el archivo MainActivity.kt . Tenga en cuenta que el nombre de dominio puede ser diferente según el nombre que eligió al crear el proyecto Flutter.

The screenshot shows the Android Studio interface with the file structure on the left and the code editor on the right. The file `MainActivity.kt` is open in the code editor. The code implements a `GeneratedPluginRegistrant` interface and registers a `MethodChannel` for platform channel device information. It also defines a private method `getDeviceInfo` that returns a string containing device details like manufacturer, model, product, and version.

```

package com.domainname.ch13platformchannel;

import android.os.Bundle;
import io.flutter.app.FlutterActivity
import io.flutter.plugin.common.MethodChannel
import android.os.Build
import io.flutter.plugin.common.GeneratedPluginRegistrant

class MainActivity: FlutterActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    GeneratedPluginRegistrant.registerWith(this)
  }

  // Platform Channel
  val deviceInfoChannel = MethodChannel(flutterView, "platformchannel.companyname.com/deviceinfo")
  deviceInfoChannel.setMethodCallHandler { call, result =>
    if (call.method == "getDeviceInfo") {
      val deviceInfo = getDeviceInfo()
      result.success(deviceInfo)
    } else {
      result.notImplemented()
    }
  }
}

private fun getDeviceInfo(): String {
  return ("Device: " + Build.DEVICE
    + "\nManufacturer: " + Build.MANUFACTURER
    + "\nModel: " + Build.MODEL
    + "\nProduct: " + Build.PRODUCT
    + "\nSDK: " + Build.VERSION.RELEASE
    + "\nVersion SDK: " + Build.VERSION.SDK_INT
    + "\nFingerprint: " + Build.FINGERPRINT)
}

```

3. Agregue dos declaraciones de importación antes de la clase `MainActivity` . La primera declaración de importación agrega soporte para `MethodChannel`, y el segundo agrega soporte para usar `Build` para consultar la información del dispositivo.

```
importar io.flutter.plugin.common.MethodChannel importar  
android.os.Build
```

4. Edite el método `onCreate` y agregará código después de la llamada `GeneratedPluginRegistrant`.

Declare la variable deviceInfoChannel iniciándola con MethodChannel.

5. Para el primer parámetro de MethodChannel , BinaryMessenger , pase la variable flutterView ; tenga en cuenta que no declaró esta variable. No necesitas declararlo ya que es lo mismo que llamar al método getFlutterView() desde la clase FlutterActivity .

6. Para el segundo parámetro, pase el nombre del canal Flutter platformchannel.companyname.com/deviceinfo, el mismo declarado en el cliente.

```
val deviceInfoChannel = MethodChannel(flutterView, "platformchannel .companyname.com/deviceinfo")
```

7. Agregue deviceInfoChannel.setMethodCallHandler para configurar una devolución de llamada que coincide con el nombre del método entrante de getDeviceInfo mediante una instrucción if-else . Si call.method coincide (==) con getDeviceInfo , llama al método getDeviceInfo() (que crea en el siguiente paso) que recupera la información del dispositivo y el resultado se guarda en la variable deviceInfo . Si call.method no coincide con el nombre del método entrante, la instrucción else devuelve result.notImplemented(). El método notImplemented() responde a la llamada de que el método es desconocido o no está implementado. deviceInfoChannel.setMethodCallHandler { llamar, resultado -> si (llamar.método == "getDeviceInfo") { val deviceInfo =

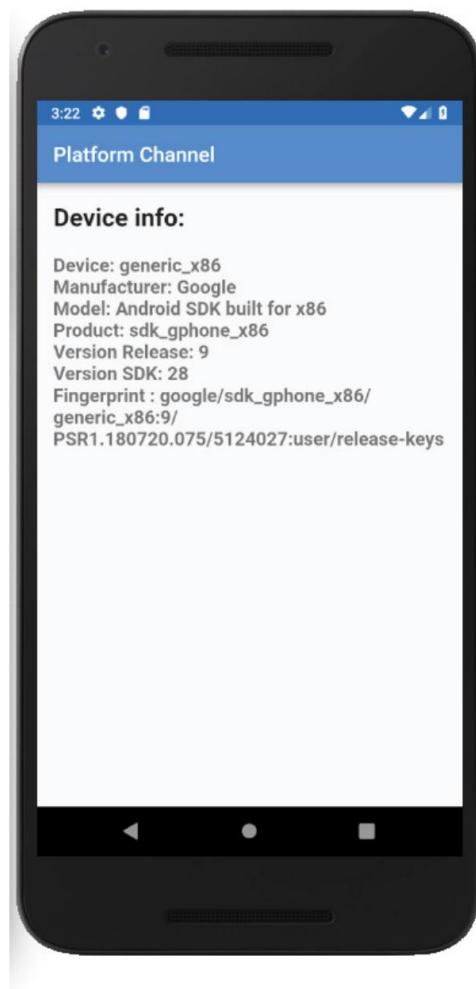
```
getDeviceInfo() result.success(deviceInfo) } else { result.notImplemented()
```

33

8. Después del método `onCreate`, agregue el método `getDeviceInfo(): String` que usa `Android Build` para consultar y recuperar la información actual del dispositivo.

Para dar formato al resultado, utilice los caracteres \n para comenzar una nueva línea para cada información. Al usar la concatenación de cadenas, usa el signo + para agregar cada línea y devolver el resultado formateado. Toda la concatenación de cadenas se incluye entre paréntesis de apertura y cierre ("...").

```
diversión privada getDeviceInfo(): String {  
    return ("\"nDispositivo: " + Construir.DISPOSITIVO  
        + "\nFabricante: " + "\nModelo: " + Build.FABRICANTE  
        + "\nProducto: " + Build.MODELO  
        + "\nLanzamiento de " + Construir.PRODUCTO  
        + "\nversión: " + "\nVersión SDK: " + Build.VERSION.RELEASE  
        + "\nHuella digital: " + Compilación.VERSIÓN.SDK_INT  
        + Compilación.FINGERPRINT)  
}
```



CÓMO FUNCIONA

En la sección de host de Android de la aplicación, está escuchando los mensajes entrantes mediante MethodChannel. MethodChannel espera dos parámetros, binaryMessenger y el nombre. El parámetro binaryMessenger es flutterView, que es el método getFlutterView() de la clase FlutterActivity que amplía la actividad de la pantalla de una aplicación de Android. El parámetro de nombre es el nombre del canal de Flutter declarado en la plataforma de la aplicación del cliente.companyname.com/deviceinfo.

Configura una devolución de llamada con setMethodCallHandler para que coincida con el nombre del método entrante de getDeviceInfo mediante una instrucción if-else . Si call.method coincide con el nombre del método, llame al método getDeviceInfo para recuperar la información del dispositivo y devolver un resultado. La información del dispositivo se obtiene consultando la clase Build . Si call.method no coincide con el nombre del método entrante, la instrucción else devuelve result.notImplemented().

RESUMEN

En este capítulo, aprendió cómo acceder y comunicarse con el código API específico de la plataforma iOS y Android mediante la implementación de canales de plataforma. Los canales de plataforma son una forma en que la aplicación Flutter (cliente) se comunica (mensajes) con iOS y Android (host) para solicitar y recibir resultados específicos del sistema operativo (SO). Para que la interfaz de usuario siga respondiendo y no se bloquee, los mensajes entre el cliente y el host son asíncronos.

Para comenzar a comunicarse desde la aplicación Flutter (cliente), aprendió a usar MethodChannel que envía mensajes que contienen llamadas a métodos para ser ejecutados por el lado de iOS y Android (host). Una vez que el host procesa el método solicitado, envía una respuesta al cliente y usted actualiza la interfaz de usuario para mostrar la información. El MethodChannel usa un nombre único y usó una combinación del nombre de la aplicación, el prefijo del dominio y el nombre de la tarea, como `plataformacanal.nombredelaempresa.com/infodeldispositivo`. Para iniciar la llamada desde el cliente y especificar qué método ejecutar en el host, aprendió a usar el constructor del método de llamada y se llama desde dentro de un método Future , ya que las llamadas son asíncronas.

Para el host de iOS y Android, aprendiste a usar FlutterMethodChannel en iOS y MethodChannel en Android para comenzar a recibir comunicaciones del cliente. El host es responsable de escuchar los mensajes entrantes del cliente. Usó setMethodCallHandler para configurar una devolución de llamada para un nombre de método coincidente entrante que se ejecuta en el código API específico de la plataforma nativa. Una vez que el método se completa, envía el resultado al cliente.

En el próximo capítulo, aprenderá a usar la persistencia local para guardar datos localmente en el área de almacenamiento del dispositivo.

LO QUE APRENDISTE EN ESTE CAPÍTULO

TEMA	CONCEPTOS CLAVE
Implementar MethodChannel (cliente)	Esto permite la comunicación desde la aplicación Flutter (cliente) mediante el envío de mensajes que contienen llamadas a métodos para ser ejecutados por iOS y Android (host).
Implementación de método de invocación (cliente)	Esto inicia (invoca) y especifica qué llamada de método ejecutar en el lado del host.
Implementar FlutterMethodChannel (anfitrión de iOS)	Esto permite la comunicación desde el host para recibir llamadas de método para ejecutar el código API específico de la plataforma.
Implementar MétodoChannel (anfitrión de Android)	
Implementar setMethodCallHandler (anfitrión de iOS y Android)	Esto configura una devolución de llamada para los nombres de métodos coincidentes entrantes para ejecutar el código API específico de la plataforma.

PARTE III

Creación de aplicaciones listas para producción

CAPÍTULO 13: Guardar datos con persistencia local

CAPÍTULO 14: Adición de Firebase y Firestore Backend

CAPÍTULO 15: Agregar administración de estado a Firestore

Aplicación de cliente

CAPÍTULO 16: Adición de BLoC a las páginas de la aplicación Firestore Client

13

Guardar datos con local Persistencia

LO QUE APRENDERÁS EN ESTE CAPÍTULO

Cómo continuar guardando y leyendo datos localmente

Cómo estructurar datos utilizando el formato de archivo JSON

Cómo crear clases modelo para manejar la serialización JSON

Cómo acceder a las ubicaciones de los sistemas de archivos locales de iOS y Android utilizando el paquete del proveedor de rutas

Cómo formatear fechas usando el paquete de internacionalización

Cómo usar la clase Future con showDatePicker para presentar un calendario para elegir fechas

Cómo usar la clase Future para guardar, leer y analizar archivos JSON

Cómo usar el constructor ListView.separated para seccionar registros con un divisor

Cómo usar List().sort para ordenar las entradas del diario por fecha

Cómo usartextInputAction para personalizar las acciones del teclado

Cómo usar FocusNode y FocusScope con el teclado onSubmit para mover el cursor al TextField de la siguiente entrada

Cómo pasar y recibir datos en una clase usando el Navegador

En este capítulo, aprenderá a conservar datos, es decir, guardar datos en el directorio de almacenamiento local del dispositivo, en los inicios de aplicaciones utilizando el formato de archivo JSON y guardando el archivo en el sistema de archivos local de iOS y Android. La notación de objetos de JavaScript (JSON) es un estándar abierto común

y formato de datos de archivo independiente del idioma con la ventaja de ser texto legible por humanos. La persistencia de datos es un proceso de dos pasos; primero usa la clase File para guardar y leer datos, y segundo, analiza los datos desde y hacia un formato JSON. Creará una clase para manejar el guardado y la lectura del archivo de datos que usa la clase File . También creará una clase para analizar la lista completa de datos mediante json.encode y json.decode y una clase para extraer cada registro. Y creará otra clase para manejar pasar una acción y una entrada de diario individual entre páginas.

Creará una aplicación de diario que guarde y lea datos JSON en el sistema de archivos local iOS NSDocumentDirectory y Android AppData . La aplicación usa un ListView para mostrar una lista de entradas de diario ordenadas por fecha, y creará una pantalla de entrada de datos para ingresar una fecha, un estado de ánimo y una nota.

ENTENDIENDO EL FORMATO JSON

El formato JSON está basado en texto y es independiente de los lenguajes de programación, lo que significa que cualquiera de ellos puede usarlo. Es una excelente manera de intercambiar datos entre diferentes programas porque es un texto legible por humanos. JSON usa el par clave/valor, y la clave está entre comillas seguidas de dos puntos y luego el valor como "id": "100". Utiliza una coma (,) para separar varios pares clave/valor.

La tabla 13.1 muestra algunos ejemplos.

TABLA 13.1: Pares Clave/Valor

LLAVE	COLON	VALOR
"identificación"	:	"100"
"cantidad"	:	3
"en stock"	:	verdadero

Los tipos de valores que puede usar son Object, Array, String, Boolean y Number. Los objetos se declaran entre corchetes ({}), y dentro de usted usa el par clave/valor y las matrices. Declaras arreglos usando los corchetes ([]), y dentro usas la clave/valor o solo el valor. La tabla 13.2 muestra algunos ejemplos.

TABLA 13.2: Objetos y matrices

TIPO	MUESTRA
Objeto	{ "id": "100", "nombre": "Vacaciones" }
Matriz solo con valores	["Familia", "Amigos", "Diversión"]

TIPO	MUESTRA
Matriz con clave/valor	[{ "id": "100", "nombre": "Vacaciones" }, { "id": "102", "nombre": "Cumpleaños" }]
Objeto con matriz	{ "id": "100", "nombre": "Vacaciones", "etiquetas": ["Familia", "Amigos", "Diversión"] }
Múltiples objetos con arreglos	{ "revistas": [{ "id": "4710827", "estado de ánimo": "Feliz" }, { "id": "427836", "mood": "Sorprendido" },], "etiquetas": [{ "id": "100", "nombre": "Familia" }, { "id": "102", "nombre": "Amigos" }]

El siguiente es un ejemplo del archivo JSON que creará para la aplicación de diario. El archivo JSON se usa para guardar y leer los datos del diario desde el área de almacenamiento local del dispositivo, lo que da como resultado la persistencia de los datos durante los inicios de la aplicación. Tiene los corchetes de apertura y cierre que declaran un objeto. Dentro del objeto, la clave del diario contiene una matriz de objetos separados por una coma. Cada objeto dentro de la matriz es una entrada de diario con pares clave/valor que declaran la identificación, la fecha, el estado de ánimo y la nota. El valor de la clave de identificación se usa para identificar de forma única cada entrada de diario y no se muestra en la interfaz de usuario. Cómo se obtiene el valor depende de los requisitos del proyecto; por ejemplo, puede usar números secuenciales o calcular un valor único usando caracteres y números (identificador único universal [UUID]).

```
{  
  "revistas": [  
    {  
      "id": "470827",  
      "date": "2019-01-13 00:27:10.167177",  
      "mood": "Feliz",  
      "note": "No puedo esperar a la noche familiar".  
    },  
    {  
      "id": "427836",  
      "date": "2019-01-12 19:54:18.786155",  
      "mood": "Feliz",  
      "note": "Gran día viendo nuestros programas favoritos".  
    }  
  ],  
}
```

USO DE CLASES DE BASE DE DATOS PARA ESCRIBIR, LEER Y SERIALIZAR JSON

Para crear código reutilizable para manejar las rutinas de la base de datos, como escribir, leer y serializar (codificar y decodificar) datos, colocará la lógica en clases. Creará cuatro clases para manejar la persistencia local, con cada clase responsable de tareas específicas.

La clase DatabaseFileRoutines usa la clase File para recuperar el documento local del dispositivo directorio y guarde y lea el archivo de datos.

La clase de base de datos es responsable de codificar y decodificar el archivo JSON y mapearlo a una Lista.

La clase Journal mapea cada entrada de diario desde y hacia JSON.

La clase JournalEdit se usa para pasar una acción (guardar o cancelar) y una apuesta de entrada de diario entre páginas.

La clase DatabaseFileRoutines requiere que importes la biblioteca dart:io para usar la clase File responsable de guardar y leer archivos. También requiere que importe el paquete path_provider para recuperar la ruta local al directorio del documento. La clase de base de datos requiere que importe la biblioteca dart:convert para decodificar y codificar objetos JSON.

La primera tarea en la persistencia local es recuperar la ruta del directorio donde se encuentra el archivo de datos en el dispositivo. Los datos locales generalmente se almacenan en el directorio de documentos de la aplicación; para iOS, la carpeta se llama NSDocumentDirectory y para Android es AppData. Para obtener acceso a estas carpetas, utiliza el paquete path_provider (complemento de Flutter). Llamará al método getApplicationDocumentsDirectory() , que devuelve el directorio que le da acceso a la variable de ruta .

```
Future<String> obtener _localPath asíncrono {  
  directorio final = espera getApplicationDocumentsDirectory();  
  
  volver directorio.ruta;  
}
```

Una vez que recupera la ruta, agrega el nombre del archivo de datos usando la clase `File` para crear un objeto `File`. Importa la biblioteca `dart:io` para usar la clase `File`, lo que le proporciona una referencia a la ubicación del archivo.

```
ruta final = espera _localPath; Archivo final  
= Archivo('$ruta/local_persistencia.json');
```

Una vez que tenga el objeto `File`, use el método `writeAsString()` para guardar el archivo pasando los datos como un argumento `String`. Para leer el archivo, utiliza el método `readAsString()` sin argumentos. Tenga en cuenta que la variable de archivo contiene la ruta de la carpeta de documentos y el nombre del archivo de datos.

```
// Escribe el archivo  
file.writeAsString('$json'); // Leer el  
archivo Cadena de  
contenido = esperar archivo.readAsString();
```

Como aprendió en la sección "Comprensión del formato JSON", utiliza un archivo JSON para guardar y leer datos del almacenamiento local del dispositivo. Los datos del archivo JSON se almacenan como texto sin formato (cadenas). Para guardar datos en el archivo JSON, utiliza la serialización para convertir un objeto en una cadena. Para leer datos del archivo JSON, utiliza la deserialización para convertir una cadena en un objeto. Utiliza el método `json.encode()` para serializar y el método `json.decode()` para deserializar los datos. Tenga en cuenta que los métodos `json.encode()` y `json.decode()` son parte de la clase `JsonCodec` de la biblioteca `dart:convert`.

Para serializar y deserializar archivos JSON, importa la biblioteca `dart:convert`. Después de llamar al método `readAsString()` para leer los datos del archivo almacenado, debe analizar la cadena y devolver el objeto JSON mediante la función `json.decode()` o `jsonDecode()`. Tenga en cuenta que la función `jsonDecode()` es una abreviatura de `json.decode()`.

```
// Cadena al objeto JSON final  
dataFromJson = json.decode(str); // O dataFromJson  
final =  
jsonDecode(str);
```

Para convertir valores en una cadena JSON, utilice la función `json.encode()` o `jsonEncode()`. Tenga en cuenta que `jsonEncode()` es una abreviatura de `json.encode()`. Es una preferencia personal decidir qué enfoque usar; en los ejercicios, utilizará `json.decode()` y `json.encode()`.

```
// Valores a cadena JSON  
json.encode(dataToJson); // O  
  
jsonEncode(dataToJson);
```

FORMATO DE FECHAS

Para dar formato a las fechas, utiliza el paquete `intl` (complemento de Flutter) que proporciona internacionalización y localización. La lista completa de formatos de fecha disponibles está disponible en el sitio de la página del paquete internacional en <https://pub.dev/packages/intl>. Para nuestros propósitos, utilizará la clase `DateFormat` para que le ayude a formatear N y analizar las fechas. Utilizará los constructores con nombre `DateFormat` para dar formato a la fecha de acuerdo con la especificación. Para dar formato a una fecha como el 13 de enero de 2019, utilice el constructor `DateFormat.yMMD()` y

luego pasa la fecha al argumento de formato , que espera un DateTime . Si pasa la fecha como una cadena, usa DateTime.parse() para convertirla a un formato DateTime .

```
// Ejemplos de formato de fecha
print(DateTime.d().format(DateTime.parse('2019-01-13')));
imprimir(DateTime.E().format(DateTime.parse('2019-01-13')));
print(DateTime.y().format(DateTime.parse('2019-01-13')));
imprimir(DateTime.yMEd().format(DateTime.parse('2019-01-13')));
print(DateTime.yMMEd().format(DateTime.parse('2019-01-13')));
print(DateTime.yMMMEEd().format(DateTime.parse('2019-01-13')));
```

```
Yo/aleteo (19337): 13
Yo/aleteo (19337): Sol
Yo/aleteo (19337): 2019
I/flutter (19337): domingo, 13/01/2019
I/flutter (19337): dom, 13 de enero de 2019
I/flutter (19337): domingo, 13 de enero de 2019
```

Para crear un formato de fecha personalizado adicional, puede encadenar y usar los métodos add_*() (sustituir el carácter * con los caracteres de formato necesarios) para agregar y combinar múltiples formatos. El siguiente código de ejemplo muestra cómo personalizar el formato de fecha:

```
// Ejemplos de formato de fecha con los métodos add_*
print(DateTime.yMEd().add_Hm().format(DateTime.parse('2019-01-13 10:30:15')));
print(DateTime.yMd().add_EEEE().add_Hms().format(DateTime.parse('2019-01-13 10:30:15')));
```

```
Yo/aleteo (19337): dom, 13/01/2019 10:30
I/aleteo (19337): 13/01/2019 domingo 10:30:15
```

ORDENAR UNA LISTA DE FECHAS

Aprendió cómo formatear fechas fácilmente, pero ¿cómo ordenaría las fechas? La aplicación de diario que creará requiere que muestre una lista de entradas, y sería genial poder mostrar la lista ordenada por fecha. En particular, desea ordenar las fechas para mostrar primero la más reciente y la última la más antigua, lo que se conoce como orden DESC (descendente). Las entradas de nuestro diario se muestran desde una lista y, para ordenarlas, llama al método List().sort() .

La Lista se ordena según el orden especificado por la función, y la función actúa como Comparador, comparando dos valores y evaluando si son iguales o si uno es más grande que el otro, como las fechas 2019-01-20 y 2019-01-22 en la Tabla 13.3. La función Comparator devuelve un número entero como negativo, cero o positivo. Si la comparación, por ejemplo, 2019-01-20 > 2019-01-22, es verdadera, devuelve 1, y si es falsa, devuelve -1. De lo contrario (cuando los valores son iguales), devuelve 0.

TABLA 13.3: Clasificación de fechas

COMPARAR	VERDADERO	MISMO	FALSO
fecha2.comparar con(fecha1)	1	0	-1
2019-01-20 > 2019-01-22			-1
2019-01-20 < 2019-01-22	1		
2019-01-22 = 2019-01-22		0	

Echemos un vistazo ejecutando la siguiente ordenación con valores DateTime reales ordenados por fecha DESC.

Tenga en cuenta que para ordenar por DESC, comience con la segunda fecha, comparándola con la primera fecha de esta manera: comp2.date.compareTo(comp1.date).

```
_database.journal.sort((comp1, comp2) => comp2.date.compareTo(comp1.date));

// Resultados de print() al registro l/flutter
(10272): -1 - 2019-01-20 15:47:46.696727 - 2019-01-22 17:02:47.678590 l/flutter (10272): -1 - 2019-01-19
15:58:23.013360 - 2019-01-20 15:47:46.696727 l/aleteo (10272): -1 - 2019-01-19 13:04:32.812748 -
2019-01-19 15:58:23.013360 l/flutter (10272): 1 - 2019-01-22 17:21:12.752577 - 2018-01-01
16:43:05.598094 l/flutter (10272): 1 - 2019-01-22 17:21:12.752577 - 2018-12-25 02:40:55.533173 l/
aleteo (10272): 1 - 2019-01-22 17:21:12.752577 - 2019-01-16 02:40:13.961852
```

Quería mostrarle el camino más largo hacia la clasificación () del código anterior para mostrar cómo se obtiene el resultado de la comparación.

```
_base de datos.journal.sort((comp1, comp2) {
  resultado int = comp2.date.compareTo(comp1.date);
  print("$resultado - ${comp2.fecha} - ${comp1.fecha}"); resultado
  devuelto;});
```

Si desea ordenar las fechas por orden ASC (ascendente), puede cambiar la declaración de comparación para comenzar con comp1.date a comp2.date.

```
_database.journal.sort((comp1, comp2) => comp1.date.compareTo(comp2.date));
```

RECUPERACIÓN DE DATOS CON EL FUTUREBUILDER

En las aplicaciones móviles, es importante no bloquear la interfaz de usuario mientras se recuperan o procesan datos. En el Capítulo 3, "Aprendizaje de los conceptos básicos de Dart", aprendió a usar un futuro para recuperar un valor posible que esté disponible en algún momento en el futuro. Un widget de FutureBuilder funciona con un futuro para recuperar los datos más recientes sin bloquear la interfaz de usuario. Las tres propiedades principales que establece son initialData, future y builder.

initialData: datos iniciales para mostrar antes de recuperar la instantánea.

Código de muestra:

[]

futuro: llama a un método asíncrono futuro para recuperar datos.

Código de muestra:

_ cargar diarios()

builder: la propiedad builder proporciona BuildContext y AsyncSnapshot (datos recuperado y estado de la conexión). AsyncSnapshot devuelve una instantánea de los datos, y también puede verificar ConnectionState para obtener un estado en el proceso de recuperación de datos.

Código de muestra:

(Contexto BuildContext, instantánea AsyncSnapshot)

AsyncSnapshot: proporciona los datos y el estado de conexión más recientes. Tenga en cuenta que los datos representados son inmutables y de solo lectura. Para comprobar si se devuelven datos, utiliza el snapshot.hasData. Para verificar el estado de la conexión, use snapshot.connectionState para ver si el estado está activo, en espera, terminado o ninguno. También puede buscar errores mediante la propiedad snapshot.hasError .

Código de muestra:

```
constructor: (contexto BuildContext, instantánea AsyncSnapshot) {  
    volver linstantánea.hasData  
    ? Indicador de progreso circular ()  
    : _ buildListView(instantánea);  
},
```

El siguiente es un código de muestra de FutureBuilder() :

```
FutureBuilder( initialData:  
[], future: _loadJournals(),  
builder: (Contexto BuildContext, instantánea AsyncSnapshot) { return !  
snapshot.hasData  
? Center(child: CircularProgressIndicator()):  
_buildListViewSeparated(instantánea);  
}, ),
```

CONSTRUYENDO LA APLICACIÓN DEL DIARIO

Creará una aplicación de diario con el requisito de que los datos persistan entre los inicios de la aplicación. Los datos se almacenan como objetos JSON con los requisitos de seguimiento de la identificación, la fecha, el estado de ánimo y la nota de cada entrada del diario (Figura 13.1). Como aprendió en la sección "Comprendiendo el formato JSON", el valor de la clave de identificación es único y se usa para identificar cada entrada de diario. La clave de identificación se usa en segundo plano para seleccionar la entrada del diario y no se muestra en la interfaz de usuario. El objeto raíz es un par clave/valor con el nombre clave de 'diario' y el valor como una matriz de objetos que contiene cada entrada de diario.

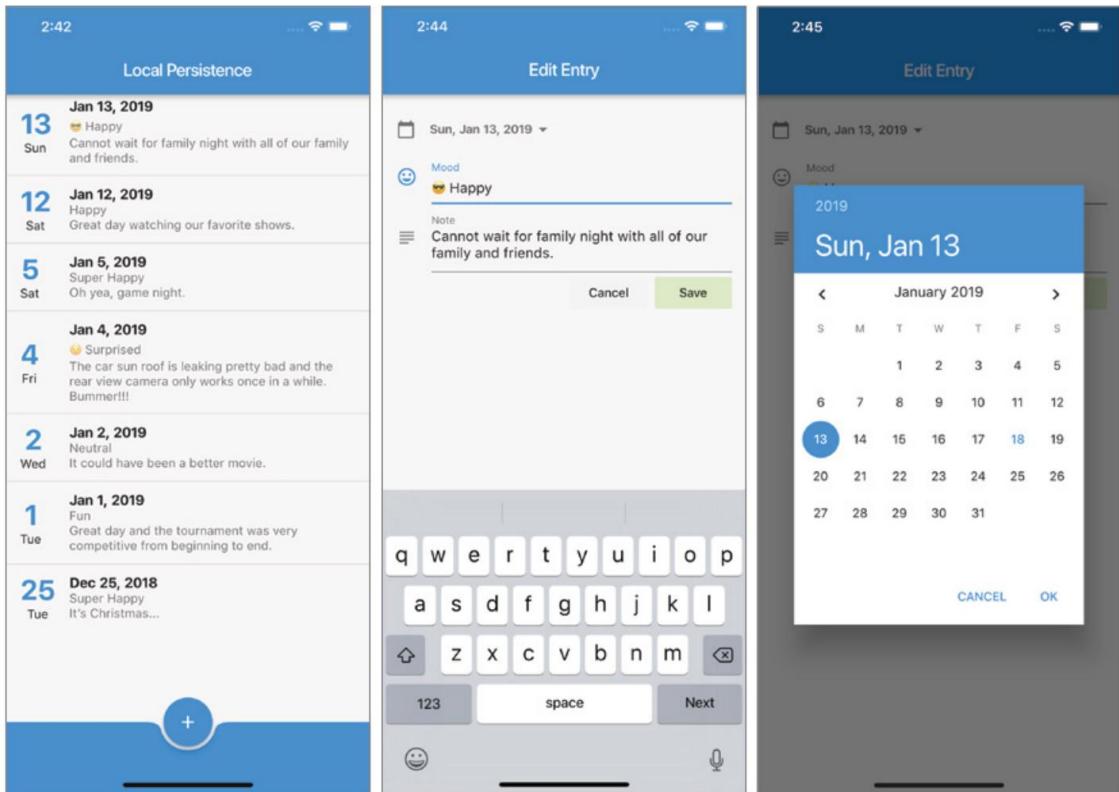


FIGURA 13.1: Aplicación Diario

La aplicación tiene dos páginas separadas; la página de presentación principal utiliza un ListView ordenado por fecha DESC, lo que significa que el último registro ingresado es el primero. Utiliza el widget ListTile para formatear cómo se muestra la Lista de registros. La segunda página son los detalles de la entrada del diario donde usa un selector de fecha para seleccionar una fecha de un calendario y widgets de TextField para ingresar el estado de ánimo y los datos de la nota. Creará un archivo Dart de base de datos con clases para manejar las rutinas de archivos, el análisis JSON de la base de datos, una clase Journal para manejar registros individuales y una clase JournalEdit para pasar datos y acciones entre páginas.

La figura 13.2 muestra una vista de alto nivel de la aplicación de la revista que detalla cómo se utilizan las clases de la base de datos en las páginas de inicio y edición.

DESCRIPCIÓN GENERAL DE LA APLICACIÓN DEL DIARIO

Creará esta aplicación en el transcurso de cuatro ejercicios de Pruébelo:

Sentar las bases de la aplicación Journal: en la primera sección, agrega los paquetes path_provider e intl a su archivo pubspec.yaml y configura la página home.dart con widgets de estructura básica.

Crear las clases de la base de datos de Journal: el segundo ejercicio se enfoca en crear el archivo database.dart con sus clases para manejar las rutinas de archivos, el análisis de la base de datos y las clases de Journal .

Creación de la página de entrada de diario: el tercer ejercicio crea el archivo edit_entry.dart para manejar la creación y edición de entradas de diario y la selección de fechas de un selector de fechas.

Finalización de la página de inicio de la revista: el cuarto ejercicio finaliza la página home.dart , que se basa en el archivo database.dart al agregar lógica para crear ListView y guardar, leer, clasificar datos y pasar datos a la página de edición de la revista.

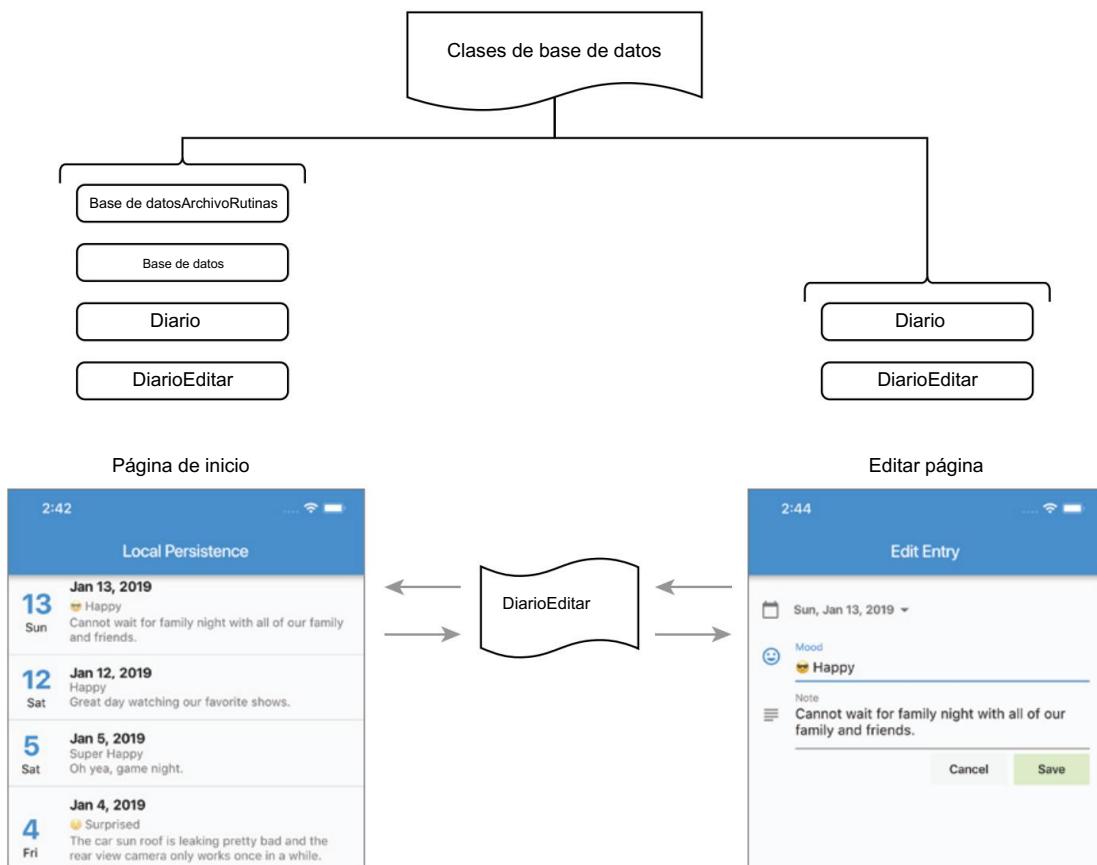


FIGURA 13.2: Relación de las clases de la base de datos de la aplicación Journal con las páginas de inicio y edición

PRUÉBELO Sentando las bases de la aplicación Journal

En esta serie de pasos, realizará una configuración esencial para la aplicación de diario, que desarrollará en las secciones Pruébelo restantes de este capítulo.

1. Cree un nuevo proyecto de Flutter y asígnale el nombre ch13_local_persistence. Puede seguir las instrucciones del Capítulo 4, "Creación de una plantilla de proyecto inicial". Para este proyecto, necesita crear solo las carpetas de páginas y clases .
2. Abra el archivo pubspec.yaml para agregar recursos. En la sección dependencies:, agregue las declaraciones path_provider: ^1.1.0 e intl: ^0.15.8 . Tenga en cuenta que su versión podría ser superior.

dependencias:

 aleteo:

 SDK: aleteo

 # Lo siguiente agrega la fuente Cupertino Icons a su aplicación.

 # Usar con la clase CupertinoIcons para íconos de estilo iOS. icons/cupertino:

 ^0.1.2

 proveedor_ruta: ^1.1.0 intl:

 ^0.15.8

3. Haga clic en el botón Guardar. Dependiendo del editor que estés usando, automáticamente ejecuta flutter packages get, y una vez que termine, mostrará un mensaje de Proceso terminado con el código de salida 0. Si no ejecuta automáticamente el comando por usted, abra la ventana Terminal (ubicada en la parte inferior de su editor) y escriba flutter packages get.

4. Abra el archivo main.dart . Agregue a ThemeData la propiedad bottomAppBarColor y configure el color a Colors.blue.

```
return MaterialApp  
  (debugShowCheckedModeBanner: falso, título:  
  'Persistencia local', tema: ThemeData  
  (primarySwatch:  
    Colors.blue, bottomAppBarColor:  
    Colors.blue, ), home: Home(), );
```

5. Abra el archivo home.dart y agregue al cuerpo un FutureBuilder(). El constructor del futuro()

La propiedad initialData es una lista vacía creada con los corchetes de apertura y cierre ([]). La propiedad future llama al método Future _loadJournals() que creó en el ejercicio "Finalizar la página de inicio de la revista". Para la propiedad del constructor , devuelve un Indicador de progreso circular () si snapshot.hasData es falso, lo que significa que aún no se han devuelto datos. De lo contrario, llame al método _buildListViewSeparated(snapshot) para generar el ListView que muestra las entradas del diario.

Como aprendió en la sección "Recuperación de datos con FutureBuilder", AsyncSnapshot devuelve una instantánea de los datos. La instantánea es inmutable, lo que significa que es de solo lectura.

cuerpo:

 FutureBuilder(initialData:

 [], futuro: _loadJournals(),

 constructor: (contexto BuildContext, instantánea AsyncSnapshot) {

```
volver linstantánea.hasData  
    ? Center(child: CircularProgressIndicator()):  
    _buildListViewSeparated(instantánea);  
, ),
```

6. Después de la propiedad del cuerpo , agregue la propiedad bottomNavigationBar y establezcala en BottomAppBar(). Establece la forma en CircularNotchedRectangle() y configura el elemento secundario en un Padding de 24,0 píxeles. Agregue la propiedad floatingActionButtonLocation y establezcala en FloatingActionButton Location.centerDocked.

```
bottomNavigationBar: BottomAppBar(  
    forma: CircularNotchedRectangle(), child:  
    Padding(padding: const EdgeInsets.all(24.0)), ), floatingActionButtonLocation:  
  
    FloatingActionButtonLocation.centerDocked,
```

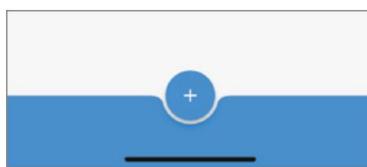
7. Agregue la propiedad floatActionButton y establezcala en FloatingActionButton(). Normalmente BottomAppBar () se usa para mostrar una fila de widgets para hacer una selección, pero para nuestra aplicación, la está usando para una apariencia estética al usarla con FloatingActionButton para mostrar una muesca. El FloatingActionButton es responsable de agregar nuevas entradas de diario. Establezca la propiedad secundaria FloatingActionButton() en Icon(Icons.add) para mostrar un signo más que transmita la acción para agregar una nueva entrada de diario.

```
botón de acción flotante: botón de acción flotante (  
    información sobre herramientas: 'Agregar  
    entrada de diario', hijo:  
    Icon(Icons.add), ),
```

8. Establezca la propiedad FloatingActionButton() onPressed como una devolución de llamada asíncrona que llama al Método _addOrEditJournal() . Agrega una llamada al método _addOrEditJournal() que toma tres argumentos: agregar, indexar y diario. En "Finalizar la página de inicio de la revista", creará el método que se basa en la creación del archivo database.dart .

Cuando el usuario toca este botón, es para agregar una nueva entrada, por lo que pasa los argumentos agregar como verdadero, indexar como -1 y diario como una entrada de Diario (clase) en blanco. Debido a que también usa el mismo método para editar una entrada (el usuario toca ListView, cubierto en el ejercicio final), pasaría los argumentos add como falsos, index como el índice de entrada de ListView y journal como el Journal seleccionado.

```
bottomNavigationBar: BottomAppBar(  
    forma: CircularNotchedRectangle(), child:  
    Padding(padding: const EdgeInsets.all(24.0)), ), flotanteActionButtonLocation:  
  
    FloatingActionButtonLocation.centerAcoplado, floatingActionButton: FloatingActionButton(  
  
        información sobre herramientas: 'Agregar  
        entrada de diario', niño: Icono  
        (Icons.add), onPressed: () {  
            _addOrEditJournal(agregar: verdadero, índice: -1, diario: Diario()); }, ),
```



CÓMO FUNCIONA

Declaró los paquetes path_provider e intl en su archivo pubspec.yaml . El path_provider le brinda acceso a las ubicaciones locales del sistema de archivos de iOS y Android, y el intl le brinda la capacidad de formatear las fechas. Agregó a la propiedad del cuerpo el FutureBuilder que llama a un método asíncrono futuro para devolver datos, y lo implementará en el cuarto ejercicio, "Finalizar la página de inicio de la revista".

Agregó un BottomAppBar y usó FloatingActionButtonLocation para acoplar el FloatingAction Button en la parte inferior central. El evento FloatingActionButton onPressed() se marca como asíncrono para llamar al método _addOrEditJournal() para agregar una nueva entrada de diario. Implementará el método _addOrEditJournal() en "Finalizar la página de inicio de la revista".

Adición de las clases de base de datos de revistas

Creará cuatro clases separadas para manejar las rutinas de la base de datos y la serialización para administrar los datos del diario. Cada clase es responsable de manejar la lógica del código específico, lo que resulta en la reutilización del código. Colectivamente, las clases de la base de datos son responsables de escribir (guardar), leer, codificar y decodificar objetos JSON hacia y desde el archivo JSON.

La clase DatabaseFileRoutines se encarga de obtener la ruta al directorio de documentos locales del dispositivo y de guardar y leer el archivo de la base de datos mediante la clase File . La clase File se usa al importar la biblioteca dart:io , y para obtener la ruta del directorio de documentos, importa el paquete path_provider .

La clase de base de datos se encarga de decodificar y codificar los objetos JSON y convertirlos en una lista de entradas de diario. Llamas a databaseFromJson para leer y analizar desde objetos JSON. Llamas a databaseToJson para guardar y analizar objetos JSON. La clase de base de datos devuelve la variable de diario que consta de una lista de clases de diario , List<Journal>. La biblioteca dart:convert se utiliza para decodificar y codificar objetos JSON.

La clase Diario maneja la decodificación y codificación de los objetos JSON para cada diario entrada. La clase Journal contiene los campos de entrada de diario id, date, mood y note almacenados como cadenas.

La clase JournalEdit maneja el paso de entradas de diario individuales entre páginas.

La clase JournalEdit contiene las variables action y journal . La variable de acción se usa para rastrear si se presiona el botón Guardar o Cancelar. La variable de diario contiene la entrada de diario individual como una clase de diario que contiene las variables de identificación, fecha, estado de ánimo y nota .

PRUÉBELO Crear las clases de la base de datos de revistas

En esta sección, creará las clases `DatabaseFileRoutines`, `Database`, `Journal` y `JournalEdit`.

Tenga en cuenta que los constructores predeterminados usan corchetes (`{}`) para implementar parámetros con nombre.

1. Cree un nuevo archivo Dart en la carpeta de clases . Haga clic con el botón derecho en la carpeta de clases , seleccione Nuevo Archivo Dart, ingrese `base de datos.dart` y haga clic en el botón Aceptar para guardar.
2. Importe el paquete `path_provider.dart` y las bibliotecas `dart:io` y `dart:convert` . Agrega una nueva línea y cree la clase `DatabaseFileRoutines` .

```
importar 'paquete: ruta_proveedor/ruta_proveedor.dart'; // Ubicaciones del sistema de archivos import  
'dart:io'; // Usado por File import 'dart:convert'; //  
Usado por json
```

```
class Base de datosArchivoRutinas {  
  
}
```

3. Dentro de la clase `DatabaseFileRoutines` , agregue el método asíncrono `_localPath` que devuelve un `Future<String>`, que es la ruta del directorio de documentos.

```
Future<String> obtener _localPath asíncrono {  
    directorio final = espera getApplicationDocumentsDirectory();  
  
    volver directorio.ruta;  
}
```

4. Agregue el método asíncrono `_localFile` que devuelve un `Future<File>` con la referencia al archivo `local_persistence.json` , que es la ruta, combinada con el nombre del archivo.

```
Future<File> get _localFile async { ruta final =  
    esperar _localPath;  
  
    devolver Archivo('$ruta/local_persistencia.json');  
}
```

5. Agregue el método asíncrono `readJournals()` que devuelve un `Future<String>` que contiene los objetos JSON. Utilizará un intento de captura en caso de que haya un problema con la lectura del archivo.

```
Future<String> readJournals() asíncrono {  
    intentar {  
  
        } atrapar (e) {  
  
    }}}
```

6. Use `file.existsSync()` para verificar si el archivo existe; si no, lo crea llamando al `writeJournals({ "journals": [] })` pasándole un objeto `journals` vacío. A continuación, se llama a `file.readAsString()` para cargar el contenido del archivo.

```
Future<String> readJournals() asíncrono {  
    prueba  
        {archivo final = espera _localFile;  
  
        if (!file.existsSync()) {
```

```

        print("El archivo no existe: ${archivo.absoluto}"); await
        writeJournals({'diarios': []}); }

        // Leer el archivo Cadena
        de contenido = esperar archivo.readAsString();

        devolver contenidos;
    } catch (e)
    { print("error de lectura de diarios: $e"); devolver
    "",

}
}

```

7. Agregue el método asíncrono `writeJournals(String json)` que devuelve un `Future<File>` para guardar los objetos JSON en un archivo.

```

Future<File> writeJournals(String json) async { archivo final = await
    _localFile;

    // Escribir el archivo return
    file.writeAsString('$json');
}

```

8. Siguiendo la clase `DatabaseFileRoutines`, cree dos métodos que llamen a la clase `Database` para manejar la decodificación y codificación JSON para toda la base de datos. Cree el método `databaseFromJson- (String str)` que devuelve una base de datos pasándole la cadena JSON. Al usar `json.decode(str)`, analiza la cadena JSON y devuelve un objeto JSON.

```

// Para leer y analizar desde datos JSON - base de datosFromJson(jsonString); Base de datos base de
datosFromJson(String str) { datos finalesFromJson =
    json.decode(str); volver Base de datos.fromJson(dataFromJson);

}

```

9. Cree el método `databaseToJson(Database data)` que devuelve una cadena. Usando el `json.encode(dataToJson)`, analiza los valores en una cadena JSON.

```

// Para guardar y analizar datos JSON - databaseToJson(jsonString); String base de datos a Json
(datos de la base de datos) { datos finales a Json = datos.
    a Json (); devuelve json.encode(dataToJson);

}

```

10. Cree la clase de base de datos , y el primer elemento a declarar es la variable de diario de un Tipo `List<Journal>` , lo que significa que contiene una lista de revistas. La clase `Diario` contiene cada registro y lo creará en el paso 13. Declare el constructor de la base de datos con el parámetro nombrado variable `this.journal` . Tenga en cuenta que está utilizando corchetes `({})` para declarar el parámetro llamado constructor.

```

base de datos de clase {
    List<diario> diario;

    Base de datos
    ({este diario,}); }

```

11. Para recuperar y asignar los objetos JSON a List<Journal> (lista de clases de Journal), cree el Factory Database.fromJson() llamado constructor. Tenga en cuenta que el constructor de fábrica no siempre crea una nueva instancia, pero puede devolver una instancia de un caché. El constructor toma el argumento de Map<String, dynamic>, que asigna la clave String con un valor dinámico , el par clave/valor JSON. El constructor devuelve List<Journal> tomando los objetos clave JSON 'journals' y asignándolos desde la clase Journal que analiza la cadena JSON al objeto Journal que contiene cada campo, como la identificación, la fecha, el estado de ánimo y la nota.

```
fábrica Base de datos.fromJson(Map<String, dynamic> json) => Base de datos(
    journal: List<Journal>.from(json["journals"].map((x) => Journal.fromJson(x))), );
```

12. Para convertir List<Journal> en objetos JSON, cree el método toJson que analiza cada Clase diario a objetos JSON.

```
Map<String, dynamic> toJson() => { "journals":
    List<dynamic>.from(journal.map((x) => x.toJson())), };
```

La siguiente es la clase completa de la base de datos :

```
base de datos de clase {
    List<diario> diario;

    Base de datos
    ({este diario,});
```

```
fábrica Base de datos.fromJson(Map<String, dynamic> json) => Base de datos(
    journal: List<Journal>.from(json["journals"].map((x) => Journal.fromJson(x))), );
```

```
Map<String, dynamic> toJson() => { "journals":
    List<dynamic>.from(journal.map((x) => x.toJson())), };

}
```

13. Cree la clase Journal y declare como tipo String las variables id, date, mood y note .

Declare el constructor Journal con los parámetros nombrados this.id , this.date , this.mood y this.note variables. Tenga en cuenta que está utilizando corchetes ({}) para declarar los parámetros con nombre del constructor.

```
diario de clase {
    ID de cadena;
    Fecha de cadena;
    Estado de ánimo de cadena;
    Nota de cuerda;

    Diario({ este.id,
        esta.fecha,
        este.estado
        de ánimo,
        esta.nota, });

}
```

14. Para recuperar y convertir el objeto JSON en una clase `Journal` , cree el constructor con nombre `Journal.fromJson()` de fábrica . El constructor toma el argumento de `Map<String, dynamic>`, que asigna la clave `String` con un valor dinámico , el par clave/valor JSON.

```
factory Journal.fromJson(Map<String, dynamic> json) => Journal( id: json["id"], date: json["date"],
mood: json["mood"],
note: json["note "], );
```

15. Para convertir la clase `Journal` en un objeto JSON, cree el método `toJson()` que analiza el Clase de diario a un objeto JSON.

```
Map<String, dynamic> toJson() => { "id": id, "fecha":  
fecha,  
"estado de ánimo":  
estado de ánimo,  
"nota": nota,  
};
```

La siguiente es toda la clase `Diario` :

```
diario de clase {  
ID de cadena;  
Fecha de cadena;  
Estado de ánimo de cadena;  
Nota de cuerda;  
  
Diario({ este.id,  
esta.fecha,  
este.estado  
de ánimo,  
esta.nota, });  
  
factory Journal.fromJson(Map<String, dynamic> json) => Journal( id: json["id"], date: json["date"],  
mood: json["mood"],  
note: json["note "], );  
  
Map<String, dynamic> toJson() => { "id": id, "fecha":  
fecha,  
"estado de ánimo":  
estado de ánimo,  
"nota": nota, };  
}
```

16. Cree la clase `JournalEdit` que es responsable de pasar la acción y una entrada de diario entre páginas. Agregue una variable de acción de cadena y una variable de diario de diario . Agregar el predeterminado Constructor `JournalEdit` .

```
diario de clase Editar {  
Acción de cadena;
```

```
diario diario;  
  
JournalEdit({this.action, this.journal}); }
```

CÓMO FUNCIONA

Creó un archivo database.dart que contiene cuatro clases para manejar la serialización y deserialización de persistencia local.

La clase DatabaseFileRoutines maneja la ubicación de la ruta del directorio de documentos locales del dispositivo a través del paquete path_provider . Usó la clase File para manejar el guardado y la lectura del archivo de la base de datos importando la biblioteca dart:io . El archivo está basado en texto y contiene el par clave/valor de objetos JSON.

La clase de base de datos usa json.encode y json.decode para serializar y deserializar objetos JSON importando la biblioteca dart:convert . Utiliza el constructor con nombre Database.fromJson para recuperar y asignar los objetos JSON a List<Journal>. Utiliza el método toJson() para convertir List<Journal> en objetos JSON.

La clase Journal es responsable de realizar un seguimiento de las entradas de diario individuales a través de las variables de identificación, fecha, estado de ánimo y nota de la cadena. Utiliza el constructor con nombre Journal.fromJson() para tomar el argumento de Map<String, dynamic>, que asigna la clave String con un valor dinámico , el par clave/valor JSON. Utiliza el método toJson() para convertir la clase Journal en un objeto JSON.

La clase JournalEdit se utiliza para pasar datos entre páginas. Declaró una variable de acción de cadena y una variable de diario de diario . La variable de acción pasa una acción a 'Guardar' o 'Cancelar', editando una entrada. Aprenderá a usar la clase JournalEdit en los ejercicios "Creación de la página de entrada del diario" y "Terminación de la página de inicio". La variable de diario pasa los valores de entrada de diario.

Adición de la página de entrada del diario

La página de entrada es responsable de agregar y editar una entrada de diario. Podría preguntar, ¿cómo sabe cuándo agregar o editar una entrada actual? Creó la clase JournalEdit en el archivo database.dart por este motivo: para permitirle reutilizar la misma página para múltiples propósitos. La página de entrada amplía un StatelessWidget con el constructor (Tabla 13.4) que tiene los tres argumentos add, index y journalEdit. Tenga en cuenta que el argumento de índice se utiliza para realizar un seguimiento de la ubicación de la entrada de diario seleccionada en la lista de la base de datos de diario de la página de inicio. Sin embargo, si se crea una nueva entrada de diario, aún no existe en la lista, por lo que se pasa un valor de -1 en su lugar. Cualquier número de índice cero y superior significaría que la entrada del diario ya existe en la lista.

TABLA 13.4: Argumentos del constructor de la clase EditEntry

VARIABLE	DESCRIPCIÓN Y VALOR
bool final agregar	Si el valor de agregar variable es verdadero, significa que está agregando un nuevo diario. Si el valor es falso, está editando una entrada de diario.
índice int final	Si el valor de la variable de índice es -1, significa que está agregando una nueva entrada de diario. Si el valor es 0 o mayor, está editando una entrada de diario y necesita realizar un seguimiento de la posición del índice en List<Journal>.

VARIABLE	DESCRIPCIÓN Y VALOR
Diario finalEditar	La clase JournalEdit pasa dos valores. El valor de la acción es 'Guardar' o 'Cancelar'.
diarioEditar	La variable journal pasa toda la clase Journal , que consta de los valores id, date, mood y note .
Acción de cadena	
diario diario	

La página de entrada tiene botones Cancelar y Guardar que llaman a una acción con el método onPressed() (Tabla 13.5). El método onPressed() devuelve a la página de inicio la clase JournalEdit con los valores apropiados según el botón que se presione.

TABLA 13.5: Guardar o Cancelar FlatButton

PRESIONADO() RESULTADO	
Cancelar	<p>La variable de acción JournalEdit se establece en 'Cancelar' y la clase vuelve a la página de inicio con Navigator.pop(context, _journalEdit).</p> <p>La página de inicio recibe los valores y no realiza ninguna acción desde que se canceló la edición.</p>
Ahorrar	<p>La variable de acción JournalEdit se establece en 'Guardar' y la variable de diario se configura con los valores de clase de diario actuales, la identificación, la fecha, el estado de ánimo y la nota. Si el valor agregado es igual a verdadero, lo que significa agregar una nueva entrada, se genera un nuevo valor de identificación . Si el valor agregado es igual a falso, lo que significa editar una entrada, se usa la identificación del diario actual.</p> <p>La página de inicio recibe los valores y ejecuta la lógica 'Guardar' con los valores recibidos.</p>

Para facilitar al usuario la selección de una fecha, utilice el selector de fechas integrado que presenta un calendario. Para mostrar el calendario, llame a la función showDatePicker() (Tabla 13.6) y pase cuatro argumentos: context, initialDate, firstDate y lastDate (Figura 13.3).

TABLA 13.6: showDatePicker

PROPIEDAD	VALOR
contexto	Pasa el BuildContext como el contexto.
initialDate	Pasa la fecha del diario que está resaltada y seleccionada en el calendario.
primera fecha	El intervalo de fechas más antiguo disponible para seleccionar en el calendario a partir de la fecha de hoy.
ultima cita	El rango de fechas más nuevo disponible para seleccionar en el calendario a partir de la fecha de hoy.

Una vez que se recupera la fecha, usará el constructor DateFormat.yMMMd() para mostrarla en formato de domingo, 13 de enero de 2018. Si desea mostrar un selector de tiempo, llame al método showTimePicker() y pase los argumentos context y initialTime .



FIGURA 13.3: Calendario selector de fecha

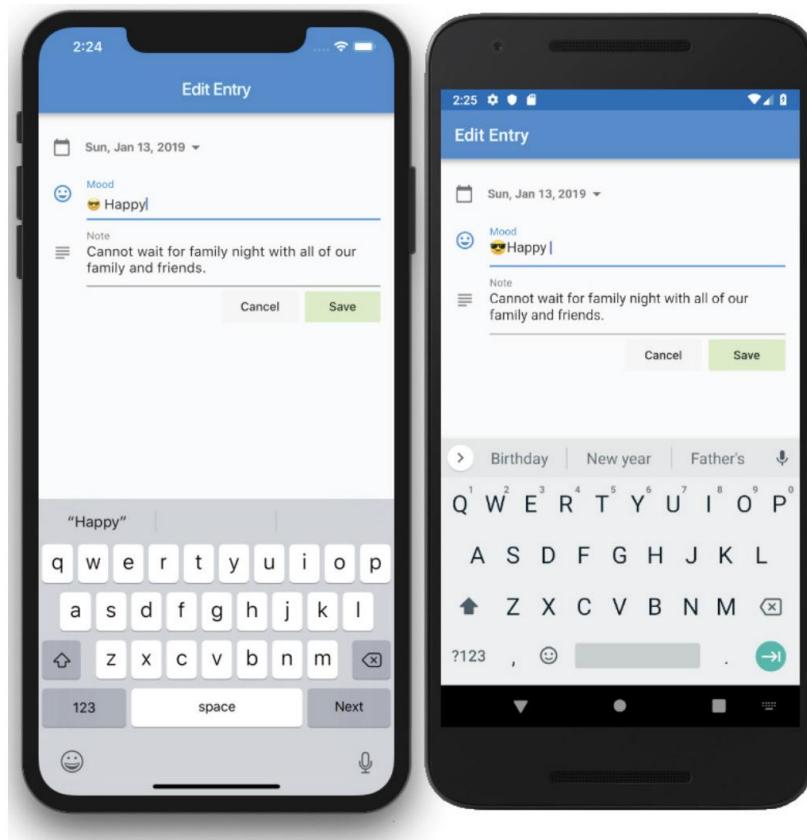
En el Capítulo 6, "Uso de widgets comunes", aprendió a usar un formulario con `TextField` para crear un formulario de entrada. Ahora usemos un enfoque diferente sin un formulario, pero usemos `TextField` con un `TextEditingController`. Aprenderá a usar `TextField TextInputType` con `FocusNode` para personalizar el botón de acción del teclado para ejecutar una acción personalizada (Figura 13.4). El botón de acción del teclado se encuentra a la derecha de la barra espaciadora. También aprenderá a personalizar las opciones de mayúsculas y minúsculas de `TextField` mediante `TextCapitalization` que configura cómo el teclado capitaliza palabras, oraciones y caracteres; los ajustes son palabras, oraciones, caracteres o ninguno (predeterminado).



FIGURA 13.4: Botón de acción del teclado para iOS y Android

PRUÉBALO Creación de la página de entrada de diario

En esta sección, creará `EditEntry StatefulWidget` con el constructor tomando los argumentos `add`, `index` y `journalEdit`. Tenga en cuenta que los constructores predeterminados usan llaves `{}` para implementar parámetros con nombre. El siguiente gráfico es la última página de entrada de diario que creará.



1. Cree un nuevo archivo Dart en la carpeta de páginas . Haga clic derecho en la carpeta de páginas , seleccione Nuevo Archivo Dart, ingrese edit_entry.dart y haga clic en el botón Aceptar para guardar.

2. Importe la clase material.dart , la clase database.dart , el paquete intl.dart y el dardo: biblioteca matemática. Agregue una nueva línea y cree la clase EditEntry que extiende un StatefulWidget.

```
importar 'paquete: flutter/material.dart'; importar 'paquete: ch13_local_persistence/classes/database.dart'; importar 'paquete: intl/intl.dart'; // Formato de fechas import 'dart:math'; // Números al azar
```

```
clase EditEntry extiende StatefulWidget { @override
```

```
    _EditEntryState createState() => _EditEntryState(); }
```

```
clase _EditEntryState extiende Estado<EditEntry> {  
    @anular
```

```
    Complición del widget (contexto BuildContext) {  
        devolver Contenedor(;
```

```
    }}
```

3. Después de que la clase EditEntry extienda StatefulWidget { y antes de @override, agregue las tres variables bool add, int index y JournalEdit journalEdit y márquelas como finales.

```
clase EditEntry extiende StatefulWidget { final bool add;
    índice int final; final
    DiarioEditar diarioEditar;

    @anular
    _EditEntryState createState() => _EditEntryState(); }
```

4. Agregue el constructor EditEntry con Key key, this.add, this.index y this.journalEdit como parámetros nombrados encerrándolos entre corchetes {}.

```
clase EditEntry extiende StatefulWidget { final
    bool add; índice
    int final; final
    DiarioEditar diarioEditar;

    const EditEntry({Key key, this.add, this.index, this.journalEdit}) : super(key: key);

    @anular
    _EditEntryState createState() => _EditEntryState(); }
```

5. Modifique la clase _EditEntryState y agregue el JournalEdit privado _journalEdit, String _title y DateTime _selectedDate variables. Tenga en cuenta que la variable privada _journalEdit se completa con el valor de la clase JournalEdit pasado al constructor EditEntry .

```
clase _EditEntryState extiende Estado<EditEntry> {
    DiarioEditar _diarioEditar;
    Cadena _título;
    Fecha y hora _fecha seleccionada;

    @anular
    Compilación del widget (contexto BuildContext) {
        devolver Contenedor();

    }}
```

6. El estado de ánimo y la nota usan el widget TextField , que requiere TextEditingController para acceder y modificar los valores. Agregue las variables _moodController y _noteController TextEditingController e inicialícelas con el constructor TextEditingController() . Tenga en cuenta que este controlador trata un valor nulo como una cadena vacía.

```
clase _EditEntryState extiende Estado<EditEntry> {
    DiarioEditar _diarioEditar;
    Cadena _título;
    Fecha y hora _fecha seleccionada;
    TextEditingController _moodController = TextEditingController();
    TextEditingController _noteController = TextEditingController();}
```

```
        @anular  
        Compilación del widget (contexto BuildContext) {  
            devolver Contenedor();  
  
        }  
    }
```

7. Declare las variables `_moodFocus` y `_noteFocus` FocusNode e inicialícelas con el Constructor `FocusNode()`. Usará `FocusNode` con `TextInputAction` para personalizar el botón de acción del teclado en los pasos 30 y 31.

```
clase _EditEntryState extiende Estado<EditEntry> {  
    DiarioEditar _diarioEditar;  
    Cadena _título;  
    Fecha y hora _fecha seleccionada;  
    TextEditingController _moodController = TextEditingController();  
    TextEditingController _noteController = TextEditingController();  
    FocusNode _moodFocus = FocusNode();  
    FocusNode _noteFocus = FocusNode();  
  
    @anular  
    Compilación del widget (contexto BuildContext) {  
        devolver Contenedor();  
  
    }  
}
```

8. Anule `initState()` e inicialicemos las variables con valores pasados al constructor `EditEntry` y asegúrese de agregar `super.initState()`.

```
@anular  
void initState() {  
    super.initState();  
}
```

9. Inicialice la variable `_journalEdit` utilizando el constructor de la clase `JournalEdit` al establecer la acción de forma predeterminada en 'Cancelar' y el valor del diario en `widget.journalEdit.journal`. Tenga en cuenta que utiliza el widget para acceder a los valores del constructor `EditEntry`. Además, tenga en cuenta que accede a la entrada de diario individual desde la clase `JournalEdit` utilizando el operador de punto y luego eligiendo la variable de diario .

```
_journalEdit = JournalEdit(acción: 'Cancelar', diario: widget.journalEdit.journal);
```

10. Inicialice la variable `_title` utilizando un operador ternario para verificar si `widget.add` es verdadero. Si es así, establezca el valor en 'Aregar' y, si es falso, establezca el valor en 'Editar'. Al usar la variable `_title` , personaliza el título de la barra de aplicaciones según la acción que el usuario está realizando. Son los pequeños detalles los que hacen que una aplicación sea excelente.

```
_title = widget.agregar? 'Aregar' : 'Editar';
```

11. Inicialice la variable `_journalEdit.journal` desde la variable `widget.journalEdit.journal` .

```
_diarioEditar.diario = widget.diarioEditar.diario;
```

12. Para completar los campos de entrada en la página, agregue una declaración `if-else` . Si el valor de `widget.add` es verdadero, lo que significa agregar un nuevo registro de diario, entonces inicialice la variable `_selectedDate` con la fecha actual usando el constructor `DateTime.now()` e inicialice `_moodController.text`

y `_noteController.text` a una cadena vacía. Si el valor de `widget.add` es falso, lo que significa editar un registro de diario actual, entonces inicialice la variable `_selectedDate` con `_journal>Edit.journal.date` y use `DateTime.parse` para convertir la fecha de cadena a un formato de fecha y hora . También inicialice `_moodController.text` con `_journalEdit.journal.mood` y `_noteController.text` con `_journalEdit.journal.note`. Cuando anule el método `initState()` , asegúrese de iniciar el método con una llamada a `super.initState()`.

```
@anular
void initState() {
    super.initState();

    _diarioEditar = DiarioEditar(acción: 'Cancelar', diario: widget.
diarioEditar.diario); _title =
    widget.agregar? 'Aregar' : 'Editar'; _diarioEditar.diario
    = widget.diarioEditar.diario; if (widget.add) { _selectedDate =
    DateTime.now();
    _moodController.text = ""; _noteController.text
    = ""; } else { _selectedDate =
    DateTime.parse(_journalEdit.journal.date);

    _moodController.text = _journalEdit.journal.mood; _noteController.text =
    _journalEdit.journal.note; } }
```

13. Anule `dispose()`, y eliminemos los dos `TextEditingController` y `FocusNode`; asegúrese de agregar `super.dispose()`.
Cuando anule el método `dispose()` , asegúrese de finalizar el método con una llamada a `super.dispose()`.

```
@anular
disponer () {

    _moodController.dispose();
    _noteController.dispose();
    _moodFocus.dispose();
    _noteFocus.dispose();

    super.dispose();
}
```

14. Agregue el método asíncrono `_selectDate(DateTime selectedDate)` que devuelve un `Futuro<FechaHora>`. Este método es responsable de llamar al `showDatePicker()` incorporado de Flutter que presenta al usuario un cuadro de diálogo emergente que muestra un calendario de Material Design para elegir fechas.

```
// Selector de fechas
Future<DateTime> _selectDate(DateTime selectedDate) asíncrono {

}
```

15. Agregue la variable `DateTime _initialDate` e inicialícela con la variable `selectedDate` pasada en el constructor.

```
Fecha y hora _fechainicial = fechaseleccionada;
```

16. Agregue una variable final DateTime _pickedDate (la fecha que el usuario elige del calendario) y inicialícelo llamando al constructor await showDatePicker(). Pase los argumentos context, initialDate, firstDate y lastDate . Tenga en cuenta que para la primera fecha usa la fecha de hoy y resta 365 días, y para la última fecha, agrega 365 días, lo que le indica al calendario un rango de fechas seleccionable.

```
fecha y hora final _pickedDate = esperar showDatePicker (
    context: context,
    initialDate: _initialDate, firstDate:
    DateTime.now().subtract(Duration(days: 365)), lastDate:
    DateTime.now().add(Duration(days: 365)), );
```

17. Agregue una declaración if que verifique que _pickedDate (la fecha que el usuario seleccionó del calendario) no es igual a nulo, lo que significa que el usuario tocó el botón Cancelar del calendario. Si el usuario eligió una fecha, modifique la variable de fecha seleccionada utilizando el constructor DateTime() y pase el año, mes y día de _pickedDate. Para la hora, pase _initialDate hora, minuto, segundo, milisegundo y microsegundo. Tenga en cuenta que dado que solo está cambiando la fecha y no la hora, usa la fecha y hora de creación del original.

```
if (_fechaelegida != nulo)
{ fechaseleccionada =
    FechaHora( _fechaelegida.año,
    _fechaelegida.mes,
    _fechaelegida.día,
    _fechainicial.hora,
    _fechainicial.minuto,
    _fechainicial.segundo,
    _fechainicial.milisegundo, _fechainicial.microsegundo);
}
```

18. Agregue una declaración de devolución para devolver la fecha seleccionada.

```
return fechaseleccionada;
```

El siguiente es el método _selectDate() completo:

```
// Selector de fechas
Future<DateTime> _selectDate(DateTime selectedDate) asincrono {
    Fecha y hora _fechainicial = fechaseleccionada;

    fecha y hora final _pickedDate = esperar showDatePicker (
        context: context,
        initialDate: _initialDate, firstDate:
        DateTime.now().subtract(Duration(days: 365)), lastDate:
        DateTime.now().add(Duration(days: 365)), ); if (_fechaelegida!= nulo)

    { fechaseleccionada =
        fechahora(_fechaelegida.año,
        _fechaelegida.mes,
        _fechaelegida.día,
        _fechainicial.hora,
        _fechainicial.minuto,
```

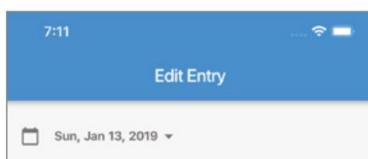
```
        _fechainicial.segundo,  
        _fechainicial.milisegundo,  
        _fechainicial.microsegundo);  
  
    } return fechaseleccionada;  
}
```

19. En el método Widget build() , reemplace Container() con los widgets UI Scaffold y AppBar, y para la propiedad del cuerpo agregue SafeArea() y SingleChildScrollView() con la propiedad secundaria como Column(). Tenga en cuenta que el título de AppBar usa el widget de texto con la variable _title para personalizar el título con Agregar o Editar entrada.

```
@anular  
Compilación del widget (contexto BuildContext) {}></line> return  
Scaffold()  
    appBar: AppBar(title:  
        Text('$_title Entry'), automaticallyLeading:  
        false, ), cuerpo: SafeArea(  
  
        hijo: SingleChildScrollView(  
            relleno: EdgeInsets.all(16.0), child:  
            Column( children:  
                <Widget>[  
  
                    ], ), ), ), );  
    }
```

20. Agregue a los hijos de la columna un widget FlatButton que se usa para mostrar la fecha seleccionada con formato, y cuando el usuario toca el botón, presenta el calendario. Establezca la propiedad de relleno FlatButton en EdgeInsets.all(0.0) para eliminar el relleno para una mejor estética y agregue a la propiedad secundaria un widget Row() .

```
FlatButton( relleno:  
    EdgeInsets.all(0.0), child: Row( children:  
        <Widget>[ ], ), ),
```



21. Agregue a la propiedad Row children el ícono Icons.calendar_day con un tamaño de 22.0 y un color propiedad establecida en Colors.black54.

```
ícono  
(Iconos.calendar_today, tamaño:  
22.0, color:  
Colors.black54, ),
```

22. Agregue un SizedBox con una propiedad de ancho establecida en 16.0 para agregar un espaciador.

```
SizedBox (ancho: 16.0,),
```

23. Agregue un widget de texto y formatee _selectedDate con el constructor DateFormat.yMMEd() .

```
Texto(DateFormat.yMMEd().format(_selectedDate),  
estilo: TextStyle (color:  
Colors.black54, fontWeight:  
FontWeight.bold),  
,)
```

24. Agregue el ícono Icons.arrow_drop_down con la propiedad de color establecida en Colors.black54.

```
ícono  
(Iconos.arrow_drop_down, color:  
Colors.black54, ),
```

25. Agregue la devolución de llamada onPressed() y márquela asíncrona ya que llamar al calendario es un evento futuro .

```
onPressed: () asíncrono {},
```

26. Agregue a onPressed() la llamada al método FocusScope.of().requestFocus() para descartar la clave tablero si alguno de los widgets de TextField tiene foco. (Este paso es opcional, pero quería mostrarle cómo se logra).

```
FocusScope.of(contexto).requestFocus(FocusNode());
```

27. Agregue una variable DateTime _pickerDate inicializada llamando al método futuro await _selectDate(_select edDate) , razón por la cual agrega la palabra clave await . Agregó este método en el paso 14.

28. Agregue setState() y dentro de la llamada modifique la variable _selectedDate con _pickerDate valor, que es la fecha seleccionada del calendario.

```
DateTime _pickerDate = esperar _selectDate(_selectedDate); setState(() { _selectedDate  
= _pickerDate; });
```

El siguiente es el código completo del widget FlatButton :

```
botón plano (  
relleno: EdgeInsets.all(0.0), hijo: Fila(
```

niños:

```
<Widget>[ Icon( Icons.calendar_today,
    size: 22.0, color:
    Colors.black54, ), SizedBox(ancho:
    16.0,),
    Text(DateTimeFormat.yMMEd().format(_selectedDate),
        estilo: TextStyle (color:
        Colors.black54, fontWeight:
        FontWeight.bold),
    ),
    Icono
    (Iconos.arrow_drop_down, color:
    Colors.black54, ], ), onPressed:
    ()>
asíncrono {
    FocusScope.of(contexto).requestFocus(FocusNode()); DateTime
    _pickerDate = esperar _selectDate(_selectedDate); setState(() { _selectedDate =
    _pickerDate; });
}, ),
```

29. Ahora es el momento de agregar los dos widgets de `TextField` para los campos de estado de ánimo y notas. ¿Cómo se establece qué `TextField` pertenece al estado de ánimo o nota? Es el controlador, por supuesto.

Para el campo de texto de estado de ánimo, configure el controlador en `_moodController` y configure el enfoque automático en verdadero para configurar automáticamente el enfoque y mostrar el teclado cuando se abre la página.

```
TextField
    (controlador: _moodController,
    enfoque automático:
    verdadero,),
```

30. Establezca `textInputAction` en `TextInputAction.next` diciéndole al botón de acción del teclado que se mueva a el siguiente campo.

```
AcciónEntradaTexto: AcciónEntradaTexto.next,
```

31. Establezca `focusNode` en `_moodFocus` y `textCapitalization` en `TextCapitalization.words`, lo que significa que cada palabra se escribe automáticamente en mayúscula.

```
focusNode: _moodFocus,
textCapitalization: TextCapitalization.words,
```

32. Establezca la decoración en `InputDecoration` con `labelText` establecido en 'Mood' y el ícono establecido en `Iconos.estado de ánimo`.

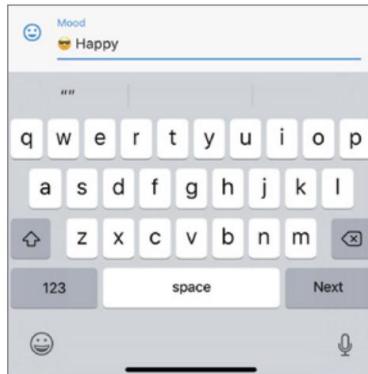
```
decoración: InputDecoration( labelText:
    'Mood', icon:
    Icon(Icons.mood),
),
```

33. Para la propiedad `onSubmitted`, ingrese el nombre del argumento tal como se envió y llame a `FocusScope.of(context).requestFocus(_noteFocus)` para que el botón de acción del teclado cambie el enfoque a la nota `TextField`. Tenga en cuenta que nombré el argumento presentado, pero puede ser cualquier nombre, como valor enviado o valor de estado de ánimo.

```
onSubmitted: (enviado)
{ FocusScope.of(context).requestFocus(_noteFocus); },
```

El siguiente es el widget `TextField` de estado de ánimo completo:

```
TextField( controlador: _moodController,
    enfoque automático:
        verdadero, textInputAction: TextInputAction.next,
    focusNode: _moodFocus,
    textCapitalization: TextCapitalization.words, decoración:
    InputDecoration( labelText: 'Mood',
        icon: Icon(Icons.mood),
    ),
    onSubmitted: (enviado)
    { FocusScope.of(context).requestFocus(_noteFocus); }, ),
```



34. Para la nota `TextField`, configure el controlador en `_noteController` y configure el enfoque automático en verdadero para configurar automáticamente el enfoque y mostrar el teclado cuando se abre la página.

```
Campo de
texto (controlador: _noteController,),
```

35. Establezca `textInputAction` en `TextInputAction.newline` diciéndole al botón de acción del teclado que inserte una nueva línea en el `TextField`.

```
acción de entrada de texto: acción de entrada de texto. nueva línea,
```

36. Establezca `focusNode` en `_noteFocus` y `textCapitalization` en `TextCapitalization.sentences`, lo que significa que cada primera palabra de una oración se escribe automáticamente en mayúsculas.

```
focusNode: _noteFocus,  
textCapitalization: TextCapitalization.sentences,
```

37. Establezca la decoración en `InputDecoration` con `labelText` establecido en 'Nota' y el ícono establecido en `Iconos.sujeto`.

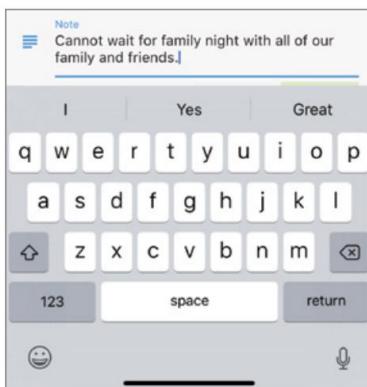
```
decoration: InputDecoration( labelText:  
'Nota', icon:  
Icon(Icons.subject), ),
```

38. Establezca la propiedad `maxLines` en nulo, lo que permite que `TextField` crezca verticalmente para mostrar todo el contenido de la nota. Usar esta técnica es una excelente manera de hacer que un `TextField` crezca automáticamente al tamaño del contenido sin escribir ninguna lógica de código.

```
maxLines: null,
```

El siguiente es el widget `TextField` de la nota completa:

```
Campo de  
texto (controlador: _noteController,  
  
textInputAction: TextInputAction.newline, focusNode:  
_noteFocus, textCapitalization:  
TextCapitalization.sentences, decoration: InputDecoration( labelText:  
'Note', icon: Icon(Icons.subject), ),  
maxLines: null, ),
```



39. La última parte de la página de entrada es agregar los botones Cancelar y Guardar. Agregar una fila y establecer `mainAxisAlignment` a `MainAxisAlignment.end` para alinear los botones al lado derecho de la página.

```
Fila( AlineaciónEjePrincipal: AlineaciónEjePrincipal.end,
```

```
hijos: <Widget>[ ], ),
```

40. Edite los hijos de la Fila y agregue un FlatButton con el hijo establecido en un widget de Texto para mostrar 'Cancelar'. Establezca la propiedad de color en Colors.grey.shade100, haciendo que el botón no sea el foco de acción principal.

```
FlatButton (hijo:  
    Texto ('Cancelar'), color:  
        Colors.grey.shade100,),
```

41. Para la propiedad onPressed , modifique _journalEdit.action a 'Cancel' y llame al Navigator.pop(context, _journalEdit) para descartar el formulario de entrada y devolver el valor a la página de llamada. Manejará esta acción en la última prueba de este capítulo ("Terminar la página de inicio del diario").

```
FlatButton (hijo:  
    Texto ('Cancelar'), color:  
        Colors.grey.shade100, onPressed: () {  
  
        _journalEdit.action = 'Cancelar';  
        Navigator.pop(contexto, _journalEdit); }, ),
```

42. Agregue un SizedBox con el ancho establecido en 8.0 para colocar un espaciador entre los dos botones.

```
SizedBox (ancho: 8.0),
```

43. Agregue el segundo FlatButton con el conjunto secundario a un widget de Texto para mostrar 'Guardar'. Establezca la propiedad de color en Colors.lightGreen.shade100, haciendo que el botón sea el foco de acción principal.

```
FlatButton (hijo:  
    Texto ('Guardar'), color:  
        Colors.lightGreen.shade100,),
```

44. Para la propiedad onPressed , modifique _journalEdit.action a 'Save'.

```
onPressed: () {  
    _journalEdit.action = 'Guardar'; },
```

45. Ya que está guardando la entrada, declare una variable String _id y use el operador ternario para verificar que la variable widget.add esté establecida en verdadero y use Random().nextInt(9999999) para generar un número aleatorio. Si widget.add es falso, use el _journalEdit.journal.id actual ya que está editando una entrada existente.

Tenga en cuenta que nextInt() establece el rango de número máximo desde cero y, en nuestro caso, establece el máximo en 9999999. Tenga en cuenta que para nuestros propósitos esto funciona muy bien, pero en un entorno de producción, le sugiero que use un UUID, que es un número de 128 bits que incluye caracteres alfanuméricos. Un UUID de muestra se ve así: 409fg342-h34c-25c8-b311-51874523574e.

```
Cadena _id = widget.add? Random().nextInt(9999999).toString() : _journalEdit.journal.id;
```

46. Modifique el valor de `_journalEdit.journal` usando el constructor de clase `Journal()` y pase la propiedad `id` con la variable `_id`, la fecha con `_selectedDate.toString()` (la fecha se guarda como una cadena), el estado de ánimo con `_moodController.text` y la nota con `_noteController.text`.

```
_journalEdit.journal = Journal( id: _id,
    date:
        _selectedDate.toString(), mood:
        _moodController.text, note:
        _noteController.text, );
```

47. Llame a `Navigator.pop(context, _journalEdit)` para descartar el formulario de entrada y devolver el valor a la página de llamada. Maneja recibir esta acción en el ejercicio "Finalizar la página de inicio del diario".

```
FlatButton
    (hijo: Texto ('Guardar'),
    color: Colors.lightGreen.shade100,
    onPressed: () {
        _journalEdit.action = 'Guardar';
        Cadena _id = widget.add? Random().nextInt(9999999).toString() : _journalEdit.journal.id;

    _journalEdit.journal = Journal( id: _id, date:

        _selectedDate.toString(), mood:
        _moodController.text, note:
        _noteController.text, );

    Navigator.pop(contexto, _journalEdit); }, ),
```

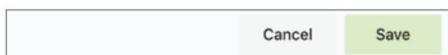
El siguiente es el código completo del widget de Fila :

```
Row( mainAxisAlignment: MainAxisAlignment.end,
children:

<Widget>[ FlatButton( child:
    Text('Cancel'), color:
    Colors.grey.shade100, onPressed: () {
        _journalEdit.action = 'Cancelar';
        Navigator.pop(contexto, _journalEdit); }, ),

SizedBox (ancho: 8,0),
FlatButton
    (hijo: Texto ('Guardar'),
    color: Colors.lightGreen.shade100,
    onPressed: () {
        _journalEdit.action = 'Guardar';
        Cadena _id = widget.add? Random().nextInt(9999999).toString() :
        _journalEdit.journal.id;
        _diarioEditar.diario = Diario(
```

```
        id: _id,  
        fecha: _selectedDate.toString(),  
        estado de ánimo:  
        _moodController.text, nota:  
  
        _noteController.text, ); Navigator.pop(contexto,  
  
        _journalEdit); }, ), ], ),
```



CÓMO FUNCIONA

Creé el archivo `edit_entry.dart` con la clase `EditEntry` extendiendo un `StatefulWidget` para manejar la adición y edición de entradas de diario. Personalizó el constructor para que tuviera los tres argumentos `add`, `index` y `journalEdit`. La variable `add` es responsable de manejar si está agregando o modificando una entrada; el índice es -1 si agrega un registro o es la ubicación real del índice de lista si está editando una entrada. La variable `journalEdit` tiene un valor de acción para 'Cancelar' o 'Guardar' y la clase `Journal` contiene los valores de entrada de diario para los valores de `id`, `fecha`, `estado de ánimo` y `nota`.

La función `showDatePicker()` muestra un cuadro de diálogo emergente que contiene el selector de fecha de Material Design. Pase los argumentos `context`, `initialDate`, `firstDate` y `lastDate` para personalizar el intervalo de fechas seleccionable.

Para dar formato a las fechas, utilice el constructor con nombre `DateFormat` apropiado. Para personalizar aún más el formato de fecha, puede usar los métodos `add_*()` (sustituir el carácter * con los caracteres de formato necesarios) para agregar y combinar múltiples formatos.

El `TextEditingController` permite el acceso al valor del widget `TextField` asociado. `TextField` `TextInputAction` le permite personalizar el botón de acción del teclado del dispositivo. El nodo de enfoque está asociado al widget `TextField`, que le permite establecer el enfoque en el campo de texto adecuado mediante programación. `TextCapitalization` le permite configurar el uso de mayúsculas del widget `TextField` mediante el uso de palabras, oraciones y caracteres.

La clase `JournalEdit` realiza un seguimiento de la acción y los valores de entrada. Utiliza la variable de acción para rastrear si se toca el botón Guardar o Cancelar. Utiliza la variable de diario para contener los valores de campo de la clase `Diario` para editar o crear nuevas entradas de diario. El método `Navigator.pop()` devuelve los valores de la clase `JournalEdit` a la página de inicio.

Finalización de la página de inicio del diario

La página de inicio es responsable de mostrar una lista de entradas de diario. En el Capítulo 9, "Creación de listas de desplazamiento y efectos", aprendió a usar `ListView.builder`, pero para esta aplicación, aprenderá a usar `ListView.separated` constructor. Al usar el constructor separado, tiene los mismos beneficios que el constructor constructor porque los constructores se llaman solo para los niños que

son visibles en la página. Es posible que haya notado que dije constructores, porque usa dos de ellos, el itemBuilder estándar para la lista de elementos secundarios (entradas de diario) y el separatorBuilder para mostrar un separador entre elementos secundarios. El separatorBuilder es extremadamente poderoso para personalizar el separador; podría ser una imagen, un ícono o un widget personalizado, pero para nuestros propósitos, usará un widget de divisor . Utilizará ListTile para dar formato a su lista de entradas de diario y personalizar la propiedad principal con una columna para mostrar la fecha y el día de la semana, lo que facilita la detección de entradas individuales (Figura 13.5).

Para eliminar entradas del diario, utilizará un Desechable, que aprendió en el Capítulo 11, "Aplicación de la interactividad". Tenga en cuenta que para que Desechable funcione correctamente y elimine la entrada de diario correcta, establecerá la propiedad clave en el campo de identificación de la entrada de diario usando la clase Clave , que toma un valor de Cadena como Clave(instantánea.datos[índice] .identificación).

Aprenderá a usar el widget FutureBuilder , que funciona con un futuro para recuperar los datos más recientes sin bloquear la interfaz de usuario. Aprendió los detalles en la sección "Recuperación de datos con FutureBuilder" de este capítulo .

Usará un futuro que aprendió a usar en los capítulos 3 y 12 ("Aprender los conceptos básicos de Dart" y "Escribir código nativo de la plataforma") para recuperar las entradas del diario. La recuperación de las entradas del diario requiere varios pasos y, para ayudarlo a administrarlas, utilizará las clases del archivo database.dart que creó en la sección "Adición de las clases de la base de datos del diario" de este capítulo. Las clases que utiliza son DatabaseFileRoutines, Database, Journal y JournalEdit.

1. Realiza las llamadas DatabaseFileRoutines para leer el archivo JSON ubicado en la carpeta de documentos del dispositivo.
2. Llama a la clase de base de datos para analizar el JSON en un formato de lista .
3. Utilice la función Ordenar lista para ordenar las entradas por fecha DESC.
4. La lista ordenada se devuelve a FutureBuilder y ListView muestra el día existente. entradas finales.

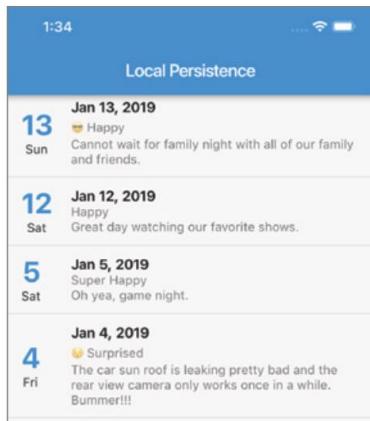


FIGURA 13.5: Lista de entradas de diario

PRUÉBALO Finalización de la página de inicio del diario

En esta sección, terminará la página de inicio agregando métodos para cargar, agregar, modificar y guardar entradas de diario. Creará un método para construir y usar el constructor ListView.separated para personalizar su lista de entradas de diario. En ListView itemBuilder, agregará un Descartable para manejar la eliminación de entradas de diario.

Importará el archivo database.dart para utilizar las clases de la base de datos para ayudarlo a serializar los objetos JSON.

1. Importe los paquetes edit_entry.dart, database.dart e intl.dart .

```
importar 'paquete: flutter/material.dart'; importar
'paquete: ch13_local_persistence/edit_entry.dart';
```

```
importar 'paquete: ch13_local_persistence/classes/database.dart'; importar  
'paquete: intl/intl.dart'; // Formatear fechas
```

2. Después de que la clase _HomeState extienda State<Home> {}, agregue la variable Database _database .

La variable _database contiene los objetos JSON analizados del objeto JSON del diario , que es su lista de entradas del diario.

```
class _HomeState extiende State<Home> {  
  Base de datos _base de datos;
```

3. Agregue el método asíncrono _loadJournals() que devuelve un Future<List<Journal>>, que es un Lista de las entradas de la clase Diario .

```
Future<List<Journal>> _loadJournals() asíncrono { }
```

4. Agregue la llamada await DatabaseFileRoutines().readJournals() y agregue con la notación de puntos un llame a entonces ((journalsJson) {}).

¿Qué es exactamente esta llamada a then()? Registra una devolución de llamada que se llamará cuando se complete Future . Lo que esto significa es que una vez que el método readJournals() se completa y devuelve el valor, then() ejecuta el código interno. Tenga en cuenta que el parámetro journalsJson recibe el valor de los objetos JSON leídos del archivo guardado local_persistence.json ubicado en la carpeta de documentos locales del dispositivo.

```
await DatabaseFileRoutines().readJournals().then((journalsJson) {});
```

5. Dentro de la devolución de llamada then() , modifique la variable _database con el valor de la llamada a databases eFromJson(revistasJson).

El método databaseFromJson en la clase database.dart usa json.decode() para analizar los objetos JSON que se leen del archivo guardado. Se llama a Database.fromJson() y devuelve los objetos JSON como una lista Dart, que es extremadamente poderosa. En este punto, está claro cómo separar la lógica del código para manejar sus datos en las clases de la base de datos se vuelve útil y sencillo.

```
_base de datos = base de datos de Json (journalsJson);
```

6. Continúe dentro de la devolución de llamada then() , y ordenemos las entradas del diario por fecha DESC, con las entradas más nuevas primero y las más antiguas al final. Use _database.journal.sort() para comparar fechas y ordenarlas.

```
_database.journal.sort((comp1, comp2) => comp2.date.compareTo(comp1.date));
```

7. Después de la devolución de llamada de then() , agregue una nueva línea con la declaración de devolución que devuelve la variable _database.journal, que contiene las entradas de diario ordenadas.

```
volver _base de datos.diario;
```

El siguiente es el método _loadJournals() completo:

```
Future<List<Journal>> _loadJournals() async { aguardar  
  DatabaseFileRoutines().readJournals().then((journalsJson) {  
    _base de datos = base de datos de Json (journalsJson);  
    _database.journal.sort((comp1, comp2) => comp2.date.compareTo(comp1.date));}); volver _base  
    de  
    datos.diario;  
}
```

8. Agregue el método `_addOrEditJournal()` que maneja la presentación de la página de entrada de edición para agregar o modificar una entrada de diario. Utiliza `Navigator.push()` para presentar la página de entrada y esperar el resultado de las acciones del usuario. Si el usuario presionó el botón Cancelar, no sucede nada, pero si presionó Guardar, entonces agrega la nueva entrada de diario o guarda los cambios en la entrada.

`_addOrEditJournal()` es un método asíncrono que toma los parámetros con nombre de `bool add, int index y Journal journal`. Consulte la Tabla 13.4 para ver la descripción de los argumentos. Inicie la variable `JournalEdit _journalEdit` con la clase `JournalEdit` con el valor de acción establecido en una cadena vacía y el valor de diario establecido en la variable de diario que se pasa desde el constructor.

```
void _addOrEditJournal({bool add, int index, Journal journal}) async {
    JournalEdit _journalEdit = JournalEdit(acción: "", diario: diario);
}
```

9. Agregue una nueva línea; va a usar el Navegador para pasar los valores del constructor a la página de entrada de edición usando la palabra clave `await` que pasa el valor a la variable local `_journalEdit`.

Para el constructor `MaterialPageRoute`, pase los valores del constructor a la clase `EditEntry()` y establezca la propiedad `fullscreenDialog` en `true`.

```
_journalEdit = espera Navigator.push( contexto,
    MaterialPageRoute( builder:
        (contexto) => EditEntry( add: add, index: index,
            journalEdit:
                _journalEdit, ),
        fullscreenDialog: true ),
```

);

Una vez que se descarta la página de entrada de edición, la declaración de cambio se ejecuta a continuación y tomará las medidas adecuadas según la selección del usuario. La sentencia `switch` evalúa la acción `_journalEdit.action` para verificar si se presionó el botón Guardar y luego verifica si está agregando o guardando la entrada con una sentencia `if-else`.

```
cambiar (_journalEdit.action) {
```

10. Agregue la primera declaración de cambio de caso que verifica el valor 'Guardar'.

Si la variable `add` se establece en `true`, lo que significa que está agregando una nueva entrada, entonces usa `setState()` y llama a `_database.journal.add(_journalEdit.journal)` pasando los valores del diario.

```
cambiar (_journalEdit.action) {
    caso 'Guardar':
        if (añadir)
            { setState(() {
                {_database.journal.add(_journalEdit.journal);});}

    } romper;
}
```

11. Si la variable add se establece en false, lo que significa que está guardando una entrada existente, entonces use setState () y modifique el valor de database.journal[index] = journalEdit.journal.

Está reemplazando los valores de la entrada de diario seleccionada `_database.journal[index]` actual por el valor de índice y reemplazándolo con el valor de `_journalEdit.journal` pasado desde la página de edición de entrada.

```
cambiar (_journalEdit.action) {  
    caso 'Guardar':  
        if (añadir)  
            { setState(()  
                { _database.journal.add(_journalEdit.journal); }); } else { setState(()  
  
                { _database.journal[index]  
                    = _journalEdit.journal;}); } romper;  
    }  
}
```

12. Para guardar los valores de entrada de diario en el directorio de documentos de almacenamiento local del dispositivo, llame `DatabaseFileRoutines().writeJournals(databaseToJson(_database))`. Agregue la segunda declaración de caso que verifica el valor 'Cancelar' , pero no es necesario agregar ninguna acción ya que el usuario canceló la edición.

```
cambiar (_journalEdit.action) {
    caso 'Guardar':
        if (añadir)
            { setState(() {
                { _database.journal.add(_journalEdit.journal); });
            } else {
                setState(() {
                    { _database.journal[index]
                        = _journalEdit.journal;
                    });
                }
            }
        }
    }

DatabaseFileRoutines().writeJournals(databaseToJson(_database));
rompido;
descanso;
```

13. Agregue la verificación predeterminada en caso de que suceda algo más, pero tampoco toma ningún pelaje otra acción.

```
cambiar (_journalEdit.action) {
    caso 'Guardar':
        if (añadir)
            { setState(() {
                { _database.journal.add(_journalEdit.journal); });
            } else { setState(() {
                { _database.journal[index]
                    = _journalEdit.journal; });
            } }
}
```

```
DatabaseFileRoutines().writeJournals(databaseToJson(_database)); romper; caso 'Cancelar':  
descanso;  
predeterminado:  
descanso;  
  
}
```

El siguiente es el método `_addOrEditJournal` completo:

```
void _addOrEditJournal({bool add, int index, Journal journal}) async {  
    JournalEdit _journalEdit = JournalEdit(acción: "", diario: diario); _journalEdit = esperar Navigator.push(  
        contexto,  
        MaterialPageRoute( builder:  
            (context) => EditEntry( add: add, index: index,  
                journalEdit:  
                _journalEdit, ),  
                fullscreenDialog: true ),  
  
    );  
    cambiar (_journalEdit.action) {  
        caso 'Guardar':  
            if (añadir)  
            { setState(()  
                { _database.journal.add(_journalEdit.journal); }); } más  
  
            { establecerEstado(() {  
                _database.journal[index] = _journalEdit.journal; });  
  
            }  
    DatabaseFileRoutines().writeJournals(databaseToJson(_database)); romper;  
  
        caso 'Cancelar':  
            romper;  
        por defecto:  
            romper;  
    }  
}
```

14. Agregue el método `_buildListViewSeparated(AsyncSnapshot snapshot)` que toma la Parámetro `AsyncSnapshot`, que es la Lista de entradas de diario. El método se llama desde `FutureBuilder()` en la propiedad del cuerpo . La forma de leer la lista de entradas de diario es accediendo a la propiedad de datos de la instantánea mediante `snapshot.data`. Se accede a cada entrada de diario usando el índice como `snapshot.data[index]`. Para acceder a cada campo, utilice `snapshot.data[index].date` o `snapshot.data[index].mood` y así sucesivamente.

```
Widget _buildListViewSeparated (instantánea AsyncSnapshot) { }
```

15. El método devuelve un ListView usando el constructor separated() . Establecer el número de elementos propiedad en snapshot.data.length y establezca la propiedad itemBuilder en (contexto BuildContext, índice int).

```
Widget _buildListViewSeparated(AsyncSnapshot snapshot) { return
    ListView.separated( itemCount:
        snapshot.data.length, itemBuilder: (BuildContext
        context, int index) { }, );
```

}

16. Dentro de itemBuilder, inicialice String _titleDate con DateFormat.yMMMd() constructor usando format() con snapshot.data[index].date. Dado que los datos están en formato de cadena , utilice el constructor DateTime.parse() para convertirlos en una fecha.

```
String _titleDate = DateFormat.yMMMd().format(DateTime.parse(snapshot .data[index].date));
```

17. Inicialice String _subtitle con los campos de estado de ánimo y nota mediante la concatenación de cadenas y sepárelos con una línea en blanco mediante el carácter '\n' .

```
String _subtitle = snapshot.data[index].mood + "\n" + snapshot.data[index].note;
```

18. Agregue return Dismissible() y lo completará en el paso 20.

```
volver Desechable();
```

19. Agregue el separadorBuilder que maneja la línea de separación entre las entradas del diario usando un Divider() con la propiedad de color establecida en Colors.grey.

```
Widget _buildListViewSeparated(AsyncSnapshot snapshot) { return
    ListView.separated( itemCount:
        snapshot.data.length, itemBuilder: (BuildContext
        context, int index) { String _titleDate =
            DateFormat.yMMMd().format(DateTime.parse(snapshot .data[index ].fecha));

            String _subtitle = snapshot.data[index].mood + "\n" + snapshot
            .datos[índice].nota;
            volver Desechable(); },
```

```
separatorBuilder: (contexto BuildContext, int index) { return Divider( color:
            Colors.grey, ); }, );
```

```
}
```

20. Finalice el widget Dismissible() , que es responsable de eliminar las entradas del diario deslizando el dedo hacia la izquierda o hacia la derecha en la entrada misma. Establezca la propiedad clave en Clave (instantánea.datos[índice].id), que crea una clave a partir del campo de identificación de la entrada del diario .

```
return Desechable( clave:
    Clave(instantánea.datos[índice].id, );
```

21. La propiedad de fondo se muestra cuando el usuario desliza el dedo de izquierda a derecha y el

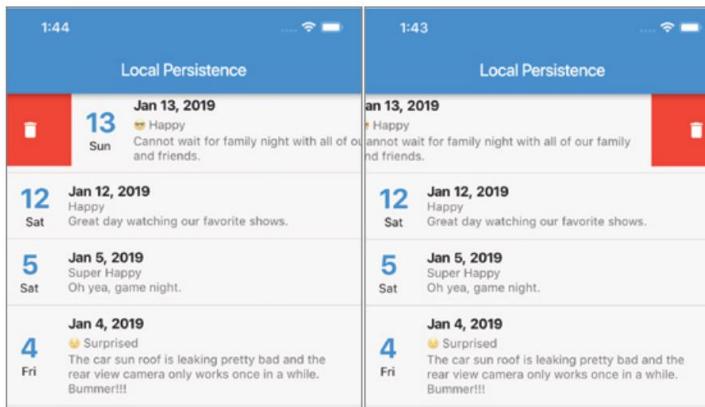
La propiedad de fondo secundario se muestra cuando el usuario desliza el dedo de derecha a izquierda. Para la propiedad de fondo , agregue un contenedor con el color establecido en Colors.red, la alineación establecida en Alignment.centerLeft, el relleno establecido en EdgeInsets.only (izquierda: 16.0) y la propiedad secundaria establecida en Icons.delete con el color propiedad establecida en Colors.white.

```
return Dismissible( clave:
    Clave(instantánea.datos[indice].id), fondo:
    Contenedor( color: Colores.rojo,
        alineación:
            Alignment.centerLeft, relleno:
                EdgeInsets.only(izquierda: 16.0), hijo: Icono(Icons.
                    eliminar, color:
                        Colors.white, ),
```

),);

22. Para el fondo secundario, use las mismas propiedades que el fondo pero cambie el

propiedad de alineación a Alignment.centerRight.



```
return Dismissible( clave:
    Clave(instantánea.datos[indice].id), fondo:
    Contenedor( color: Colores.rojo,
        alineación:
            Alignment.centerLeft, relleno:
                EdgeInsets.only(izquierda: 16.0), hijo: Icono(Icons.
                    eliminar, color:
                        Colors.white, ),
```

),
fondo secundario: Contenedor (color:
Colors.red, alineación:
Alignment.centerRight, padding: EdgeInsets.only
(right: 16.0), child: Icon(

```

Iconos.eliminar,
color: Colores.blanco, ),

),
child: ListTile(), onDismissed:
(dirección) { setState(() {

{ _database.journal.removeAt(index); });

DatabaseFileRoutines().writeJournals(databaseToJson(_database)); }, );

```

23. Termina el widget `ListTile()` , que es responsable de mostrar cada entrada del diario. Eres va a personalizar la propiedad principal para mostrar el día de la fecha y la descripción del día de la semana. Para la propiedad principal , agregue una columna () con la lista de elementos secundarios de dos widgets de texto .

```

hijo: ListTile(
principal: Columna (hijos:
<Widget>[
    Texto(),
    Texto(),
    ],
),
),

```



24. El primer widget de texto muestra el día; vamos a formatearlo con el constructor `DateFormat.d()` usando `format()` con `snapshot.data[index].date`. Dado que los datos están en formato de cadena , utilice el constructor `DateTime.parse()` para convertirlos en una fecha.

```
Texto(DateFormat.d().format(DateTime.parse(instantánea.datos[indice].fecha)), ),
```

25. Establezca la propiedad de estilo en `TextStyle` con un `fontWeight` de `FontWeight.bold`, `fontSize` de 32.0, y el color establecido en `Colors.blue`.

```

Texto(DateFormat.d().format(DateTime.parse(instantánea.datos[indice].fecha)),
estilo: TextStyle(
(fontWeight: FontWeight.bold, fontSize:
32.0, color: Colors.blue),
),
)

```

26. El segundo widget de texto muestra el día de la semana; vamos a formatearlo con el constructor DateFormat.E() usando format() con snapshot.data[index].date.

Dado que los datos están en formato de cadena , utilice el constructor DateTime.parse() para convertirlos en una fecha.

```
Texto(DateTime.parse(snapshot.data[index].date).format(DateFormat.E()))),
```

27. Establezca la propiedad de título en un widget de texto con la variable _titleDate y la propiedad de estilo en TextStyle con fontWeight establecido en FontWeight.bold.

```
texto del título(  
    _titleDate, estilo:  
        TextStyle(fontWeight: FontWeight.bold), ),
```

28. Establezca la propiedad de subtítulos en un widget de texto con la variable _subtitle .

```
subtítulo: Texto (_subtitle),
```

29. Agregue la propiedad onTap que llama al método _addOrEditJournal() y pase la propiedad add como falsa, lo que significa que no agrega una nueva entrada sino que modifica la entrada actual. Establezca la propiedad index en index, que es el índice de entrada actual en la lista.

Establezca la propiedad del diario en snapshot.data[index], que es la clase Diario con los detalles de la entrada que contienen los campos de identificación, fecha, estado

```
de ánimo y  
nota . onTap: ()
```

```
{ _addOrEditJournal( add: false, index: index, journal: snapshot.data[index],  
);},
```

El siguiente es el widget ListTile completo:

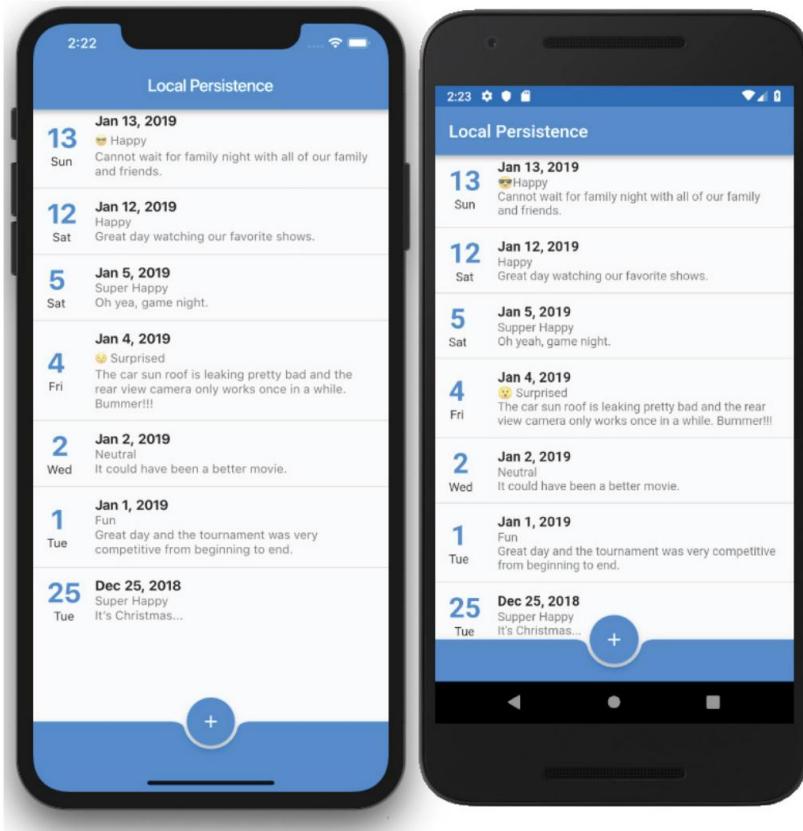
```
hijo: ListTile(  
    inicial: Column (hijos:  
  
        <Widget>[ Text(DateTime.parse(snapshot.data[index].date).format(DateFormat.E())),  
            estilo: TextStyle  
                (fontWeight: FontWeight.bold, fontSize:  
                    32.0, color: Colors.blue),  
  
        ),  
        Texto(DateTime.parse(snapshot.data[index].date))), ],  
  
    ),  
    texto del título(  
        _titleDate, estilo:  
            TextStyle(fontWeight: FontWeight.bold), ), subtítulo: Texto(_subtitle),  
  
    onTap: () { _addOrEditJournal( add:  
        false, index:  
            index,
```

```
diario: instantánea.datos[índice],  
); }, ),
```

El siguiente es el método `_buildListViewSeparated()` completo :

```
// Cree ListView con el widget Separator  
_buildListViewSeparated(AsyncSnapshot snapshot) { return  
    ListView.separated(itemCount:  
        snapshot.data.length, itemBuilder: (BuildContext  
        context, int index) { String _titleDate = DateFormat.yMMMd().format(DateTime.  
        analizar(instantánea  
.datos[índice].fecha));  
        String _subtitle = snapshot.data[index].mood + "\n" + snapshot  
.datos[índice].nota;  
        return Dismissible( clave:  
            Clave(instantánea.datos[índice].id), fondo:  
            Contenedor( color: Colores.rojo,  
                alineación:  
                Alignment.centerLeft, relleno:  
                EdgeInsets.only(izquierda: 16.0), hijo: Icono(Icons.  
                eliminar, color:  
                Colors.white, ),  
  
        ),  
        fondo secundario: Contenedor( color:  
            Colors.red, alineación:  
            Alignment.centerRight, padding: EdgeInsets.only  
(right: 16.0), child: Icon(Icons.delete, color: Colors.white, ),  
  
        ),  
        hijo: ListTile(  
            inicial: Columna (hijos:  
  
            <Widget>[ Text(DateTimeFormat.d().format(DateTime.parse(snapshot.data[index].date))),  
            estilo: TextStyle  
                (fontWeight: FontWeight.bold, fontSize:  
                32.0, color: Colors.blue),  
  
            ),  
            Texto(DateTimeFormat.E().format(DateTime.parse(snapshot.data[index].date))), ],  
  
        ),  
        texto del título(  
            _titleDate, estilo:  
            TextStyle(fontWeight: FontWeight.bold), ), subtítulo: Texto(_subtitle),  
  
        onTap: () {
```

```
_addOrEditJournal( añadir:  
    falso, índice:  
    índice, diario:  
    instantánea.datos[índice],  
  
); }, onDismissed: (dirección)  
{ setState(()  
{ _database.journal.removeAt(index); });  
  
DatabaseFileRoutines().writeJournals(databaseToJson(_database)); }, ); },  
  
separatorBuilder: (contexto BuildContext, int index) { return  
    Divider( color:  
    Colors.grey, ); }, );  
  
}
```



CÓMO FUNCIONA

Completó el archivo home.dart que es responsable de mostrar una lista de entradas de diario con la capacidad de agregar, modificar y eliminar registros individuales.

FutureBuilder () llama al método _loadJournals() que recupera las entradas del diario y, mientras se cargan los datos, se muestra un CircularProgressIndicator() , y cuando se devuelven los datos, el constructor llama al método _buildListViewSeparated(snapshot) pasando la instantánea, que es la lista de entradas de diario.

El método _loadJournals() recupera las entradas del diario llamando a las clases de la base de datos para leer el archivo de la base de datos local, convertir objetos JSON en una lista, ordenar las entradas por fecha DESC y devolver la lista de entradas del diario.

El método _addOrEditJournal() maneja la adición de nuevas entradas o la modificación de una entrada de diario. El constructor toma tres parámetros con nombre para ayudarlo si está agregando o modificando una entrada. Utiliza la clase de base de datos JournalEdit para rastrear la acción a realizar dependiendo de si el usuario presionó el botón Cancelar o Guardar. Para mostrar la página de entrada de edición, pasa los argumentos del constructor llamando a Navigator.push() y usa la palabra clave await para recibir la acción realizada desde la página de entrada de edición a la variable _journalEdit . La declaración de cambio se utiliza para evaluar la acción realizada para guardar la entrada de diario o cancelar los cambios.

El método _buildListViewSeparated(snapshot) utiliza el constructor ListView.separated() para crear la lista de entradas de diario. El itemBuilder devuelve un widget Dismissible() que maneja la eliminación de entradas de diario deslizando el dedo hacia la izquierda o hacia la derecha en la entrada. La propiedad secundaria Dismissible() usa ListTile() para dar formato a cada entrada de diario en ListView. SeparatorBuilder devuelve el widget Divider() para mostrar una línea divisoria gris entre las entradas del diario .

RESUMEN

En este capítulo, aprendió cómo conservar los datos guardando y leyendo localmente en el sistema de archivos del dispositivo iOS y Android. Para el dispositivo iOS, usó NSDocumentDirectory , y para el dispositivo Android, usó el directorio AppData . El popular formato de archivo JSON se utilizó para almacenar las entradas del diario en un archivo. Creó una aplicación de registro de estado de ánimo que ordena la lista de entradas por fecha DESC y permite agregar y modificar registros.

Aprendió a crear las clases de base de datos para manejar la persistencia local para codificar y decodificar objetos JSON y escribir y leer entradas en un archivo. Aprendió a crear la clase Rutinas de archivo de base de datos para obtener la ruta del directorio de documentos del dispositivo local y guardar y leer el archivo de la base de datos usando la clase Archivo . Aprendió cómo crear la clase de base de datos que maneja la decodificación y codificación de objetos JSON y cómo los convierte en una lista de entradas de diario. Aprendió a usar json.encode para analizar valores en una cadena JSON y json.decode para analizar la cadena en un objeto JSON. La clase de base de datos devuelve una lista de clases de diario , List<Diario>. Aprendió a crear la clase Diario para manejar la decodificación y codificación de los objetos JSON para cada entrada de diario.

La clase Journal contiene los campos id, date, mood y note almacenados como tipo String . Aprendió a crear la clase JournalEdit responsable de pasar una acción y entradas individuales de la clase Journal entre páginas.

Aprendió a crear una página de entrada de diario que maneja tanto la adición como la modificación de una entrada de diario existente. Aprendió a usar la clase `JournalEdit` para recibir una entrada de diario y devolverla a la página de inicio con una acción y la entrada modificada. Aprendió a llamar a `showDatePicker()` para presentar un calendario para seleccionar una fecha de diario. Aprendió a usar la clase `DateFormat` con diferentes constructores de formato para mostrar fechas como "domingo, 13 de enero de 2019". Aprendió a usar `DateTime.parse()` para convertir una fecha guardada como `String` en una instancia de `DateTime`. Aprendió a usar el widget `TextField` con `TextEditingController` para acceder a los valores de entrada. Aprendió a personalizar el botón de acción del teclado configurando la acción de entrada de texto de `TextField`. Aprendió a mover el foco entre los widgets de `TextField` usando `FocusNode` y el botón de acción del teclado.

Aprendió a crear la página de inicio para mostrar una lista de entradas de diario ordenadas por fecha DESC separadas por un divisor. Aprendió a usar el constructor `ListView.separated` para separar fácilmente cada entrada de diario con un separador. `ListView.separated` usa dos constructores, y aprendió a usar `itemBuilder` para mostrar la lista de entradas de diario y `separatorBuilder` para agregar un widget de divisor entre entradas. Usó `ListTile` para formatear la Lista de entradas de diario fácilmente. Aprendió a personalizar la propiedad inicial con una columna para mostrar el día y el día de la semana en el lado inicial de `ListTile`. Usó el widget `Descartable` para facilitar la eliminación de entradas de diario deslizando el dedo hacia la izquierda o hacia la derecha en la entrada misma (`ListTile`). Usó el constructor `Key('id')` para establecer una clave única para cada widget descartable para asegurarse de que se elimine la entrada de diario correcta.

En el próximo capítulo, aprenderá a configurar la base de datos NoSQL backend de Cloud Firestore. Cloud Firestore le permite almacenar, consultar y sincronizar datos entre dispositivos sin configurar su servidores propios.

LO QUE APRENDISTE EN ESTE CAPÍTULO

TEMA	CONCEPTOS CLAVE
Clases de base de datos	Las cuatro clases de base de datos administran colectivamente la persistencia local escribiendo, leyendo, codificando y decodificando objetos JSON hacia y desde el archivo JSON.
Clase DatabaseFileRoutines	Esto maneja la clase de archivo para recuperar el directorio de documentos locales del dispositivo y guardar y leer el archivo de datos.
Clase de base de datos	Esto maneja la decodificación y codificación de los objetos JSON y su conversión a una lista de entradas de diario.
clase de diario	Esto maneja la decodificación y codificación de los objetos JSON para cada entrada de diario.
Clase JournalEdit	Esto maneja el paso de entradas de diario individuales entre páginas y la acción tomada.
showDatePicker	Esto presenta un calendario para seleccionar una fecha.
Formato de fecha	Esto da formato a las fechas utilizando diferentes constructores de formato.
DateTime.parse()	Esto convierte una cadena en una instancia de fecha y hora .
Campo de texto	Esto permite la edición de texto.
Controlador de edición de texto	Esto permite el acceso al valor del TextField asociado.
AcciónEntradaTexto	TextField TextInputAction permite la personalización del botón de acción del teclado.
FocusNode	Esto mueve el foco entre los widgets de TextField .
ListView.separated	Esto utiliza dos constructores, itemBuilder y separatorBuilder.
descartable	Desliza para descartar arrastrando. Use onDismissed para llamar a acciones personalizadas, como eliminar un registro.
Lista().ordenar	Ordenar una lista usando un comparador.
Navegador	Esto se utiliza para navegar a otra página. Puede pasar y recibir datos en una clase utilizando el Navegador.
Futuro	Puede recuperar posibles valores disponibles en el futuro.
FuturoConstructor	Esto funciona con un futuro para recuperar los datos más recientes sin bloquear la interfaz de usuario.

TEMA	CONCEPTOS CLAVE
CircularProgressIndicator Este es un indicador de progreso circular giratorio que muestra que se está ejecutando una acción.	
paquete ruta_proveedor	Puede acceder a las ubicaciones locales del sistema de archivos de iOS y Android.
paquete internacional	Puede usar DateFormat para dar formato a las fechas.
biblioteca dart:io	Puede utilizar la clase de archivo .
dardo:convertir biblioteca	Puede decodificar y codificar objetos JSON.
dardo: matemáticas	Esto se usa para llamar al generador de números Random() .

14

Agregar Firebase y Servidor de Firestore

LO QUE APRENDERÁS EN ESTE CAPÍTULO

Cómo crear un proyecto de Firebase

Cómo registrar los proyectos iOS y Android para usar Firebase

Cómo agregar una base de datos de Cloud Firestore

Cómo estructurar y crear un modelo de datos para la base de datos de la tienda
Cloud Fire

Cómo habilitar y agregar la autenticación de Firebase

Cómo crear reglas de seguridad de Firestore

Cómo crear la estructura base de la aplicación del cliente Flutter

Cómo agregar Firebase a iOS y cómo agregar los proyectos de Android con los archivos
del servicio de Google

Cómo agregar los paquetes Firebase y Cloud Firestore

Cómo agregar el paquete intl para formatear fechas

Cómo personalizar la apariencia de AppBar y BottomAppBar usando los
widgets BoxDecoration y LinearGradient

En este capítulo, en el Capítulo 15 y en el Capítulo 16, utilizará técnicas que ha aprendido en capítulos anteriores junto con nuevos conceptos y los unirá para crear una aplicación de diario de estado de ánimo a nivel de producción. En los capítulos anteriores, creó muchos proyectos que le enseñaron diferentes formas de implementar tareas y objetivos específicos. En una aplicación de nivel de producción, debe combinar lo que ha aprendido para mejorar el rendimiento al volver a dibujar solo los widgets con

cambios de datos, pasar estado entre páginas y subir el árbol de widgets, manejar las credenciales de autenticación del usuario, sincronizar datos entre dispositivos y la nube, y crear clases que manejen la lógica independiente de la plataforma entre aplicaciones móviles y web.

Debido a que Google tiene compatibilidad con Flutter de código abierto para aplicaciones web y de escritorio, los servicios backend de Firebase se pueden usar para aplicaciones web y de escritorio de Flutter, no solo para dispositivos móviles. Es por eso que en este capítulo aprenderá cómo desarrollar una aplicación móvil de nivel de producción.

Especificamente, aprenderá cómo usar la autenticación y conservar los datos en una base de datos en la nube usando Firebase de Google (infraestructura de servidor backend), Firebase Authentication y Cloud Firestore.

Aprenderá a crear y configurar un proyecto de Firebase utilizando Cloud Firestore como base de datos en la nube.

Cloud Firestore es una base de datos de documentos NoSQL para almacenar, consultar y sincronizar datos con soporte sin conexión para aplicaciones móviles y web. ¿Dije fuera de línea? Sí, lo hice. La capacidad de una aplicación móvil para funcionar mientras no hay una conexión a Internet disponible es una característica imprescindible que los usuarios esperan. Otra gran característica de usar Cloud Firestore es la capacidad de sincronizar datos en vivo entre dispositivos automáticamente. La sincronización de datos es rápida, lo que permite la colaboración entre diferentes dispositivos y usuarios. La parte sorprendente de usar estas potentes funciones es que no tiene que ocuparse de configurar y administrar la infraestructura del servidor. Esta función le permite crear aplicaciones sin servidor.

Configurará el proveedor de autenticación de back-end de Firebase, la base de datos y las reglas de seguridad para sincronizar datos entre varios dispositivos y plataformas. Para habilitar los servicios de base de datos y autenticación para el proyecto Flutter del lado del cliente, agregará los paquetes `firebase_auth` y `cloud_firestore`. En el Capítulo 15 y el Capítulo 16, aprenderá cómo implementar la administración de estado local y de toda la aplicación mediante el uso de la clase `InheritedWidget` y maximizar el uso compartido del código de la plataforma mediante la implementación del patrón Business Logic Component. Utilizará la administración estatal local y de toda la aplicación para solicitar datos de diferentes clases de servicio.

¿QUÉ SON FIREBASE Y CLOUD FIRESTORE?

Antes de comenzar a configurar, echemos un vistazo a lo que abarca Firebase. Firebase consta de una plataforma con una multitud de productos que comparten y funcionan juntos. Firebase maneja toda la infraestructura del servidor back-end que conecta iOS, Android y aplicaciones web.

Crear aplicaciones

Cloud Firestore: almacene y sincronice datos NoSQL en documentos y colecciones entre dispositivos

Base de datos en tiempo real: almacene y sincronice datos NoSQL como un gran árbol JSON entre dispositivos

Almacenamiento en la nube: almacene y sirva archivos

Funciones en la nube: ejecute el código de back-end

Autenticación: autenticación segura del usuario

Hosting: entregue activos de aplicaciones web

Garantice la calidad de la aplicación

Crashlytics: informes de fallas en tiempo real

Supervisión del rendimiento: rendimiento de la aplicación

Laboratorio de pruebas : pruebe aplicaciones en dispositivos alojados por Google

hacer crecer el negocio

Mensajería en la aplicación: envíe mensajes a los usuarios

Google Analytics: realice análisis de aplicaciones

Predicciones: segmentación de usuarios basada en el comportamiento

Prueba A/B : optimice la experiencia de la aplicación

Mensajería en la nube: envíe mensajes y notificaciones

Configuración remota : modifique la aplicación sin implementar una nueva versión

Enlaces dinámicos: enlaces profundos de aplicaciones

Indexación de aplicaciones: dirija el tráfico de búsqueda a una aplicación móvil

Wow, esto suena increíble, pero ¿cuesta una fortuna? En realidad no; Google ofrece el Plan Spark, que le brinda acceso gratuito a la mayoría de los productos, especialmente a Cloud Firestore. Puede ver los detalles y las restricciones en <https://firebase.google.com/pricing>.

En este capítulo, me centraré en el uso de Cloud Firestore para almacenar y sincronizar datos entre dispositivos. Para obtener información general sobre Firebase, puede navegar por <https://firebase.google.com>.

¿Qué es Cloud Firestore? Almacena datos en documentos organizados en colecciones, similar a JSON. Puede escalar datos complejos y jerárquicos mediante el uso de subcolecciones dentro de los documentos. Lo examinará detalladamente en la siguiente sección, "Estructuración y modelado de datos Cloud Firestore". Ofrece soporte sin conexión para iOS, Android y aplicaciones web. Para las plataformas iOS y Android, la persistencia sin conexión está habilitada de manera predeterminada, pero para la Web, la persistencia sin conexión está deshabilitada de manera predeterminada. Para optimizar la consulta de datos, admite consultas indexadas con clasificación y filtrado. Puede usar transacciones que se repiten automáticamente hasta que se completa la tarea. El escalado de datos se maneja automáticamente por usted. Utiliza los SDK móviles para integrar diferentes funciones de los productos de Firebase en sus aplicaciones.

Estructuración y modelado de datos Cloud Firestore

Para comprender la estructura de datos de Cloud Firestore, comparémosla con una base de datos de SQL Server estándar (consulte la Tabla 14.1). La base de datos de SQL Server es un sistema de administración de bases de datos relacionales (RDBMS) que admite el modelado de datos de tablas y filas (Figura 14.1). La comparación no es uno a uno sino una guía ya que las estructuras de datos difieren.

TABLA 14.1: Comparación de estructuras de datos

BASE DE DATOS DEL SERVIDOR SQL	FUEGO EN LA NUBE
Mesa	Recopilación
Fila	Documento
columnas	Datos

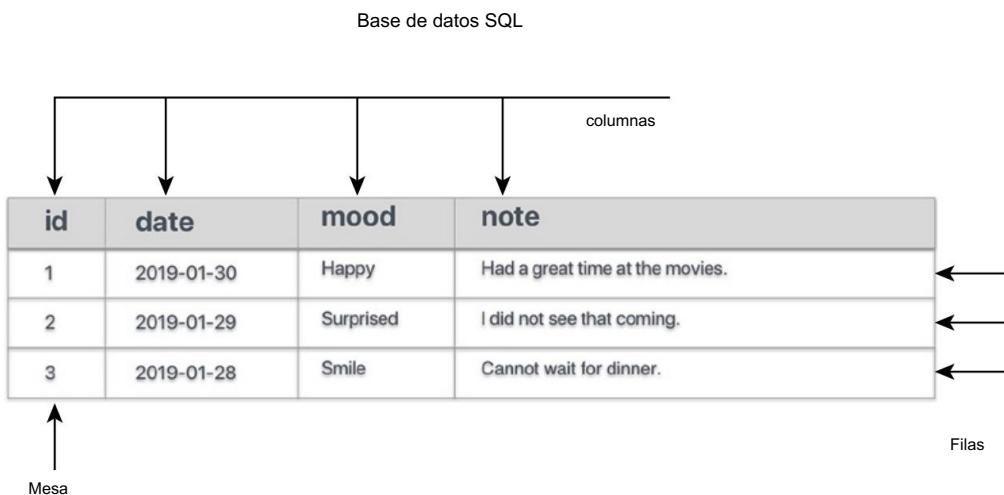


FIGURA 14.1: Modelo de datos de la base de datos de SQL Server

En Cloud Firestore, una colección solo puede contener documentos. Un documento es un par clave-valor y, opcionalmente, puede apuntar a subcolecciones. Los documentos no pueden apuntar a otro documento y deben almacenarse en colecciones (Figura 14.2).

¿Cuál es la responsabilidad de la colección? Las colecciones son contenedores de documentos; los sostienen de la misma manera que una carpeta sostiene las páginas.

¿Cuál es la responsabilidad del documento? Los documentos contienen datos que se almacenan como un par clave-valor similar a JSON. Los documentos admiten tipos de datos adicionales que JSON no admite. Cada documento se identifica por su nombre y su tamaño está limitado a 1 MB (ver Tabla 14.2).

Tienda de fuego en la nube

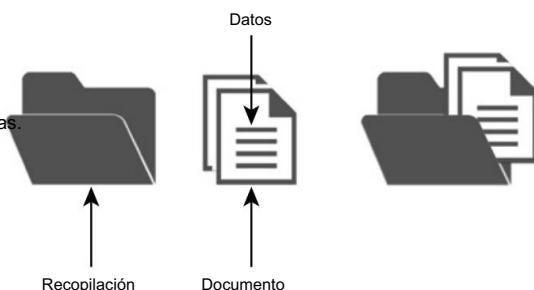


FIGURA 14.2: Modelo de datos de Cloud Firestore

TABLA 14.2: Datos de muestra de Cloud Firestore

TIPO	VALOR
Recopilación	diarios
Documento	R5NcTWAaWtHTttYtPoOd
Documentar datos como par clave-valor	<p>fecha: "2019-0202T13:41:12.537285"</p> <p>Estado de ánimo: "Feliz"</p> <p>nota: "Gran película".</p> <p>uid: "F1GGeKiwp3jRpoCVskdBNmO4GUN4"</p>

Echemos un vistazo a los datos de muestra de Cloud Firestore como objetos JSON y en la consola de Cloud Firestore (Figura 14.3). Tenga en cuenta que el nombre del documento es una identificación única que Cloud Firestore puede crear automáticamente, o puede generarla manualmente.

```
{
  "revistas":{ {
    "R5NcTWAaWtHTttYtPoOd1":{
      "date": "2019-0202T13:41:12.537285", "mood":"Feliz",
      "note":"Gran película".
      "uid":
      "F1GGeKiwp3jRpoCVskdBNmO4GUN4",
    }
  }
}
```

The screenshot shows the Cloud Firestore interface. At the top, there's a navigation bar with icons for home, collections, and documents. Below it, a sidebar shows a collection named 'journals' with a sub-document named 'R5NcTWAaWtHTttYtPoOd'. The main area displays the document's data in a JSON-like format:

```

{
  "date": "2019-02-02T13:41:12.537285",
  "mood": "Happy",
  "note": "Great movie.",
  "uid": "F1GGeKiwp3jRpoCVskdBNmO4GUN4"
}

```

FIGURA 14.3: Colección y Documento

Cloud Firestore admite muchos tipos de datos, como matriz, booleano, byte, fecha y hora, número de coma flotante, punto geográfico, entero, mapa, referencia, cadena de texto y nulo. Una de las principales ventajas de usar Cloud Firestore es la sincronización automática de datos entre dispositivos y la capacidad de la aplicación cliente para continuar trabajando sin conexión mientras Internet no está disponible.

Ver las capacidades de autenticación de Firebase

Agregar seguridad a una aplicación es extremadamente importante para mantener la información privada y segura. Fire base Authentication proporciona servicios de backend integrados a los que se puede acceder desde el SDK del cliente para admitir la autenticación completa. La siguiente es una lista de los proveedores de inicio de sesión de autenticación disponibles actualmente:

Correo electrónico/Contraseña

Teléfono

Google

Jugar Juegos (Google)

Centro de juegos (Apple)

Facebook _

Gorjeo

GitHub

Anónimo

Cada proveedor de inicio de sesión de autenticación está deshabilitado de manera predeterminada y usted habilita los proveedores necesarios para las especificaciones de su aplicación (Figura 14.4). Le mostraré cómo habilitar un proveedor de inicio de sesión más adelante en este capítulo.

The screenshot shows the Firebase Authentication interface in a web browser. On the left, there's a sidebar with 'Project Overview' and sections for 'Develop' (Authentication, Database, Storage, Hosting, Functions, ML Kit), 'Quality' (Crashlytics, Performance, Test Lab), 'Analytics' (Dashboard, Events, Conversions, A/B Testing), and 'Grow' (Predictions, A/B Testing, Cloud Messaging). At the bottom of the sidebar are 'Spark' (Free \$0/month) and 'Upgrade' buttons. The main content area has a blue header 'Authentication'. Below it are tabs for 'Users', 'Sign-in method', 'Templates', and 'Usage'. The 'Sign-in providers' section lists various providers with their status: Email/Password (Disabled), Phone (Disabled), Google (Disabled), Play Games (Disabled), Game Center (Beta, Disabled), Facebook (Disabled), Twitter (Disabled), GitHub (Disabled), and Anonymous (Disabled). There's also a 'Web setup' button and a 'Go to docs' link.

FIGURA 14.4: Proveedores de inicio de sesión de Firebase Authentication

Una nota importante sobre el uso del proveedor anónimo es que si el usuario elimina la aplicación del dispositivo y la vuelve a instalar, se crea un nuevo usuario anónimo y no tendrá acceso a los datos anteriores. Los datos del usuario anónimo original aún se almacenan en el backend, pero la aplicación del cliente no tiene forma de conocer la identificación del usuario anónimo original ya que la aplicación se eliminó del dispositivo. Un ejemplo del uso del proveedor de inicio de sesión anónimo es permitir al usuario acceso gratuito a la aplicación y permitir una ruta de actualización paga a funciones avanzadas.

Desde su aplicación móvil, recupera las credenciales de autenticación del usuario de cualquiera de los proveedores de inicio de sesión disponibles. Pasas estas credenciales al SDK de autenticación de Firebase del cliente y los servicios de backend de la base de Fire verifican si las credenciales son válidas y devuelven una respuesta a la aplicación del cliente. Si las credenciales son válidas, permite que el usuario acceda a los datos y las páginas de la aplicación según las reglas de seguridad que aprenderá en la siguiente sección, "Visualización de las reglas de seguridad de Cloud Firestore".

Una vez que se habilite un proveedor de inicio de sesión en particular, use el SDK de autenticación de Firebase para crear un objeto de usuario de Firebase guardado en el backend de Firebase. El objeto Usuario de Firebase tiene un conjunto de vínculos de propiedades que puede personalizar, a excepción de la ID única. Puede personalizar la dirección de correo electrónico principal, el nombre y la URL de una foto (normalmente, la foto del usuario o un avatar). Al usar la identificación única del objeto de usuario de Firebase, vincula todos los datos que el usuario ha creado a la identificación.

Ver las reglas de seguridad de Cloud Firestore

Para proteger el acceso a las colecciones y los documentos, implementa las reglas de seguridad de Cloud Firestore. En la sección anterior, aprendiste que creas un objeto de usuario de Firebase a través del SDK de autenticación de Firebase. Una vez que tenga el ID único del objeto de usuario de Firebase, lo utilizará con las reglas de seguridad de Cloud Firestore para proteger y bloquear los datos de cada usuario. El siguiente código muestra las reglas de seguridad que creará para proteger la base de datos de Cloud Firestore:

```
service cloud.firestore { match /  
databases/{database}/documents { match /journals/  
{document=**} { permitir lectura, escritura: if  
resource.data.uid == request.auth.uid; permitir crear: si request.auth.uid != null;  
}  
}}
```

Las reglas se pueden editar desde la página de reglas de la base de datos en el sitio web de la consola de Firebase. Las reglas consisten en utilizar las sentencias de coincidencia para identificar documentos, con las expresiones allow para controlar el acceso a los documentos. Cada vez que cambia las reglas y las guarda, se crea automáticamente un historial de cambios, lo que le permite revertir los cambios si es necesario (Figura 14.5).

Echemos un vistazo a un ejemplo que requiere reglas para permitir que un usuario lea y escriba los documentos que se le asignan. La declaración de la primera coincidencia /bases de datos/{base de datos}/documentos le dice a las reglas que coincidan con cualquier base de datos de Cloud Firestore en el proyecto.

```
coincidencia /bases de datos/{base de datos}/documentos
```

La segunda y principal parte de la comprensión es usar la declaración de coincidencia para apuntar a la colección y la expresión a evaluar, como en match /journals/{document=**}. La declaración de diarios es el nombre del contenedor, y la expresión a evaluar es document=** (todos los documentos para el

colección de revistas) entre corchetes. Dentro de esta coincidencia en particular, usa la expresión allow para los privilegios de lectura y escritura usando la instrucción if para ver si el valor del recurso .data.uid es igual al de request.auth.uid. resource.data.uid es el campo uid dentro del documento, y request.auth.uid es el ID único del usuario que ha iniciado sesión.

```
coincide con /journals/{document=**}
  { permitir leer, escribir: if resource.data.uid == request.auth.uid;
  }
```

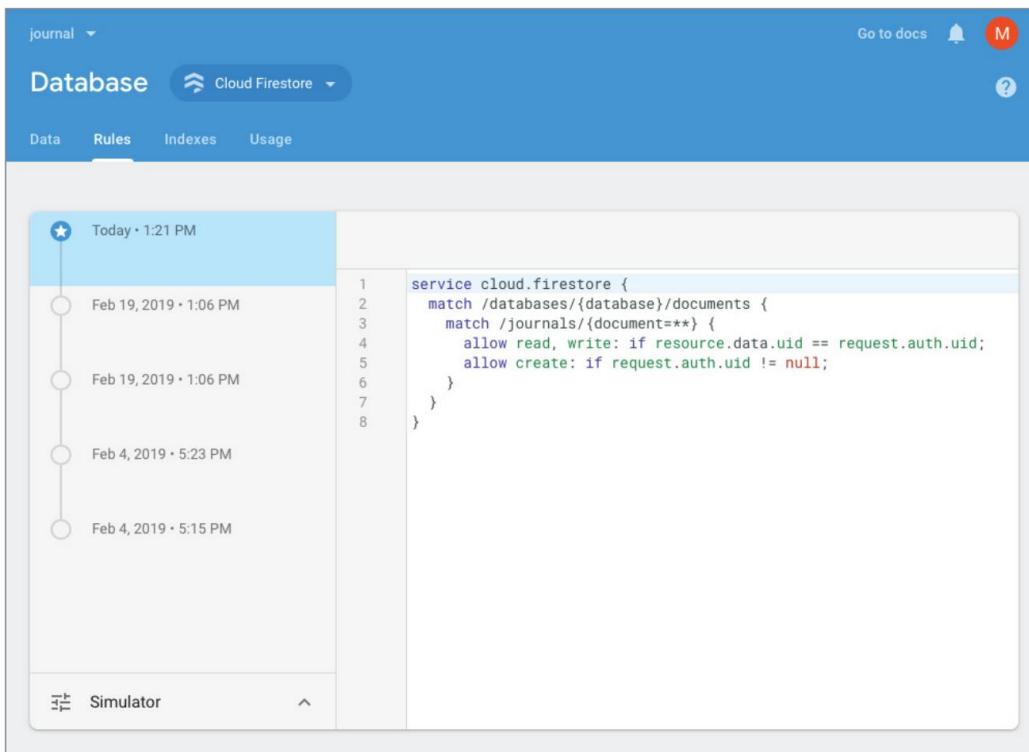


FIGURA 14.5: Reglas de Cloud Firestore

Puede desglosar las reglas aún más usando una declaración de coincidencia para cada acción de lectura o escritura . Para la regla de lectura , se puede dividir en obtener y listar. Para la regla de escritura , se puede dividir en crear, actualizar y eliminar.

```
// Leer
permite obtener: si <condición>;
lista de permitidos: si <condición>;

// Escribir
permitir crear: si <condición>; permitir
actualización: si <condición>; permitir
eliminar: si <condición>;
```

El siguiente ejemplo utiliza la regla de creación para permitir que solo los usuarios autenticados agreguen nuevos registros comprobando si el valor de `request.auth.id` no es igual a un valor nulo :

```
permitir crear: si request.auth.uid != null;
```

CONFIGURAR EL PROYECTO FIREBASE

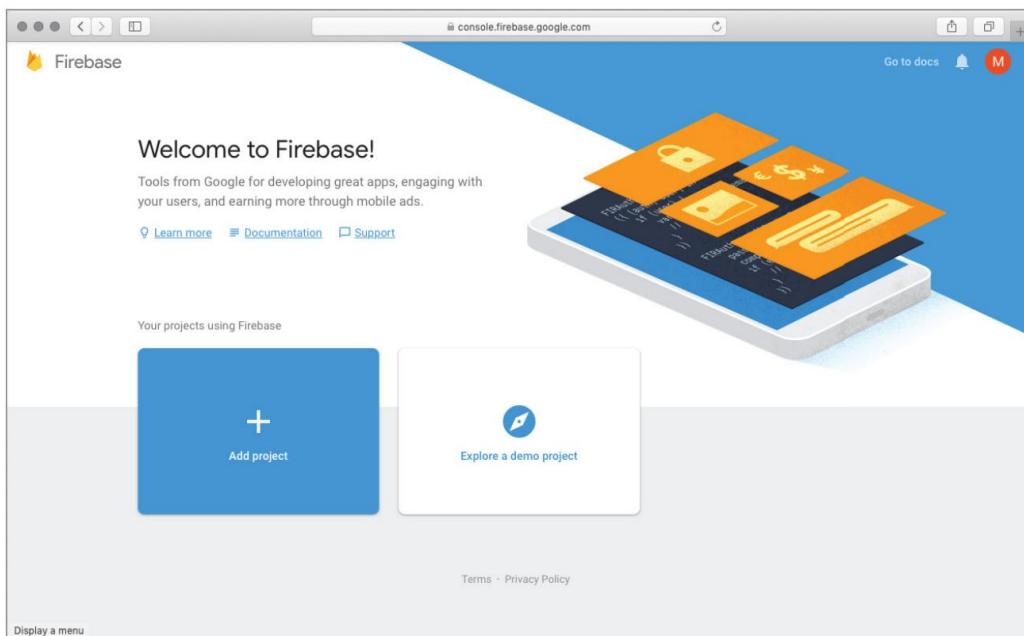
Ahora comprende cómo Cloud Firestore almacena datos y los beneficios de sincronizar varios dispositivos con persistencia de datos sin conexión. La función fuera de línea funciona almacenando en caché una copia de los datos activos de la aplicación, haciéndola accesible cuando el dispositivo está fuera de línea. Antes de poder usar Cloud Firestore en su aplicación, debe crear un proyecto de Firebase.

Un proyecto de Firebase está respaldado por Google Cloud Platform, que permite escalar las aplicaciones. El proyecto Firebase es un contenedor que admite funciones compartidas, como la base de datos, las notificaciones, los usuarios, la configuración remota, los informes de fallas y el análisis (muchas más) entre iOS, Android y las aplicaciones web. Cada cuenta puede tener varios proyectos, por ejemplo, para separar aplicaciones diferentes y no relacionadas.

PRUÉBALO Crear el proyecto de Firebase

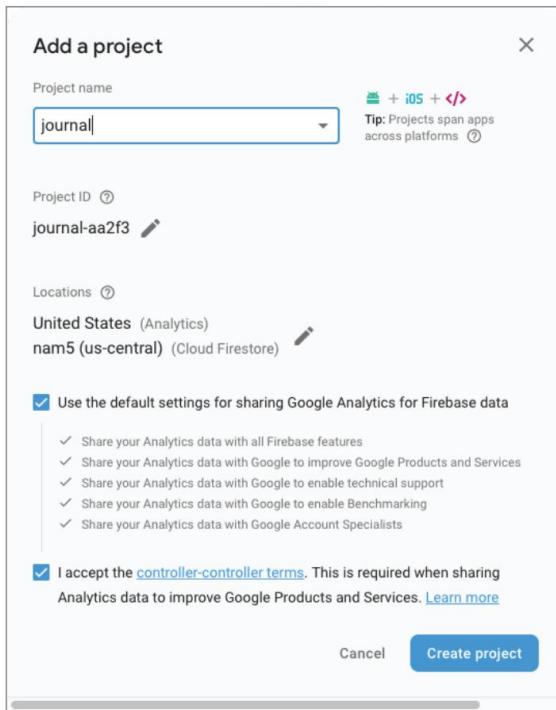
Debe crear un proyecto de Firebase que configure un contenedor para comenzar a agregar su base de datos de Cloud Firestore y habilitar la autenticación. Comenzará agregando la aplicación de iOS y luego continuará agregando la aplicación de Android.

1. Navegue a <https://console.firebaseio.google.com> e inicie sesión en Google Firebase con su cuenta de Google. Si no tiene una cuenta de Google, puede crear una en <https://accounts.google.com/SignUp>.

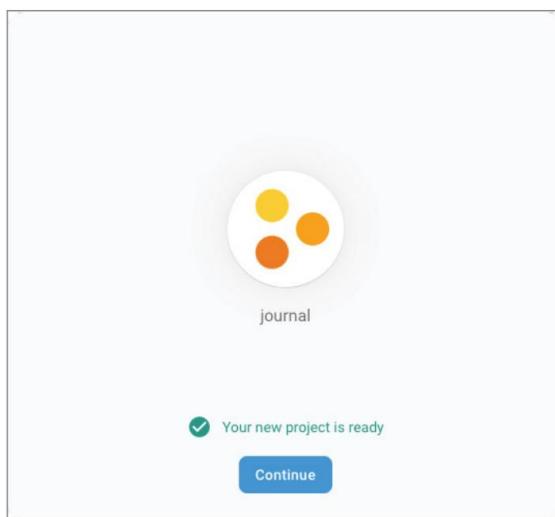


2. Haga clic en el botón Agregar proyecto en Firebase; se abrirá el cuadro de diálogo Agregar un proyecto. Para el nombre del proyecto, ingrese diario; el ID del proyecto se crea automáticamente para usted. Observe que toma el nombre del proyecto y le agrega un identificador único como journal-aa2f3. (Tu ID será diferente porque cada nombre de proyecto debe ser único).

La ubicación del proyecto de Cloud Firestore se selecciona automáticamente, pero puede cambiarla si lo desea. Deberá seleccionar cada casilla de verificación para aceptar los términos de Google Analytics; luego haga clic en el botón Crear proyecto.

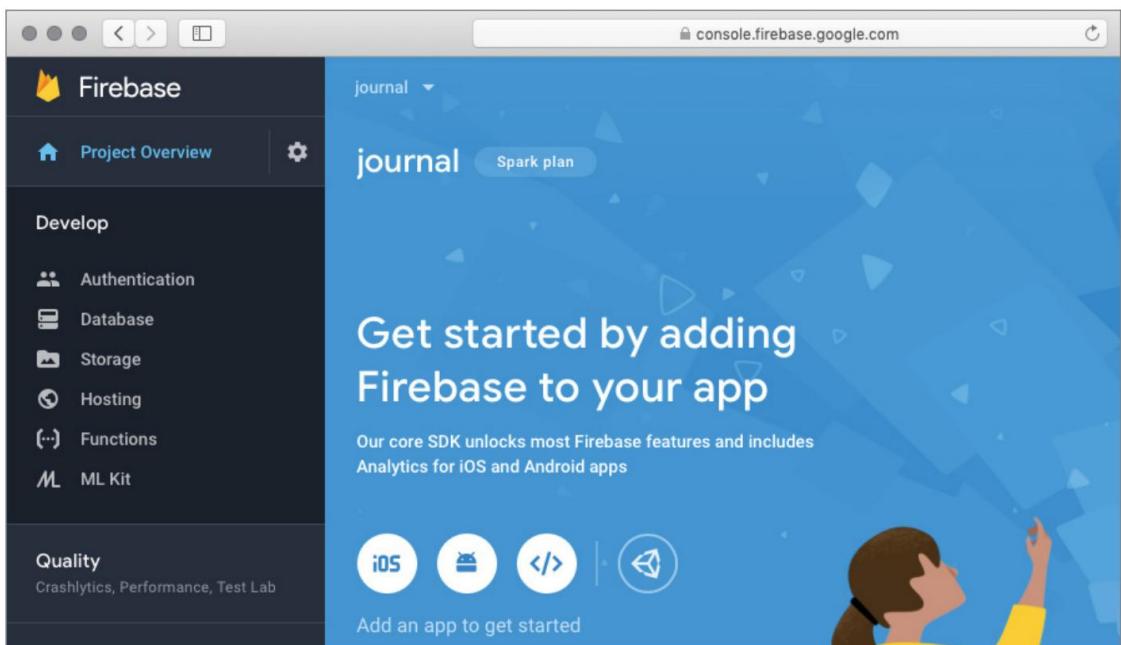


3. Cuando se le presente el cuadro de diálogo que muestra que el nuevo proyecto está listo, haga clic en Continuar botón.

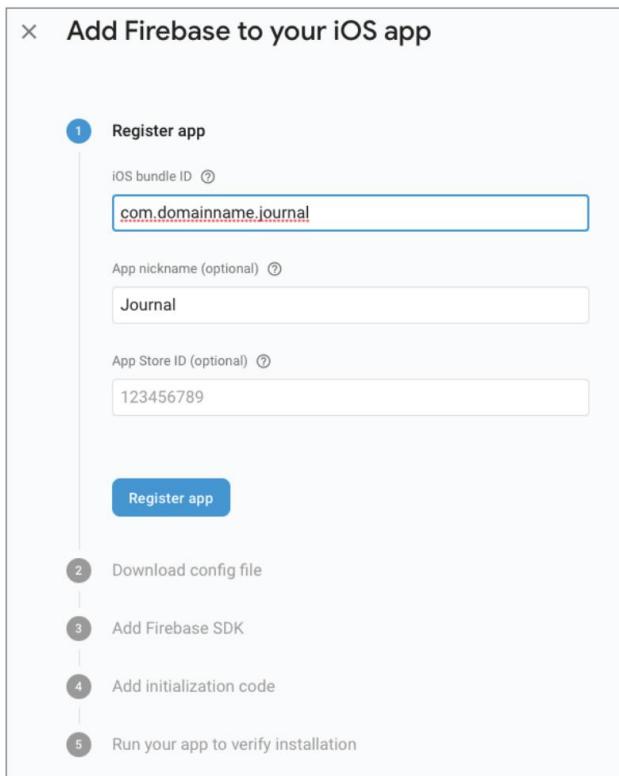


iOS

4. Desde la página principal del proyecto Firebase, haga clic en el botón iOS para agregar Firebase a la aplicación iOS que creó en la sección "Agregar paquetes de autenticación y Cloud Firestore a la aplicación cliente".



5. Ingrese el ID del paquete de iOS, como en com.domainname.journal. El ID del paquete es el dominio inverso nombre, com.domainname, combinado con el nombre de la aplicación Flutter de la revista.
6. Ingrese el apodo de la aplicación opcional Diario y omita la ID de la tienda de aplicaciones opcional, ya que se obtiene una vez que se crea una aplicación desde iTunes Connect de Apple para enviar una aplicación para aprobación de distribución. Haga clic en el botón Registrar aplicación.



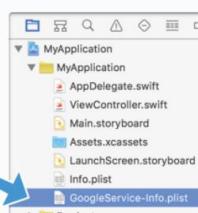
7. Haga clic en el botón Descargar GoogleService-Info.plist. (Agregará el archivo GoogleService-Info.plist descargado a su proyecto Xcode en la sección "Agregar paquetes de autenticación y Cloud Firestore a la aplicación del cliente").
8. Haga clic en el botón Siguiente y omita los pasos Agregar SDK de Firebase y Agregar código de inicialización. Se salta estos pasos porque el SDK de Firebase se agrega en la sección "Agregar paquetes de autenticación y Cloud Firestore a la aplicación cliente".

Add Firebase to your iOS app

1 Register app
iOS bundle ID: com.domainname.ch1415journal, App nickname: Journal

2 Download config file
[Download GoogleService-Info.plist](#)

Move the GoogleService-Info.plist file you just downloaded into the root of your Xcode project and add it to all targets.



Previous **Next**

3 Add Firebase SDK

4 Add initialization code

5 Run your app to verify installation

9. Haga clic en Omitir este paso en el paso Ejecute su aplicación para verificar la instalación.

Add Firebase to your iOS app

1 Register app
iOS bundle ID: com.domainname.ch1415journal, App nickname: Journal

2 Download config file

3 Add Firebase SDK

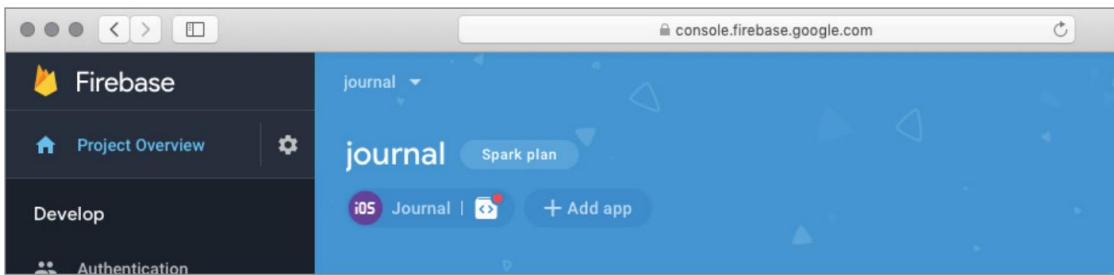
4 Add initialization code

5 Run your app to verify installation

 Checking if the app has communicated with our servers. You may need to uninstall and reinstall your app.

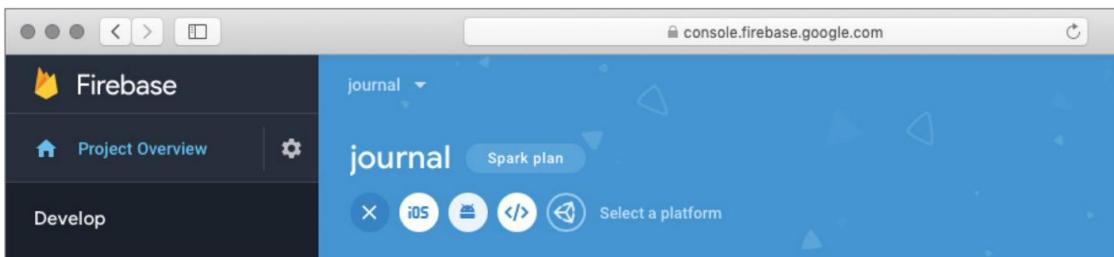
Previous Continue to console [Skip this step](#)

10. En la página principal del proyecto de Firebase, haga clic en el botón Agregar aplicación.

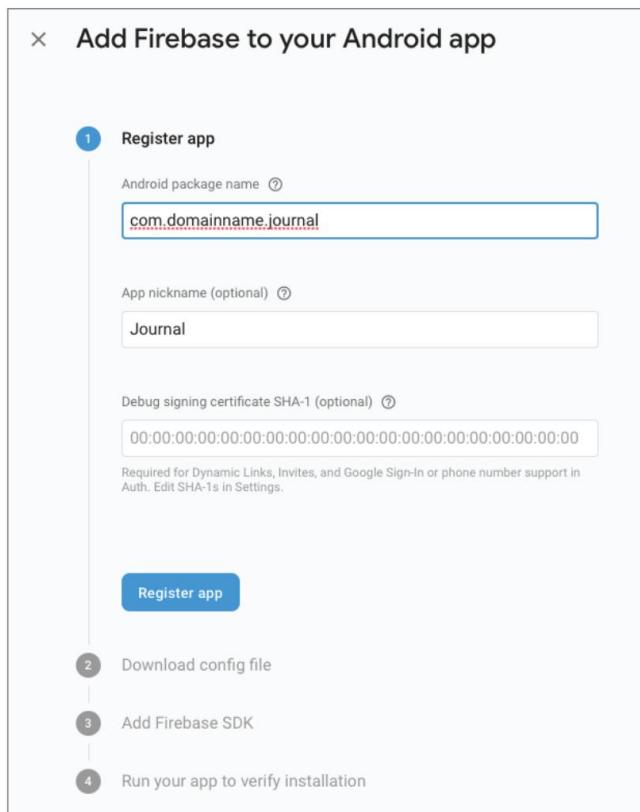


ANDROIDE

11. Haga clic en el botón de Android para agregar Firebase al proyecto de Android que creó en la sección "Agregar paquetes de autenticación y Cloud Firestore a la aplicación cliente".



12. Introduzca el nombre del paquete de Android, como com.domainname.journal. El ID del paquete es el nombre de dominio inverso, com.domainname, combinado con el nombre de la revista de la aplicación Flutter. Creará la aplicación Flutter en la sección "Agregar paquetes de autenticación y Cloud Firestore a la aplicación cliente".
13. Ingrese el apodo opcional de la aplicación Diario y omita el paso del certificado de firma de depuración opcional SHA-1. Haga clic en el botón Registrar aplicación.



14. Haga clic en el botón Descargar googleservices.json. (Agregará el archivo .json de googleservices descargado a su proyecto de Android en la sección "Agregar paquetes de autenticación y Cloud Firestore a la aplicación del cliente").
15. Haga clic en el botón Siguiente y omita el paso Agregar SDK de Firebase. Se salta este paso porque el SDK de Firebase se agrega en la sección "Agregar paquetes de autenticación y Cloud Firestore a la aplicación cliente".

×

Add Firebase to your Android app

1 Register app
Android package name: com.domainname.ch1415journal, App nickname: Journal

2 Download config file

[Download google-services.json](#)

Instructions for Android Studio below | C++

Switch to the Project view in Android Studio to see your project root directory.

Move the google-services.json file you just downloaded into your Android app module root directory.

Project view in Android Studio showing the project structure:

- MyApplication (~/Desktop/MyApplication)
- build
- libs
- src
- .gitignore
- app.iml
- build.gradle
- google-services.json
- proguard-rules.pro
- gradle

Next

3 Add Firebase SDK

4 Run your app to verify installation

16. Haga clic en Omitir este paso en el paso Ejecute su aplicación para verificar la instalación.

×

Add Firebase to your Android app

1 Register app
Android package name: com.domainname.ch1415journal, App nickname: Journal

2 Download config file

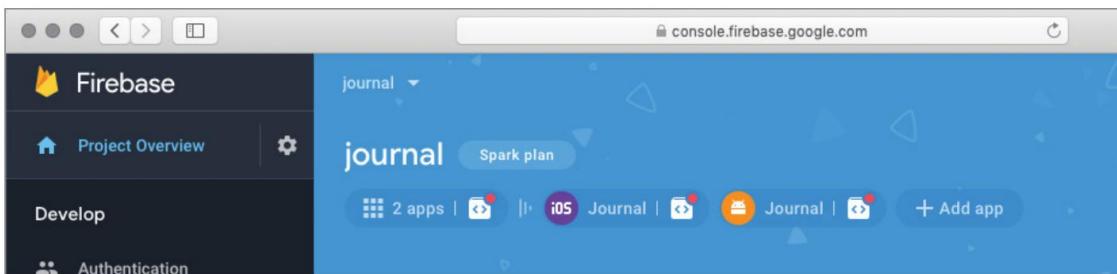
3 Add Firebase SDK

4 Run your app to verify installation

Checking if the app has communicated with our servers. You may need to uninstall and reinstall your app.

Previous Continue to console Skip this step

Accederá automáticamente a la página principal del proyecto de Firebase, que muestra los proyectos de iOS y Android que acaba de agregar.



CÓMO FUNCIONA

Navegue al panel de la consola de Firebase en <https://console.firebaseio.google.com> para agregar o seleccionar proyectos existentes. Cuando agrega un proyecto y le da un nombre, se crea automáticamente una ID de proyecto única con la opción de cambiarle el nombre. Una vez que se crea el proyecto, el ID del proyecto no se puede cambiar. La ubicación del servidor se elige automáticamente, pero tiene la opción de cambiarla.

Una vez que se crea un proyecto de Firebase, registra los proyectos de iOS y Android para usar Firebase. Para registrar cada proyecto, ingresa com.domainname.journal para el ID del paquete de iOS y el nombre del paquete de Android.

Para el proyecto de iOS, descargue el archivo GoogleService-Info.plist y para el proyecto de Android, el archivo googleservices.json . Estos archivos se agregan a la aplicación Flutter en los proyectos Xcode y Android en la sección "Agregar paquetes de autenticación y Cloud Firestore a la aplicación cliente".

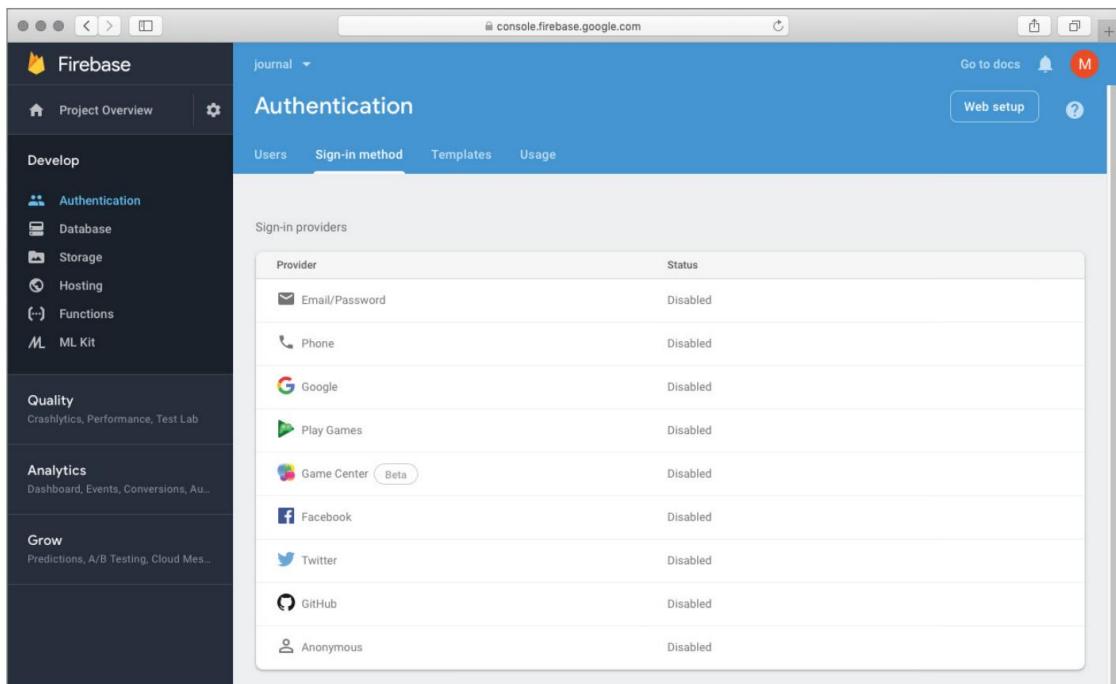
AGREGAR UNA BASE DE DATOS DE CLOUD FIRESTORE Y IMPLEMENTACIÓN DE SEGURIDAD

Ha aprendido a crear un proyecto de Firebase, lo que hace posible agregar la base de datos de Cloud Firestore y la autenticación de Firebase. Aprendió sobre los diferentes métodos de inicio de sesión disponibles y, en esta sección, explicará cómo implementar reglas de seguridad y habilitar un método de inicio de sesión de autenticación, específicamente, un proveedor de autenticación de correo electrónico/contraseña.

PRUÉBELO Crear la base de datos de Cloud Firestore y habilitar la autenticación

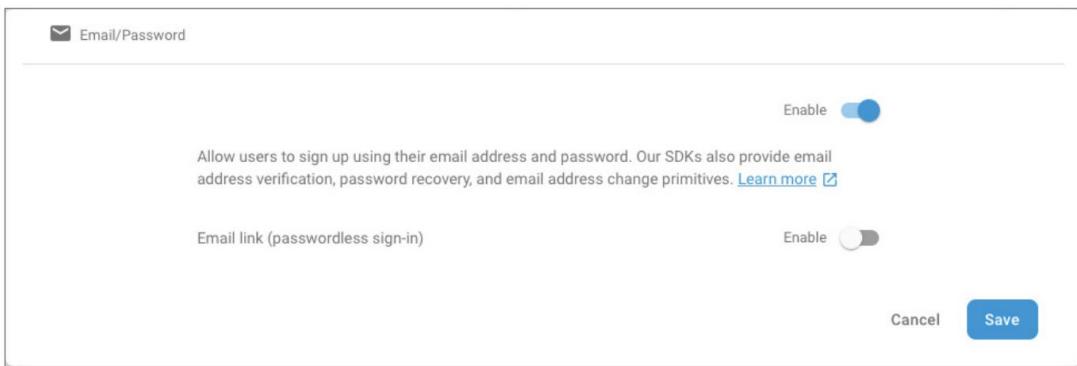
En este ejercicio, aprenderá cómo habilitar la autenticación de Firebase, crear una base de datos de Cloud Firestore e implementar reglas de seguridad para mantener la privacidad de los datos para cada usuario.

1. Navegue a <https://console.firebaseio.google.com> y seleccione el proyecto del diario.
2. En el menú de la izquierda, haga clic en el enlace Autenticación en la sección Desarrollar. Si la sección Desarrollar está cerrada, haga clic en el enlace Desarrollar para abrir el submenú. Haga clic en la pestaña Método de inicio de sesión que muestra una lista de proveedores de inicio de sesión disponibles.



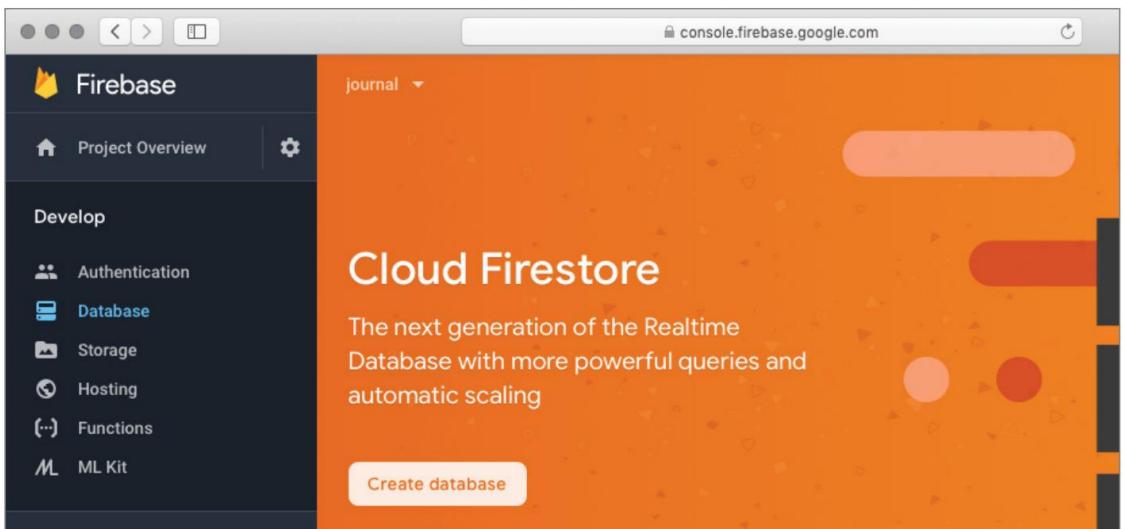
The screenshot shows the Firebase console's Authentication page. On the left sidebar, under the 'Develop' section, 'Authentication' is selected. The main content area is titled 'Authentication' and contains tabs for 'Users', 'Sign-in method', 'Templates', and 'Usage'. Below these tabs, the 'Sign-in providers' section lists various authentication methods. Each provider has a status indicator: Email/Password, Phone, Google, Play Games, Game Center (Beta), Facebook, Twitter, GitHub, and Anonymous, all listed as 'Disabled'.

3. Haga clic en la opción Correo electrónico/Contraseña, haga clic en Habilitar para activar la función y haga clic en el botón Guardar.

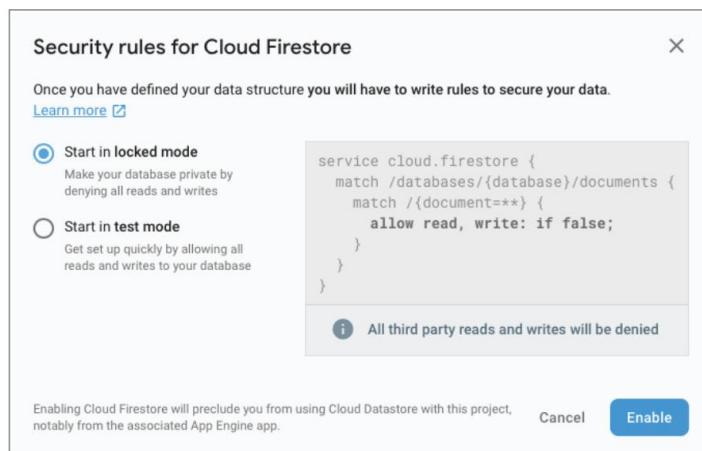


This screenshot shows the configuration for the 'Email/Password' sign-in provider. It features an 'Enable' toggle switch which is turned on (blue). Below the switch, a descriptive text explains that it allows users to sign up with their email address and password, mentioning email address verification, password recovery, and email address change primitives, with a link to 'Learn more'. Another 'Enable' toggle switch for 'Email link (passwordless sign-in)' is shown as off (grey). At the bottom right are 'Cancel' and 'Save' buttons, with 'Save' being highlighted in blue.

4. En el menú de la izquierda, haga clic en el enlace Base de datos y haga clic en el botón Crear base de datos (Cloud Firestore).



5. En el cuadro de diálogo Reglas de seguridad para Cloud Firestore, deje seleccionado el botón de opción de modo bloqueado y haga clic en el botón Habilitar. El modo bloqueado crea las reglas básicas de seguridad con el acceso bloqueado para privilegios de lectura y escritura.



El siguiente código muestra las reglas de seguridad predeterminadas que se crean automáticamente al hacer clic en el botón Habilitar. Las reglas de seguridad actuales niegan todas las solicitudes de lectura y escritura, lo que lo hace completamente seguro pero también inaccesible, lo cual modificará en el paso 6.

```

service cloud.firestore {
    partido /bases
    de datos/{base de datos}/documentos { partido /{document=**}
        { permitir lectura, escritura: si es
            falso;
        }

    }
}

```

6. Toque la pestaña Reglas para editar las reglas bloqueadas predeterminadas y cambiar la coincidencia /{document=**} para coincidir con /journals/{document=**}. {document=**} coincide con cualquier documento, pero utilizará un enfoque granular al hacer coincidir la colección de diarios y el documento con el ID de usuario que inició sesión. Este enfoque le permite restringir cada documento a cada ID de usuario, manteniendo los datos seguros para el propietario legítimo de los datos.
7. Cambiar permitir leer, escribir: si es falso; para permitir la lectura, escriba: if resource.data.uid == request.auth.uid;, restringiendo el uid del campo de datos para que coincida con el uid de inicio de sesión . El recurso .data.uid es el campo uid dentro del documento, y request.auth.uid es el ID único del usuario que ha iniciado sesión. Agregar permitir crear: si request.auth.uid != null; para permitir la creación de nuevos registros si el usuario está autenticado.

```

service cloud.firestore {
    match /
    databases/{database}/documents { match /journals/
        {document=**} { permitir lectura, escritura: if
            resource.data.uid == request.auth.uid; permitir crear: si request.auth.uid != null;

        }

    }
}

```

Tenga en cuenta que se crea automáticamente un registro histórico de cambios, lo que permite revertir los cambios si es necesario.

The screenshot shows the Firebase console interface. On the left, there's a sidebar with 'Project Overview' and sections for 'Develop' (Authentication, Database, Storage, Hosting, Functions, ML Kit) and 'Quality' (Crashlytics, Performance, Test Lab). The main area is titled 'Database' and shows a 'Journal' view. It has tabs for 'Data', 'Rules', 'Indexes', and 'Usage'. Under 'Rules', there's a code editor with the following content:

```

1  service cloud.firestore {
2      match /databases/{database}/documents {
3          match /journals/{document=**} {
4              allow read, write: if resource.data.uid == request.auth.uid;
5              allow create: if request.auth.uid != null;
6          }
7      }
8  }

```

On the left side of the code editor, there's a timeline showing three previous versions of the rules:

- Today • 1:21 PM (the current version)
- Feb 19, 2019 • 1:06 PM (an older version)
- Feb 19, 2019 • 1:06 PM (another older version)
- Feb 4, 2019 • 5:23 PM (the original version)

CÓMO FUNCIONA

Vaya a <https://console.firebaseio.google.com> y seleccione el proyecto del diario. Vaya a la página Autenticación y seleccione la pestaña Método de inicio de sesión para habilitar y deshabilitar los proveedores de inicio de sesión. Navegue a la página Base de datos para crear o editar bases de datos y, en este caso, Cloud Firestore. En la página Base de datos, seleccione la pestaña Reglas para ver o modificar las reglas de seguridad actuales. Con cada edición guardada, el historial de cambios se crea automáticamente, lo que facilita revertir los cambios si es necesario.

CONSTRUYENDO LA APLICACIÓN DEL DIARIO DEL CLIENTE

El proceso de creación de la aplicación de registro de estados de ánimo abarca desde este capítulo hasta el Capítulo 16. En esta sección, creará la estructura base de la aplicación y configurará los proyectos de iOS y Android para usar Firebase Authentication y la base de datos de Cloud Firestore. Modificará la apariencia básica de la aplicación mediante el uso de degradados de color.

El objetivo de la aplicación de diario de estado de ánimo es tener la capacidad de enumerar, agregar y modificar las entradas del diario mediante la recopilación de una fecha, un estado de ánimo, una nota y la identificación del usuario. Aprenderá a crear una página de inicio de sesión para autenticar a los usuarios a través del proveedor de inicio de sesión de Firebase Authentication de correo electrónico/contraseña. La página de presentación principal implementa un widget ListView mediante el uso del constructor separado ordenado por fecha DESC (descendente), lo que significa que el último registro ingresado primero. El widget ListTile formatea fácilmente cómo mostrará la lista de registros. La página de entrada del diario usa el widget showDatePicker para seleccionar una fecha de un calendario, un widget DropdownButton para seleccionar de una lista de estados de ánimo y un widget TextField para ingresar la nota.

Agregar paquetes de autenticación y Cloud Firestore al

Aplicación de cliente

Es hora de crear la aplicación Flutter y agregar los SDK de Firebase Authentication y Cloud Firestore instalando los paquetes Firebase Flutter. El equipo de Flutter crea los diferentes paquetes de Firebase y, al igual que otros paquetes, el código fuente completo del paquete está disponible en la página de GitHub del paquete correspondiente.

Instalará los paquetes `firebase_auth`, `cloud_firestore` e `intl`. Descargará los archivos de servicio de Google (configuración) para iOS y Android que contienen las propiedades necesarias para acceder a los productos de Firebase desde la aplicación cliente.

PRUÉBALO Creación de la aplicación Journal

En este ejemplo, creará una aplicación de diario de nivel de producción similar a la del Capítulo 13, pero en su lugar, utiliza Firebase Authentication para la seguridad y la base de datos de Cloud Firestore para almacenar y sincronizar datos. La estructura de cómo lee y guarda datos es completamente diferente. También agregará seguimiento del estado de ánimo a cada entrada del diario. En este ejercicio, utilizará algunos paquetes nuevos y familiares.

Para agregar seguridad a su aplicación de diario, utilizará el paquete `firebase_auth`, que proporciona autenticación.

Para agregar capacidades de almacenamiento de datos, utilizará el paquete `cloud_firestore`, que proporciona sincronización y almacenamiento en la nube.

Para dar formato a las fechas, utilizará el paquete `intl`, que proporciona internacionalización y localización.

Aprendió a usarlo en el Capítulo 13 en la sección "Dar formato a las fechas".

1. Cree un nuevo proyecto de Flutter y asígnale el nombre diario. Puede seguir las instrucciones del Capítulo 4, "Creación de una plantilla de proyecto inicial".

Tenga en cuenta que, dado que esta aplicación continúa en los próximos dos capítulos, para simplificar las cosas, el nombre del proyecto no comienza con el número de capítulo. Nombrar el diario del proyecto también da como resultado el nombre del paquete `com.domainname.journal`, que coincide con el ID del paquete de iOS y el nombre del paquete de Android. Dado que está utilizando Cloud Firestore, el nombre del paquete debe coincidir exactamente con lo que ingresó cuando registró los proyectos de iOS y Android en la consola de Firebase. Como nota al margen, también puede cambiar manualmente el nombre del paquete al crear un nuevo proyecto de Flutter.

Para este proyecto, debe crear las carpetas de páginas, clases, servicios, modelos y bloques .

2. Abra el archivo `pubspec.yaml` para agregar recursos. En la sección `dependencies`, agregue las declaraciones `firebase_auth:^0.11.1+6` y `cloud_firestore:^0.12.5` e `intl:^0.15.8`. Tenga en cuenta que la versión de su paquete puede ser superior.

dependencias:

aleteo:

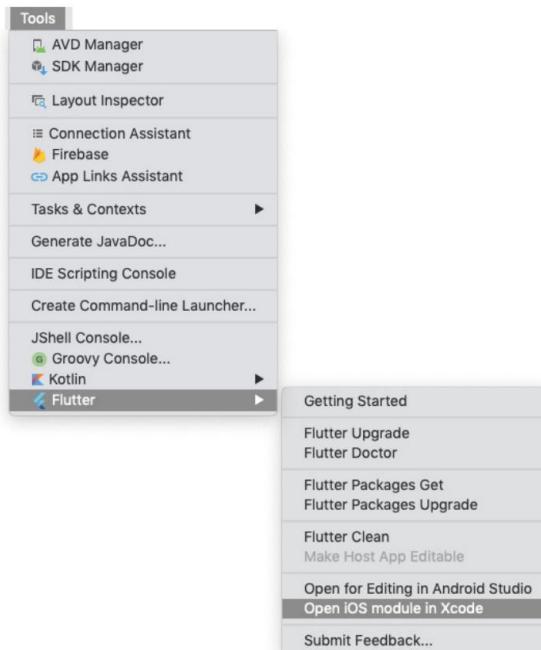
SDK: aleteo

```
# Lo siguiente agrega la fuente Cupertino Icons a su aplicación.  
# Usar con la clase CupertinoIcons para iconos de estilo iOS. icons_cupertino:  
^0.1.2
```

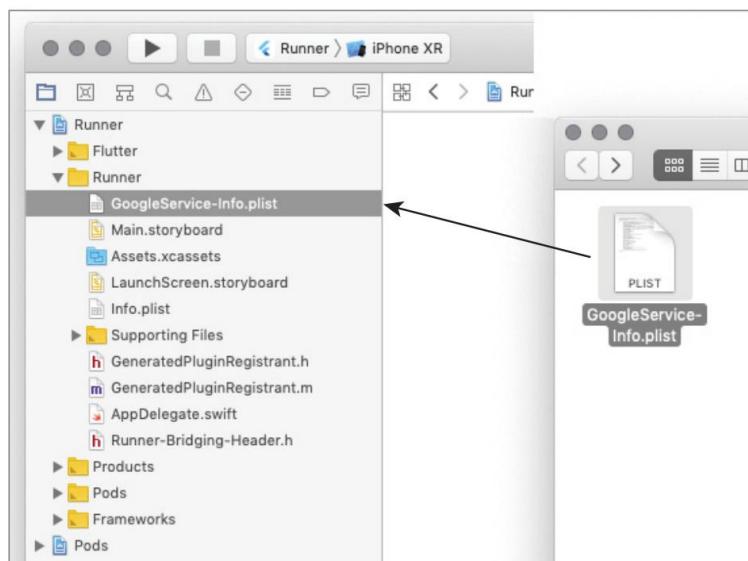
```
firebase_auth: ^0.11.1+6  
cloud_firestore: ^0.12.5  
internacional: ^0.15.8
```

3. Haga clic en el botón Guardar y, según el editor que esté utilizando, se ejecuta automáticamente el paquetes de aleteo obtener; una vez terminado, muestra el mensaje El proceso terminó con el código de salida 0. Si no ejecuta automáticamente el comando por ti, abre la ventana Terminal (ubicada en la parte inferior de tu editor) y escribe `flutter packages get`.

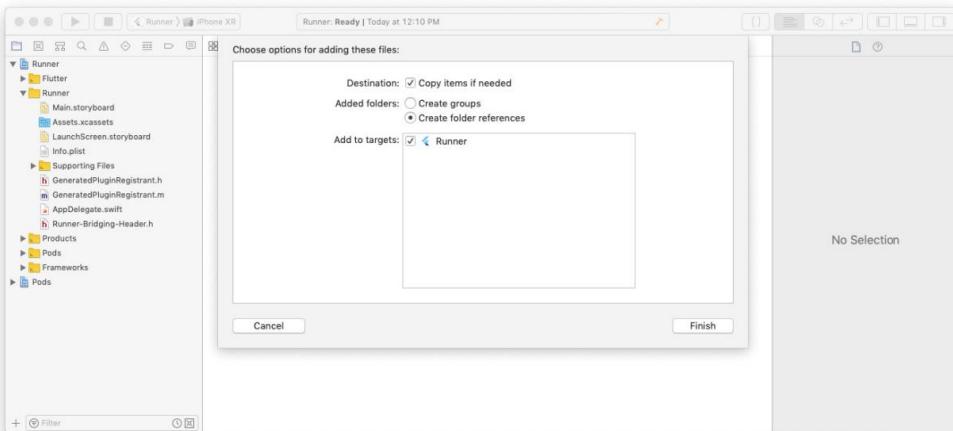
4. Desde el proyecto Flutter, abra el proyecto iOS Xcode para agregar Firebase. Desde Android Studio, haz clic en la barra de menú y selecciona Herramientas → Flutter → Abrir módulo iOS en Xcode.



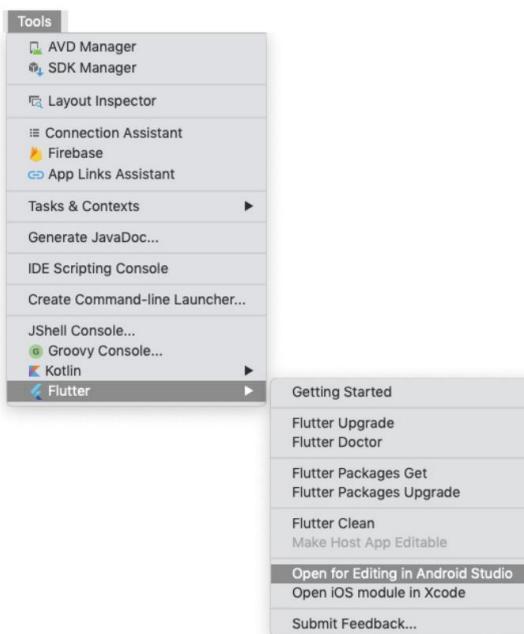
5. Arrastre el archivo GoogleService-Info.plist descargado a la carpeta Runner en el proyecto Xcode.



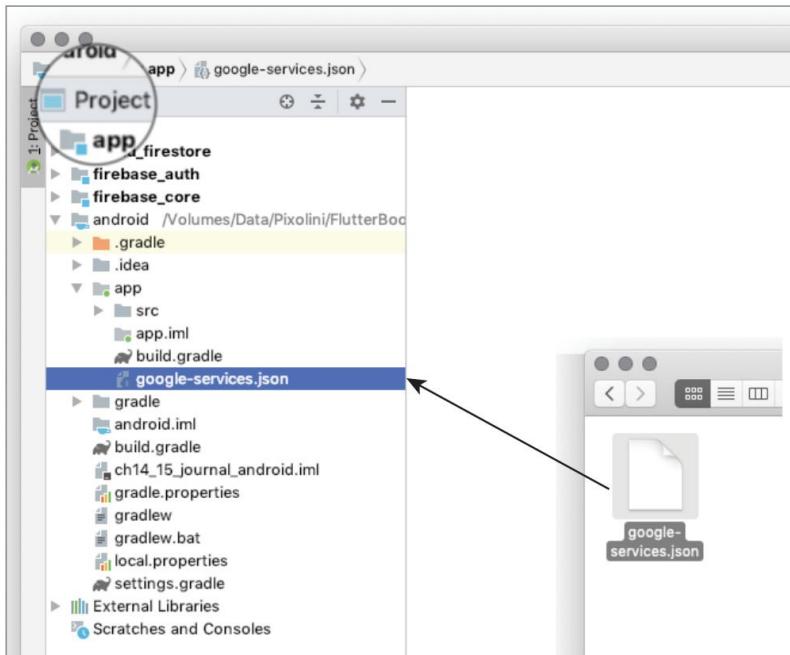
6. En el siguiente cuadro de diálogo, termine de agregar el archivo GoogleService-Info.plist y asegúrese de que Copiar elementos si es necesario esté marcado, el botón de opción Crear referencias de carpeta esté seleccionado y la opción Agregar a destinos Corredor esté marcada. El proyecto iOS Xcode ahora está configurado para manejar Firebase y Firestore. Una vez copiado el archivo, cierre Xcode.



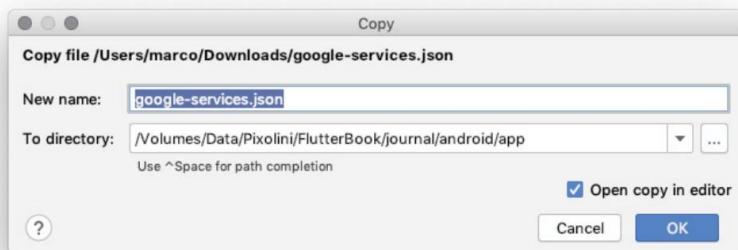
7. Desde el proyecto Flutter, abra el proyecto Android Studio para agregar Firebase. Desde Android Studio, haz clic en la barra de menú y selecciona Herramientas → Flutter → Abrir para editar en Android Studio.



8. Arrastre el archivo google-services.json descargado a la carpeta de la aplicación en el proyecto de Android. Si no ve la carpeta de la aplicación , asegúrese de que en la ventana de herramientas de Android Studio esté seleccionada la vista Proyecto (arriba a la izquierda), no la vista Android.



9. Termine de agregar el archivo google-services.json y haga clic en el botón Aceptar.



10. Para el proyecto de Android, debe editar dos archivos manualmente. Para el primer archivo, abra el archivo de nivel de aplicación build.gradle ubicado en android/app/build.gradle. Agregue al final del archivo el complemento Gradle de servicios de Google especificando aplicar complemento: 'com.google.gms.google services'. En este archivo build.gradle de nivel de aplicación , asegúrese de estar usando compileSdkVersion 28, minSdkVersion 16 y targetSdkVersion 28 y guarde.

11. Para evitar recibir "el archivo .dex no puede exceder el error 64" cuando intenta ejecutar la aplicación Flutter para Android, debe agregar multiDexEnabled true en la sección defaultConfig y guardar.

Tenga en cuenta que es posible que este paso no sea necesario en futuras actualizaciones. Puede revisar la página de la guía del usuario de Android Studio multidex en <https://developer.android.com/studio/build/multidex>.

```
Android
{ compileSdkVersion 28

    conjuntos de fuentes {...}

    lintOpciones {...}

    defaultConfig
        { applicationId "com.domainname.journal"
            minSdkVersion 16
            targetSdkVersion 28 // ...

            // Habilite si obtiene un error en la aplicación Flutter: el archivo .dex no puede exceder los 64 K
            multiDexEnabled true // Habilite
        }

    tipos de compilación {...}
}

aleteo {...}

dependencias
{ // ...}

// Agregar al final del archivo aplicar
complemento: 'com.google.gms.google-services'
```

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply from: "$flutterRoot/packages/flutter_tools/gradle/flutter.gradle"

android {
    compileSdkVersion 28

    sourceSets {
        main.java.srcDirs += 'src/main/kotlin'
    }

    lintOptions {
        disable 'InvalidPackage'
    }

    defaultConfig {
        // TODO: Specify your own unique Application ID (https://developer.android.com/studio/build/application-id.html)
        applicationId "com.domainname.journal"
        minSdkVersion 16
        targetSdkVersion 28
        versionCode flutterVersionCode.toInt()
        versionName flutterVersionName
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
        // Enable if you get error in Flutter app - .dex file cannot exceed 64K
        multiDexEnabled true // Enable
    }

    buildTypes {...}
}

flutter {source '../../'}

dependencies {
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.2'
    androidTestImplementation 'com.android.support.test.espresso:espresso-core:3.0.2'
    // Add if you get error in Flutter app - .dex file cannot exceed 64K
    // The multidex library as a dependency
    implementation 'com.android.support:multidex:1.0.3'
}

apply plugin: 'com.google.gms.google-services'
```

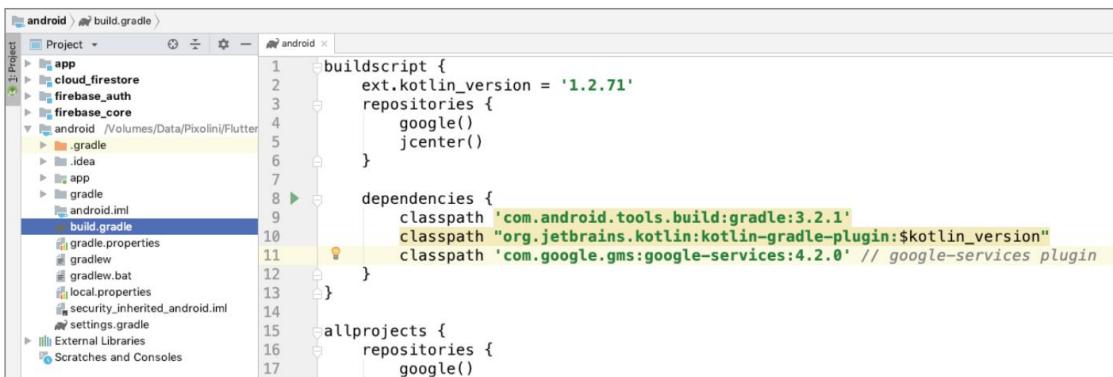
12. Para el segundo archivo, abra el archivo build.gradle a nivel de proyecto ubicado en android/build.gradle.

Agregue a las dependencias el classpath del complemento de servicios de Google y guarde.

```
buildscript { // ...

    dependencies
    { // ...
        // Agregue la siguiente línea:
        classpath 'com.google.gms:google-services:4.2.0' // complemento de googleservices

    }
}
```



13. Para evitar recibir errores de AndroidX, edite el archivo gradle.properties a nivel de proyecto agregando las siguientes dos líneas y guarde el archivo. Tenga en cuenta que es posible que este paso manual no sea necesario en futuras actualizaciones. Habilitar la compatibilidad con AndroidX depende del requisito de complementos.

```

android.useAndroidX=true
android.enableJetifier=true

```

14. Verá una barra amarilla con un aviso de que los archivos gradle han cambiado. Haga clic en Sincronizar ahora botón, y una vez hecho el proceso, cierre el proyecto de Android.



CÓMO FUNCIONA

Declaró los paquetes firebase_auth, cloud_firestore e intl en su archivo pubspec.yaml . El paquete firebase_auth le brinda la posibilidad de usar Firebase Authentication para proteger la aplicación.

El paquete cloud_firestore le brinda la posibilidad de usar la base de datos de Cloud Firestore para sincronizar y almacenar datos en la nube. El paquete intl le brinda la posibilidad de formatear fechas.

Usó Xcode para importar el archivo GoogleService-info.plist al proyecto de iOS. Usó Android Studio para importar el archivo google-services.json al proyecto de Android. Para el proyecto de Android, modificó el proyecto y los archivos build.gradle a nivel de la aplicación para habilitar los complementos de Firebase. Los archivos de servicio de Google importan todas las propiedades necesarias para acceder a los productos del proyecto Firebase.

Desde la consola de Firebase, registró los proyectos de iOS y Android con el ID del paquete de iOS y el nombre del paquete de Android de com.domainname.journal, que coincide exactamente con el nombre del paquete del proyecto de Flutter de com.domainname.journal.

Agregar diseño básico a la aplicación del cliente

En la sección anterior, aprendiste cómo agregar y configurar Firebase Authentication y la base de datos de Cloud Firestore. El siguiente paso es trabajar en el proyecto Flutter para personalizar la apariencia de la aplicación de diario.

Personalizará el color de fondo de la aplicación configurando la propiedad `MaterialApp canvasColor` en un tono verde claro. Las personalizaciones de `AppBar` y `BottomAppBar` muestran un degradado de color de verde claro a un tono de verde muy claro, que se fusiona con el color de fondo de la aplicación. Para lograr el efecto de color, configure la propiedad de degradado `BoxDecoration` usando un widget `LinearGradient` que aprendió en el Capítulo 6, "Uso de widgets comunes".

PRUEBE Agregar un diseño básico a la aplicación Journal

En esta sección, continúe editando el proyecto del diario personalizando los colores y la apariencia de la aplicación, dándole ese aspecto profesional.

1. Abra el archivo `main.dart`. Modifique la propiedad de título de `MaterialApp` a `Diario` y modifique el Propiedad `ThemeData primarySwatch` a `Colors.lightGreen`. Agregue la propiedad `canvasColor` y establezca el color en `Colors.lightGreen.shade50`. Agregue la propiedad `bottomAppBarColor` y establezca el color en `Colors.lightGreen`.

```
return
  MaterialApp( debugShowCheckedModeBanner:
    false, title: 'Journal',
    theme: ThemeData(
      PrimarySwatch: Colors.lightGreen,
      canvasColor: Colors.lightGreen.shade50,
      bottomAppBarColor: Colors.lightGreen, ), home:
        Home(), );
```

2. Abra el archivo `home.dart`, configure el widget de texto de la propiedad de título de `AppBar` en `Journal` y configure el propiedad de color `TextStyle` a `Colors.lightGreen.shade800`.
3. Para personalizar el color de fondo de la barra de aplicaciones con un degradado, elimine la sombra del widget de la barra de aplicaciones configurando la propiedad de elevación en `0.0`. Para aumentar la altura de `AppBar`, establezca la propiedad `inferred` en un widget `PreferredSize` con la propiedad secundaria como un widget `Contenedor` y la propiedad `PreferredSize` en `Size.fromHeight(32.0)`.
4. Establezca la propiedad `flexibleSpace` en un widget de contenedor, con la propiedad de decoración establecida en un Widget de decoración de cajas .
5. Establezca la propiedad de gradiente `BoxDecoration` en un `LinearGradient` con la propiedad de colores establecida en una lista de `[Colors.lightGreen, Colors.lightGreen.shade50]`.

6. Establezca la propiedad de inicio en Alignment.topCenter y la propiedad final en Alignment.bottomCenter. El efecto LinearGradient dibuja el color AppBar de un color verde claro y se desvanece gradualmente a un color verde claro.shade50 .

```
appBar: AppBar(título:  
    Texto('Diario',  
        estilo: TextStyle(color: Colors.lightGreen.shade800)), elevación: 0.0, parte  
    inferior: PreferredSize  
    (  
        hijo: Contenedor(), tamaño preferido: Tamaño.desdeAltura(32.0)), espacio flexible:  
    Contenedor  
        decoración: BoxDecoration(degradado:  
            LinearGradient(  
                colores: [Colores.verde claro, Colores.verde claro.sombra50], comienzo:  
                Alignment.topCenter, fin:  
                Alignment.bottomCenter, ), ), ), ), ),
```



7. Agregue un widget IconButton a la propiedad de acciones de AppBar . Establezca la propiedad de ícono en Iconos .exit_to_app , con la propiedad de color establecida en Colors.lightGreen.shade800 .
8. Agregue a la propiedad onPressed un TODO: comentario con un recordatorio para agregar un método para cerrar la sesión usuario actual.

acciones:

```
<Widget>[ IconButton(icon:  
    Icon(Icons.exit_to_app, color:  
  
        Colors.lightGreen.shade800, ), onPressed:  
        ()  
        { // TODO: Add signOut method }, ),  
    ],
```

9. Agregue la propiedad Scaffold bottomNavigationBar y establezcalo en un widget BottomAppBar . A personalice el color de fondo de BottomAppBar con un degradado, elimine la sombra del widget BottomAppBar configurando la propiedad de elevación en 0.0. Establezca la propiedad secundaria BottomAppBar en un widget de contenedor con la propiedad de altura establecida en 44,0 .

10. Establezca la propiedad de decoración del contenedor en un widget BoxDecoration . Establezca la propiedad de degradado BoxDecoration en LinearGradient , con la propiedad de colores configurada en una lista de [Colors .lightGreen.shade50, Colors.lightGreen].

11. Establezca la propiedad de inicio en Alignment.topCenter y establezca la propiedad final en Alignment .bottomCenter. El efecto LinearGradient dibuja el color AppBar de un verde claro. shade50 y gradualmente se desvanece a un color verde claro .

```
bottomNavigationBar: BottomAppBar( elevación:  
    0.0, hijo:  
    Contenedor( altura:  
        44.0, decoración:  
        BoxDecoration( degradado:  
            LinearGradient(  
                colores: [Colores.verde claro.sombra50, Colores.verde claro], comienzo:  
                Alignment.topCenter, fin:  
                Alignment.bottomCenter, ), ), ), ), ),
```



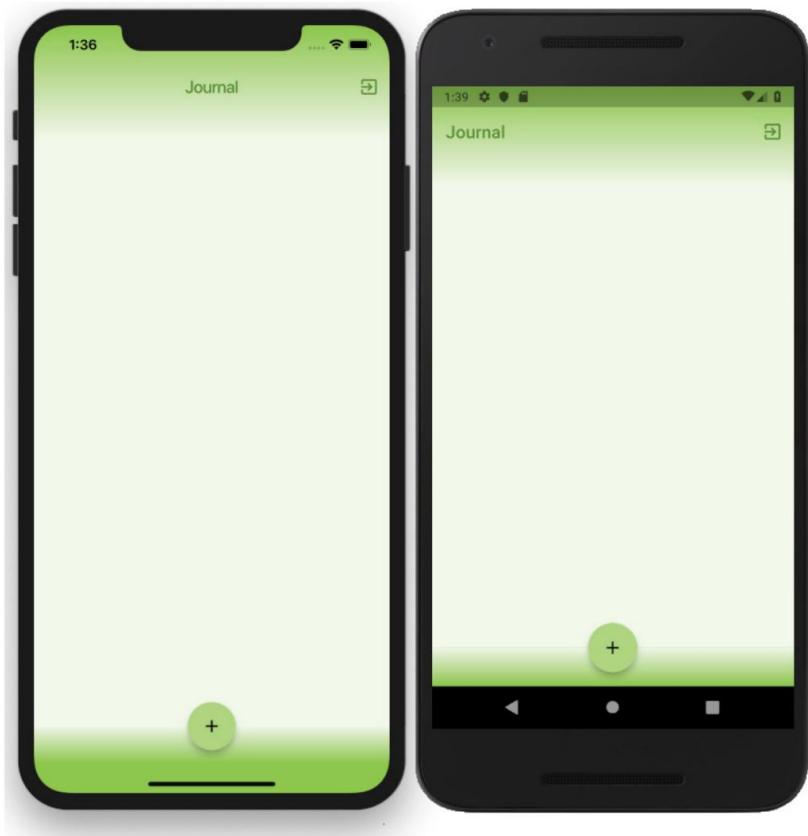
12. Agregue la propiedad Scaffold floatingActionButtonLocation y establezcala en FloatingActionButtonLocation.centerDocked.

```
flotanteActionButtonLocation: FloatingActionButtonLocation.centerDocked,
```

13. El FloatingActionButton es responsable de agregar nuevas entradas de diario. Agregue la propiedad Scaffold floatingActionButton y establezcala en un widget FloatingActionButton , con la propiedad de información sobre herramientas establecida en Agregar entrada de diario y con la propiedad backgroundColor establecida en Colors.lightGreen.shade300. Establezca la propiedad secundaria en Icons.add.

14. Agregue la propiedad FloatingActionButton onPressed y márquela con la palabra clave asíncrona y un TODO: comentario con un recordatorio para agregar un método para agregar entradas de diario.

```
botón de acción flotante: botón de acción flotante (  
    información sobre herramientas:  
    'Agregar entrada de diario', backgroundColor:  
    Colors.lightGreen.shade300,  
    child: Icon(Icons.add),  
    onPressed: () async { // TODO: Agregar método  
        _addOrEditJournal }, ),
```



CÓMO FUNCIONA

Modificó la propiedad `MaterialApp ThemeData canvasColor` a verde claro y una propiedad `bottomAppBar Color` a verde claro. Personalizó los widgets `AppBar` y `BottomAppBar` de la página de inicio para mostrar un degradado de color verde claro. Personalizó la propiedad de degradado `BoxDecoration` para usar un `LinearGradient` para lograr un sombreado de color de degradado suave. Agregó a la propiedad de acciones de `AppBar` un `IconButton` que se usa para cerrar la sesión de un usuario. Para acoplar `FloatingActionButton` al widget `BottomAppBar`, agregó una propiedad `floatingActionButtonLocation` establecida en `centerDocked`. Agregó un widget `FloatingActionButton` que se usa para agregar una nueva entrada de diario.

Adición de clases a la aplicación del cliente

Deberá crear dos clases para manejar el formato de las fechas y el seguimiento de los íconos de estado de ánimo.

La clase `FormatDates` usa el paquete `intl` para dar formato a las fechas. La clase `MoodIcons` almacena referencias al título, el color, la rotación y el ícono de los iconos de estado de ánimo.

PRUÉBALO Agregar las clases FormatDates y MoodIcons

En esta sección, continúe editando el proyecto del diario agregando las clases FormatDates y MoodIcons .

1. Cree un nuevo archivo Dart en la carpeta de clases . Haga clic derecho en la carpeta de clases , seleccione Nuevo Dart Archivo, ingrese FormatDates.dart y haga clic en el botón Aceptar para guardar.
2. Importe el paquete intl.dart y cree la clase FormatDates . Aprendiste a usar el Clase DateFormat en la sección "Formato de fechas" del Capítulo 13. Agregue los métodos dateFormatShortMonth DayYear, dateFormatDayNumber y dateFormatShortDayName .

```
importar 'paquete: intl/intl.dart';

clase Fechas de formato {
    String dateFormatShortMonthDayYear(String date) {
        devuelve DateFormat.yMMMd().format(DateTime.parse(date));
    }

    String dateFormatDayNumber(String date) { return
        DateFormat.d().format(DateTime.parse(date));
    }

    String dateFormatShortDayName(String fecha) {
        return DateFormat.E().format(DateTime.parse(fecha));

    }
}
```

3. Cree un nuevo archivo Dart en la carpeta de clases . Haga clic derecho en la carpeta de clases , seleccione Nuevo Archivo Dart, ingrese mood_icons.dart y haga clic en el botón Aceptar para guardar. Importe el paquete material.dart y cree la clase MoodIcons .

```
iconos de estado de ánimo de clase
{}
```

4. Agregue las variables de título, color, rotación e ícono y márgenes como finales. Dependiendo de estado de ánimo, cada ícono se muestra con un color y una rotación diferentes; por ejemplo, el ícono feliz se gira hacia la izquierda y el ícono triste se gira hacia la derecha.
5. Agregue una nueva línea e ingrese el constructor de MoodIcons con los parámetros nombrados encerrándolos entre corchetes ({}). Haga referencia a cada variable de título, color, rotación e ícono mediante el uso de azúcar sintáctico para acceder a los valores con la palabra clave this , haciendo referencia al estado actual de la clase.

```
clase MoodIcons { título
    final de la cadena; color
    final color; doble rotación
    final; ícono final de IconData;
```

```
const MoodIcons({this.title, this.color, this.rotation, this.icon}); }
```

6. Terminará de agregar métodos a la clase MoodIcons en el paso 8 porque necesitan acceso a la _moodIconsList variable que contiene la lista de iconos que configuran el estado de ánimo. La lista contiene cinco configuraciones de estado de ánimo que contienen título, color, rotación e ícono.

Agregue una línea después de la clase MoodIcons y declare la variable _moodIconsList como List<MoodIcons> utilizando la palabra clave const para aumentar el rendimiento, ya que la lista no cambiará. Inicialice _moodIconsList con la lista de MoodIcons utilizando la palabra clave const .

7. Agregue cinco clases de MoodIcons() usando la palabra clave const y complete el constructor con el siguientes valores:

```
const List<MoodIcons> _moodIconsList = const <MoodIcons>[
    const MoodIcons(título: 'Muy satisfecho', color: Colors.amber, rotación: 0.4,
    ícono: Iconos.sentimiento_muy_satisfecho),
    const MoodIcons(título: 'Satisfecho', color: Colors.green, rotación: 0.2, ícono:
    Iconos.sentimiento_satisfecho),
    const MoodIcons(título: 'Neutro', color: Colors.grey, rotación: 0.0, ícono: Iconos.
    sentimiento_neutro),
    const MoodIcons(título: 'Insatisfecho', color: Colors.cyan, rotación: -0.2, ícono: Icons.sentiment_dissatisfied),
    const MoodIcons(título: 'Muy insatisfecho',
    color: Colors.red, rotación: -0.4, ícono: Iconos.sentimiento_muy_insatisfecho), ];
```

8. Regrese a la clase MoodIcons y agregue getMoodIcons, getMoodColor, getMoodRota

tion y getMoodIconsList métodos para recuperar los valores de atributo de ícono apropiados. Los primeros tres métodos usan el método indexWhere del objeto List para encontrar los atributos de ícono coincidentes . El último método devuelve la lista completa de iconos de estado de ánimo.

```
IconData getMoodIcon(String estado de ánimo) {
    return _moodIconsList[_moodIconsList.indexWhere((ícono) => ícono.title == estado de ánimo)].icon; }
```

```
Color getMoodColor(String humor) {
    return _moodIconsList[_moodIconsList.indexWhere((ícono) => ícono.title == humor)].color; }
```

```
doble getMoodRotation(String estado de ánimo) {
    return _moodIconsList[_moodIconsList.indexWhere((ícono) => ícono.title == estado de ánimo)].rotation; }
```

```
List<MoodIcons> getMoodIconsList() { return
    _moodIconsList;
}
```

RESUMEN

En este capítulo, aprendió cómo conservar y proteger los datos durante los lanzamientos de aplicaciones mediante el uso de Firebase, Firebase Authentication y Cloud Firestore de Google. Firebase es la infraestructura que no requiere que el desarrollador configure o mantenga servidores backend. La plataforma Firebase le permite conectarse y compartir datos entre iOS, Android y aplicaciones web. El proyecto de Firebase se configura con la consola web en línea. En este capítulo, registró los proyectos de iOS y Android con el nombre del paquete com.domainname.journal para conectar la aplicación cliente a los productos de Firebase.

Creó una base de datos de Cloud Firestore que almacena de forma segura los datos de la aplicación cliente en una base de datos en la nube. Cloud Firestore es una base de datos de documentos NoSQL para almacenar, consultar y sincronizar datos con soporte sin conexión para aplicaciones móviles y web. Las bases de datos de Cloud Firestore se estructuran y modelan mediante el uso de una colección para almacenar documentos que contienen datos como un par clave-valor similar a JSON.

La protección de la base de datos de Cloud Firestore se realiza mediante la creación de reglas de seguridad de Cloud Firestore. Las reglas de seguridad consisten en utilizar sentencias de coincidencia para identificar documentos, con la expresión allow para controlar el acceso.

Firebase Authentication proporciona servicios de backend integrados a los que se puede acceder desde el SDK del cliente, que admite la autenticación completa del usuario. Habilitar el proveedor de inicio de sesión de autenticación de correo electrónico/contraseña permite a los usuarios registrarse e iniciar sesión en la aplicación. Al pasar las credenciales de usuario (correo electrónico/contraseña) al SDK de autenticación de Firebase del cliente, los servicios de backend de Firebase verifican si las credenciales son válidas y devuelven una respuesta a la aplicación del cliente.

Creó la estructura base de la aplicación de registro de estado de ánimo del cliente y la conectó a los servicios de Firebase mediante la instalación de los paquetes firebase_auth y cloud_firestore . Configuró los proyectos cliente de iOS y Android para usar Firebase. Agregó el archivo GoogleService-Info.plist al proyecto de iOS y el archivo google-services.json al proyecto de Android. Los archivos de servicio de Google contienen las propiedades necesarias para acceder a los productos de Firebase desde la aplicación del cliente. Modificó la apariencia básica de la aplicación mediante el uso del widget BoxDecoration , con la propiedad de degradado establecida en un degradado lineal para crear un degradado de color verde claro suave.

En el próximo capítulo, aprenderá a implementar la administración de estado local y en toda la aplicación mediante el uso de la clase InheritedWidget y cómo maximizar el uso compartido y la separación del código de la plataforma mediante la implementación del patrón Business Logic Component. Utilizará la administración de estado para implementar la autenticación de Firebase, acceder a la base de datos de Cloud Firestore e implementar clases de servicio.

LO QUE APRENDISTE EN ESTE CAPÍTULO

TEMA	CONCEPTOS CLAVE
Base de fuego de Google	Firebase consta de una plataforma con muchos productos que funcionan juntos como una infraestructura de servidor back-end que conecta iOS, Android y aplicaciones web.
Tienda de fuego en la nube	Aprendió a estructurar y modelar datos de la base de datos de Cloud Firestore. Cloud Firestore es una base de datos de documentos NoSQL para almacenar, consultar y sincronizar datos con soporte sin conexión.
Autenticación de base de fuego	Firebase Authentication proporciona servicios de backend integrados a los que se puede acceder desde el SDK del cliente, lo que admite la autenticación completa.
Tienda de fuego en la nube	Para proteger el acceso a los datos, aprendió a implementar las reglas de seguridad de Cloud Firestore. Las reglas consisten en utilizar las sentencias de coincidencia para identificar documentos con las expresiones allow .
Colección Cloud Firestore	Una colección solo puede contener documentos.
Documento de Cloud Firestore	Un documento es un par clave-valor y, opcionalmente, puede apuntar a subcolecciones. Los documentos no pueden apuntar a otro documento y deben almacenarse en colecciones.
paquete firebase_auth	Aprendió a agregar el paquete firebase_auth para habilitar la autenticación en la aplicación Flutter.
paquete cloud_firestore	Aprendió a agregar el paquete cloud_firestore para proporcionar almacenamiento de bases de datos y sincronización en la nube en la aplicación Flutter.
Archivo GoogleService-Info .plist	Aprendió a agregar el archivo google-services.json al proyecto iOS Xcode para conectar la aplicación cliente a los servicios de Firebase.
archivo .json de servicios de google	Aprendió a agregar el archivo google-services.json al proyecto de Android para conectar la aplicación cliente a los servicios de Firebase. Para el proyecto de Android, también aprendiste cómo modificar el proyecto y los archivos Gradle a nivel de la aplicación para usar Firebase Authentication y Cloud Firestore.
Diseño base de la aplicación Flutter	Aprendió a usar el widget BoxDecoration configurando la propiedad de gradiente en una lista de colores LinearGradient para mejorar la apariencia de la aplicación.

15

Agregar administración de estado a la aplicación Firestore Client

LO QUE APRENDERÁS EN ESTE CAPÍTULO

Cómo usar la administración de estado para controlar la autenticación de Firebase y la base de datos de Cloud Firestore

Cómo utilizar el patrón BLoC para separar la lógica empresarial

Cómo usar la clase InheritedWidget como proveedor para administrar y pase el estado

Cómo implementar clases abstractas

Cómo usar StreamBuilder para recibir los últimos datos de Firebase Autenticación y la base de datos de Cloud Firestore

Cómo usar StreamController, Stream y Sink para manejar eventos de datos de Firebase Authentication y Cloud Firestore

Cómo crear clases de servicio para manejar la autenticación de Firebase y llamadas a la API de Cloud Firestore con clases Stream y Future

Cómo crear una clase modelo para entradas de diario individuales y convertir Cloud Firestore QuerySnapshot y asignarla al clase de diario

Cómo usar la Transacción Firestore opcional para guardar datos en el Base de datos de Firestore

Cómo crear una clase para manejar íconos de estado de ánimo, descripciones y rotación

Cómo crear una clase para manejar el formato de fecha

Cómo usar el constructor con nombre ListView.separated

En este capítulo, continuará editando la aplicación de registro de estados de ánimo creada en el Capítulo 14. Para su comodidad, puede usar el proyecto ch14_final_journal como punto de partida y asegúrese de agregar su archivo GoogleService-Info.plist al proyecto Xcode. y el archivo google-services.json al proyecto de Android que descargó en el Capítulo 14 desde su consola Firebase.

Aprenderá a implementar la administración de estado local y de toda la aplicación que utiliza la clase InheritedWidget como proveedor para administrar y pasar el estado entre widgets y páginas.

Aprenderá a usar el patrón Business Logic Component (BLoC) para crear clases BLoC, por ejemplo, administrar el acceso a las clases de servicio de base de datos Firebase Authentication y Cloud Firestore.

Aprenderá a usar un enfoque reactivo usando StreamBuilder, StreamController y Stream para completar y actualizar datos.

Aprenderá a crear una clase de servicio para administrar la API de autenticación de Firebase implementando una clase abstracta que administra las credenciales de inicio de sesión del usuario. Creará una clase de servicio separada para manejar la API de la base de datos de Cloud Firestore. Aprenderá a crear una clase de modelo Journal para manejar la asignación de Cloud Firestore QuerySnapshot a registros individuales. Aprenderá a crear una clase de iconos de estado de ánimo para gestionar una lista de iconos de estado de ánimo, una descripción y una posición de rotación de iconos según el estado de ánimo seleccionado. Aprenderá cómo crear una clase de formato de fecha usando el paquete intl .

IMPLEMENTACIÓN DE LA GESTIÓN ESTATAL

Antes de sumergirse en la administración estatal, echemos un vistazo a lo que significa estado. En su forma más básica, el estado son datos que se leen sincrónicamente y pueden cambiar con el tiempo. Por ejemplo, el valor de un widget de texto se actualiza para mostrar la última puntuación del juego y el estado del widget de texto es el valor. La gestión de estado es la forma de compartir datos (estado) entre páginas y widgets.

Puede tener una administración de estado en toda la aplicación para compartir el estado entre diferentes páginas. Por ejemplo, el administrador de estado de autenticación supervisa al usuario que ha iniciado sesión y, cuando el usuario cierra sesión, toma la acción adecuada para redirigirlo a la página de inicio de sesión. La Figura 15.1 muestra la página de inicio obteniendo el estado de la página principal; esta es la gestión estatal de toda la aplicación.

Puede hacer que la administración del estado local se limite a una sola página o un solo widget. Por ejemplo, la página muestra un artículo seleccionado y el botón de compra debe habilitarse solo si el artículo está en stock.

El widget de botón necesita acceder al estado del valor en stock. La figura 15.2 muestra un botón Agregar que sube de estado en el árbol de widgets; esto es gestión local-estatal.

Hay muchas técnicas diferentes para manejar la administración del estado, y no hay una respuesta correcta o incorrecta sobre qué enfoque tomar porque depende de sus necesidades y preferencias personales. La belleza es que puede crear un enfoque personalizado para la gestión del estado. Ya domina una de las técnicas de gestión de estado, el método setState() . En el Capítulo 2, aprendió a usar StatefulWidget y a llamar al método setState() para propagar los cambios en la interfaz de usuario. Usar el método setState() es la forma predeterminada en que una aplicación Flutter administra los cambios de estado, y lo ha usado para todas las aplicaciones de ejemplo que ha creado a partir de este libro.

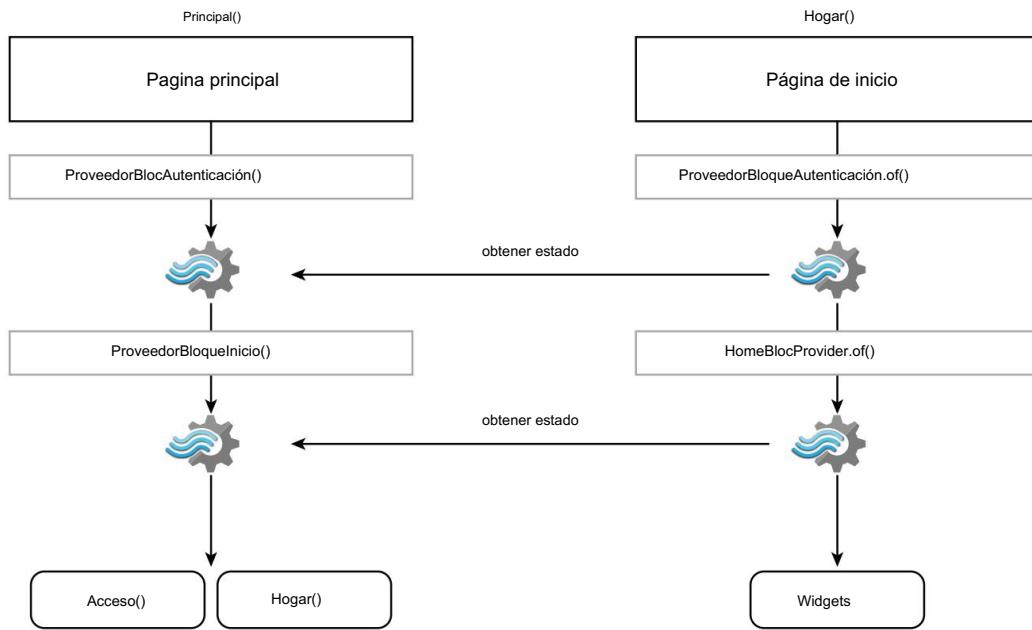


FIGURA 15.1: Gestión de estado de toda la aplicación

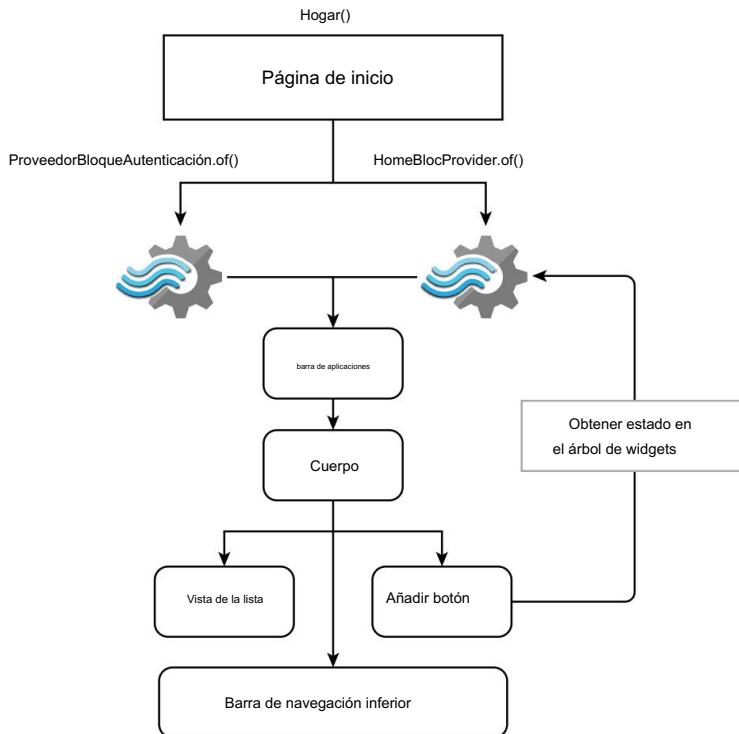


FIGURA 15.2: Gestión local-estatal

Para mostrar diferentes enfoques de administración de estado, la aplicación de diario utiliza una combinación de una clase abstracta y una clase de widget heredado como proveedores, además de una clase de servicio, una clase de utilidad de estado de ánimo, una clase de utilidad de fecha y el patrón BLoC para separar la lógica del código comercial. de la interfaz de usuario.

Implementación de una clase abstracta Uno de los principales

beneficios de usar una clase abstracta es separar los métodos de la interfaz (llamados desde la interfaz de usuario) de la lógica del código real. En otras palabras, declara los métodos sin ninguna implementación (código). Otro beneficio es que la clase abstracta no se puede instanciar directamente, lo que significa que no se puede crear un objeto a menos que defina un constructor de fábrica. Las clases abstractas lo ayudan a programar las interfaces, no la implementación. Las clases concretas implementan los métodos de la clase abstracta.

Por defecto, las clases (concretas) definen una interfaz que contiene todos los miembros y métodos que implementa. El siguiente ejemplo muestra la clase AuthenticationService declarando una variable y métodos que contienen la lógica del código:

```
servicio de autenticación de clase {  
    final FirebaseAuth _firebaseAuth = FirebaseAuth.instance;  
  
    Future<void> sendEmailVerification() asincrónico {  
        FirebaseUser usuario = esperar _firebaseAuth.currentUser();  
        usuario.sendEmailVerification();  
    }  
  
    Future<bool> isEmailVerified() async { FirebaseUser user  
        = await _firebaseAuth.currentUser(); volver usuario.isEmailVerified;  
  
    }  
}
```

En esta sección , utilizará una clase abstracta para definir su interfaz de autenticación. La clase abstracta tiene métodos a los que se puede llamar sin contener el código real (implementación), y se denominan métodos abstractos. Para declarar una clase abstracta , usa el modificador abstracto antes de la declaración de clase, como la autenticación de clase abstracta {}. Los métodos abstractos funcionan con una clase que implementa una o más interfaces y se declara utilizando la cláusula implements como este ejemplo: class AuthenticationService implements Authentication {}. Tenga en cuenta que el método abstracto se declara usando un punto y coma (;) en lugar del cuerpo declarado por corchetes ({}). La lógica del código para los métodos abstractos se implementa (implementación concreta) en la clase que implementa la clase abstracta .

El siguiente ejemplo declara la clase de autenticación como una clase abstracta utilizando el modificador abstracto y contiene dos métodos abstractos. La clase AuthenticationService usa la cláusula implements para implementar los métodos declarados por la clase Authentication .

```
Autenticación de clase abstracta {  
    Future<void> sendEmailVerification();  
    Future<bool> isEmailVerified();  
}
```

```
clase AuthenticationService implementa autenticación {  
    final FirebaseAuth _firebaseAuth = FirebaseAuth.instance;  
  
    Future<void> sendEmailVerification() asíncrono {  
        FirebaseUser usuario = esperar _firebaseAuth.currentUser();  
        usuario.sendEmailVerification();  
    }  
  
    Future<bool> isEmailVerified() async { FirebaseUser  
        user = await _firebaseAuth.currentUser(); volver usuario.isEmailVerified;  
  
    }  
}
```

¿Por qué usar una clase abstracta en lugar de declarar una clase con variables y métodos? Por supuesto, puede usar la clase sin crear una clase abstracta para la interfaz ya que la clase ya los declara por defecto. Pero uno de los beneficios de usar una clase abstracta es imponer restricciones de implementación y diseño.

Para la aplicación de diario, los principales beneficios de usar clases abstractas son usarlas en las clases BLoC, y con la inserción de dependencias , inyecta las clases de implementación dependientes de la plataforma, lo que hace que las clases BLoC sean independientes de la plataforma. La inyección de dependencia es una forma de hacer que una clase sea independiente de sus dependencias. La clase no contiene código específico de la plataforma (bibliotecas), pero se pasa (inyecta) en tiempo de ejecución. Aprenderá sobre el patrón BLoC en la sección "Implementación del patrón BLoC" de este capítulo.

Implementando el widget heredado

Una de las formas de pasar Estado entre páginas y el árbol de widgets es usar la clase InheritedWidget como proveedor. Un proveedor tiene un objeto y lo proporciona a sus widgets secundarios. Por ejemplo, usa la clase InheritedWidget como proveedor de una clase BLoC responsable de realizar llamadas de API a la API de autenticación de Firebase. Como recordatorio, el patrón BLoC se trata en la sección "Implementación del patrón BLoC". En el ejemplo que veremos ahora, cubro cómo usar la clase de widget heredado con una clase BLoC, pero también podría usarla con una clase de servicio regular. Aprenderá a crear clases de servicio en la sección "Implementación de la clase de servicio".

Para la aplicación de diario, la relación entre la clase InheritedWidget y la clase BLoC es uno a uno, lo que significa una clase InheritedWidget para una clase BLoC. Usará of(context) para obtener una referencia a la clase BLoC; por ejemplo, AuthenticationBlocProvider.of(context). bloque de autenticación.

El siguiente ejemplo muestra la clase AuthenticationBlocProvider que amplía (subclases) la clase InheritedWidget . La variable BLoC authenticationBloc se marca como final, lo que hace referencia a la clase AuthenticationBloc BLoC. El constructor AuthenticationBlocProvider toma una clave, un widget y la variable this.authenticationBloc .

```
// Authentication_provider.dart clase  
AuthenticationBlocProvider extiende InheritedWidget {  
    bloque de autenticación final bloque de autenticación;
```

```
const AuthenticationBlocProvider({Key key, Widget child, this.authenticationBloc})  
: super(clave: clave, hijo: hijo);  
  
AuthenticationBlocProvider estático de (contexto BuildContext) {  
    devolver (context.inheritFromWidgetOfExactType(AuthenticationBlocProvider) como  
proveedor de bloque de autenticación);  
}  
  
@override  
bool updateShouldNotify(AuthenticationBlocProvider old) => authenticationBloc != old.authenticationBloc; }
```

Para acceder a la clase AuthenticationBlocProvider desde una página, utiliza el método of() . Cuando se carga una página, se debe llamar a InheritedWidget desde el método didChangeDependencies , no desde el método initState . Si los valores heredados cambian, no se volverán a llamar desde initState , pero para asegurarse de que el widget se actualice cuando cambien los valores, debe usar el método didChangeDependencies .

```
// página.dardo  
@override  
void hizoCambiarDependencias() {  
    super.didChangeDependencies();  
    _authenticationBloc = AuthenticationBlocProvider.of(contexto).authenticationBloc; }  
  
// Cerrar sesión de un usuario desde un widget de botón  
_authenticationBloc.logoutUser.add(true);
```

Implementando la clase modelo

La clase modelo es responsable de modelar la estructura de datos. El modelo de datos representa la estructura de datos de cómo se almacenan los datos en la base de datos o el almacenamiento de datos. La estructura de datos declara el tipo de datos para cada variable como String o Boolean. También puede implementar métodos que realicen una función específica como mapear datos de un formato a otro.

El siguiente ejemplo muestra una clase de modelo que declara la estructura de datos y un método para mapear y convertir datos:

```
diario de clase {  
    Cadena documentoID;  
    Fecha de cadena;  
  
    Diario({ este.documentoID,  
             esta.fecha  
        });  
  
    fábrica Diario.fromDoc(documento dinámico) => Diario(documentoID:  
                doc.documentoID, fecha: doc["fecha"]  
  
            );  
}
```

Implementación de la clase de servicio

La aplicación de diario usa Firebase Authentication para verificar las credenciales de los usuarios y la base de datos de Cloud Firestore para almacenar datos en la nube. Los servicios se invocan realizando la llamada API adecuada.

Crear una clase para agrupar todos los servicios del mismo tipo es una excelente opción. Otro beneficio de separar los servicios en clases de servicio es que esto facilita la creación de clases separadas para implementar servicios adicionales o alternativos. Para la aplicación de diario, aprenderá cómo implementar las clases de servicio como clases abstractas, pero para este ejemplo, quería mostrarle cómo implementar la clase de servicio básica.

El siguiente ejemplo muestra una clase DbFirestoreService que implementa métodos para llamar al API de base de datos de Cloud Firestore:

```
clase DbFirestoreService
    { Firestore _firestore = Firestore.instance; String
        _colecciónDiarios = 'diarios';

    DbFirestoreService()
        { _firestore.settings(timestampsInSnapshotsEnabled: true); }

    Corriente<Lista<Diario>> getJournalList(String uid) {
        return

        _firestore .collection(_colecciónJournals) .where('uid', isEqualTo:
        uid) .snapshots() .map((QuerySnapshot snapshot) { List<Journal> _journalDocs =
            snapshot.documents.map((doc) => Journal .fromDoc (doc)).toList(); _journalDocs.sort((comp1, comp2) =
        }

        Future<bool> addJournal(Diario diario) asíncrono {} void
        updateJournal(Diario diario) asíncrono {} void
        updateJournalWithTransaction(Diario diario) asíncrono {} void
        deleteJournal(Diario diario) asíncrono {} }
```

Implementando el Patrón BLoC

BLoC significa Business Logic Component, y fue concebido para definir una interfaz independiente de la plataforma para la lógica empresarial. El patrón BLoC fue desarrollado internamente por Google para maximizar el uso compartido de código entre las aplicaciones web Flutter mobile y AngularDart. Fue presentado públicamente por primera vez por Paolo Soares en la conferencia DartConf 2018. El patrón BLoC expone flujos y sumideros para manejar el flujo de datos, lo que hace que el patrón sea reactivo. En su forma más simple, la programación reactiva maneja el flujo de datos con flujos asíncronos y la propagación de cambios de datos. El patrón BLoC omite cómo se implementa el almacén de datos; eso depende del desarrollador para elegir de acuerdo con los requisitos del proyecto. Al separar la lógica empresarial, no importa qué infraestructura utilice para crear su aplicación; otras partes de la aplicación pueden cambiar y la lógica comercial permanece intacta.

Paolo Soares compartió las siguientes pautas de patrones BLoC en la conferencia DartConf 2018.

El patrón BLoC tiene pautas de diseño a las que se debe adherir:

Las entradas y salidas son solo flujos y sumideros

Dependencias e independientes de la plataforma deben ser inyectables

No se permite la bifurcación de la

plataforma La implementación depende del desarrollador, como la programación reactiva

El patrón BLoC tiene pautas de diseño de interfaz de usuario que cumplir:

Cree un BLoC para cada componente lo suficientemente complejo

Los componentes deben enviar entradas tal cual

Los componentes deben mostrar los resultados lo más cerca posible de lo que es.

Todas las bifurcaciones deben basarse en salidas booleanas BLoC simples

Aprenderá a usar InheritedWidget como proveedor para acceder y pasar las clases BLoC entre páginas. También aprenderá a crear instancias de un BLoC sin un proveedor para páginas que no requieren compartir una referencia entre ellas, por ejemplo, la página de inicio de sesión.

A continuación, se resumen las pautas de patrones de BLoC presentadas en la conferencia DartConf 2018.

Mover la lógica empresarial a BLoC

Mantenga los componentes de la interfaz de usuario simples

Las reglas de diseño no son negociables

Echemos un vistazo a una estructura de clases BLoC. Aunque no es obligatorio, nombro la clase con un nombre descriptivo y la palabra Bloc como HomeBloc. El siguiente ejemplo de la clase HomeBloc maneja las llamadas de la base de datos a la clase de servicio de la API DbFirestoreService , recupera la lista de entradas de diario convertidas y luego las envía de regreso al widget (IU). DbFirestoreService se inyecta en el constructor HomeBloc() , lo que lo hace independiente de la plataforma. En la clase HomeBloc , la clase abstracta DbApi es independiente de la plataforma y recibe la clase DbFirestoreService inyectada . El componente de lógica de negocios procesa, formatea y envía la salida de vuelta al widget, y la aplicación cliente receptora puede ser móvil, web o de escritorio, lo que resulta en una separación de la lógica de negocios y un uso compartido máximo de código entre plataformas.

El siguiente ejemplo muestra cómo la clase DbFirestoreService() dependiente de la plataforma se inyecta en el constructor HomeBloc(DbFirestoreService()) , lo que da como resultado que la clase HomeBloc() permanezca independiente de la plataforma.

```
// Inyectar DbFirestoreService() a HomeBloc() desde la página de widgets de UI // mediante inyección de  
dependencia  
BloqueInicio(DbFirestoreService());
```

```
// Clase de patrón BLoC // El  
constructor HomeBloc(this.dbApi) recibe // la clase DbFirestoreService()  
inyectada
```

```

clase HomeBloc
    { final DbApi dbApi;

        final StreamController<Lista<Diario>> _journalController =
        StreamController<Lista<Diario>>();
        Sink<Lista<Diario>> get _addJournal => _journalController.sink;
        Stream<List<Journal>> get listJournal => _journalController.stream;

        // Constructor
        HomeBloc(this.dbApi)
        { _startListeners(); }

        // Cierra StreamControllers cuando ya no los necesites void
        dispose()
        { _journalController.close(); }

        void _startListeners() {
            // Recuperar registros de diario de Firestore como Lista<Diario> no DocumentSnapshot
            dbApi.getJournalList().listen((journalDocs)
            { _addListJournal.add(journalDocs); });

        }
    }
}

```

Implementación de StreamController, Streams, Sinks y StreamBuilder

El StreamController es responsable de enviar datos, eventos realizados y errores en la propiedad de transmisión . StreamController tiene una propiedad de receptor (entrada) y una propiedad de flujo (salida). Para agregar datos a la transmisión, utiliza la propiedad del receptor y, para recibir datos de la transmisión, escucha los eventos de la transmisión configurando oyentes. La clase Stream son eventos de datos asincrónicos y la clase Sink permite agregar valores de datos a la propiedad de flujo StreamController .

El siguiente ejemplo muestra cómo usar la clase StreamController . Utiliza la clase Sink para agregar datos con la propiedad Sink y la clase Stream para enviar datos con la propiedad Stream de StreamController. Tenga en cuenta el uso de la palabra clave get para las declaraciones _addUser (Sink) y user (Stream) . La palabra clave get se denomina captador y es un método especial que proporciona acceso de lectura y escritura a las propiedades del objeto.

```

final StreamController<String> _authController = StreamController<String>(); Sink<String> get
_addUser => _authController.sink; Stream<String> get usuario
=> _authController.stream;

```

Para agregar datos a la propiedad de flujo , utilice el método add(event) de la propiedad del receptor .

```
_addUser.add(estado de ánimo);
```

Para escuchar eventos de transmisión , utiliza el método listen() para suscribirse a la transmisión. También usa el widget StreamBuilder para escuchar eventos de transmisión .

```

_authController.listen((estado de ánimo)
{ print('Mi estado de ánimo:
$estado de ánimo'); })

```

Cuando necesite varios oyentes para la propiedad de transmisión StreamController , use el constructor de transmisión StreamController . Por ejemplo, podría tener un widget StreamBuilder y un oyente escuchando la misma clase StreamController .

```
final StreamController<String> _authController = StreamController<String> .broadcast();
```

El widget StreamBuilder se reconstruye a sí mismo en función de la última instantánea de nuevos eventos de una clase Stream , y lo usará para crear sus widgets reactivos para mostrar datos. En otras palabras, Stream Builder se reconstruye cada vez que recibe un nuevo evento de una transmisión.

```
StreamBuilder(initialData:  
    ", flujo: usuario,  
  
    constructor: (contexto BuildContext, instantánea AsyncSnapshot) { return Text('Hello  
    $snapshot.data'); },  
  
)
```

Utilice la propiedad initialData para configurar la instantánea de datos inicial antes de que la transmisión envíe los eventos de datos más recientes. Siempre se llama al constructor antes de que el detector de flujo tenga la oportunidad de procesar los datos y, al configurar initialData , se muestra un valor predeterminado en lugar de un valor en blanco.

datos iniciales: ",

La propiedad de flujo se establece en el flujo responsable de los eventos de datos más recientes, por ejemplo, la propiedad de flujo StreamController .

corriente: usuario,

La propiedad del constructor se usa para agregar su lógica para construir un widget de acuerdo con los resultados de los eventos de transmisión de datos. La propiedad del constructor toma los parámetros BuildContext y AsyncSnapshot . La instantánea asíncrona contiene la información de conexión y datos que aprendió en la sección "Recuperación de datos con FutureBuilder" del Capítulo 13. Asegúrese de que el constructor devuelva un widget; de lo contrario, recibirá un error de que las funciones de compilación devolvieron un valor nulo. En Flutter, las funciones de compilación nunca deben devolver un valor nulo .

```
constructor: (contexto BuildContext, instantánea AsyncSnapshot) { return Text('Hello  
    $snapshot.data'); },
```

El siguiente ejemplo muestra cómo usar el widget StreamBuilder para cambiar de forma reactiva el widget de la interfaz de usuario según el valor de la propiedad de flujo . Esta programación reactiva da como resultado una mejora del rendimiento, ya que solo este widget en el árbol de widgets se reconstruye para volver a dibujar un nuevo valor cuando cambia la transmisión. Tenga en cuenta que la secuencia de usuario está configurada a partir del ejemplo anterior de StreamController .

```
StreamBuilder(initialData:  
    ", flujo: usuario,  
  
    constructor: (contexto BuildContext, instantánea AsyncSnapshot) {  
        if (instantánea.connectionState == ConnectionState.esperando) {  
            contenedor de retorno (color: Colors.yellow,); } else if  
(instantánea.hasData) { return Container(color:  
            Colors.lightGreen,); } else { return Container(color: Colors.red,);  
  
    } }, ),
```

GESTIÓN DEL ESTADO DE LA EDIFICACIÓN

Antes de implementar la gestión de estado para la aplicación de diario de cliente que creó en el Capítulo 14, repasemos el plan general y los pasos prioritarios. En orden de creación, creará la clase de modelo, las clases de servicio, las clases de utilidad, las clases de validación, las clases de BLoC y la clase InheritedWidget como proveedor y, por último, agregará la administración de estado y los BLoC para todas las páginas. Comienza modificando la página principal, creando la página de inicio de sesión, modificando la página de inicio y creando la página de entrada.

Tenga en cuenta que, desde las páginas del widget de la interfaz de usuario, injectará las clases de servicio authentication.dart y db_firestore.dart específicas de la plataforma en el constructor de la clase BLoC. La clase BLoC utiliza la clase abstracta de la API para recibir la clase de servicio específica de la plataforma inyectada, lo que hace que la plataforma de la clase BLoC sea independiente. Si también estuviera creando una versión web de la aplicación de diario, injectaría las clases de servicio de base de datos y autenticación apropiadas para la web en las clases de BLoC, y simplemente funcionarían; estos son algunos de los beneficios de usar el patrón BLoC.

La Tabla 15.1 enumera la estructura de carpetas y páginas para la aplicación de diario.

TABLA 15.1: Estructura de carpetas y archivos

CARPETAS	ARCHIVOS
bloques	autenticación_bloc.dart proveedor_bloque_autenticación.dart home_bloc.dart home_bloc_proveedor.dart journal_edit_bloc.dart journal_edit_bloc_provider.dart login_bloc.dart
clases	format_dates.dart mood_icons.dart validadores.dart
modelos	diario.dart
paginas	edit_entry.dart casa.dart iniciar sesión.dart
servicios	autenticación.dart autenticación_api.dart db_firestore.dart db_firestore_api.dart
Carpeta raíz	dardo principal

Para ayudarlo a visualizar la aplicación que está desarrollando, la Figura 15.3 muestra el diseño final de la aplicación de registro de estados de ánimo. De izquierda a derecha, muestra la página de inicio de sesión, la página de inicio, la eliminación de entradas de diario y la página de edición de entradas.

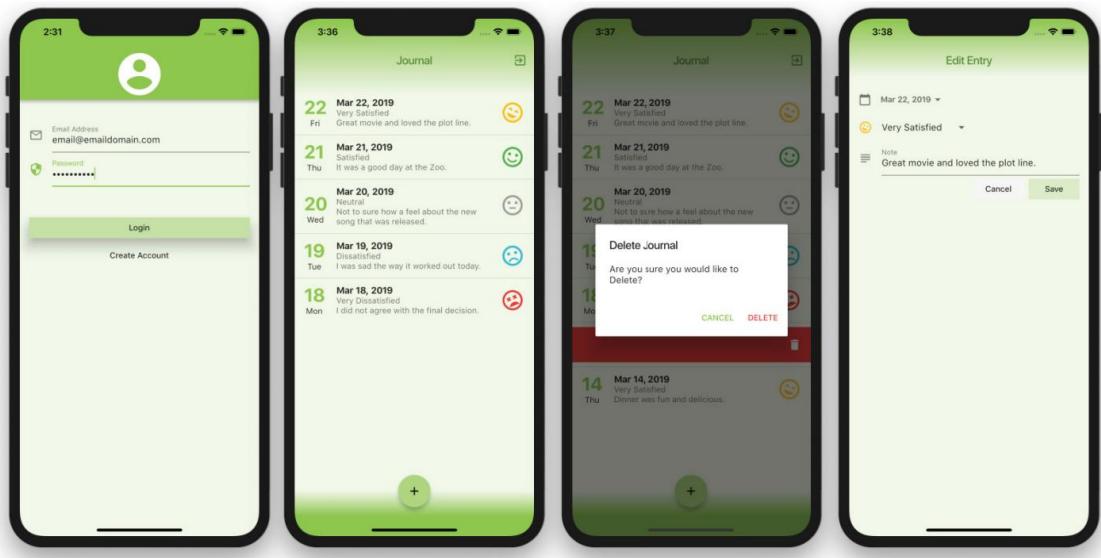


FIGURA 15.3: La última aplicación de diario de estado de ánimo

Adición de la clase de modelo de revista

Para la aplicación de diario, creará una clase de modelo de diario que es responsable de mantener entradas de diario individuales y asignar un documento de Cloud Firestore a una entrada de diario . La clase `Journal` contiene los campos `documentID`, `date`, `mood`, `note` y `uid` String . La variable `documentID` almacena una referencia al ID único del documento de la base de datos de Cloud Firestore. La variable `uid` almacena el ID único del usuario que ha iniciado sesión. La variable de fecha tiene el formato de un estándar ISO 8601, por ejemplo, 2019-03-18T13:56:54.985747. La variable de estado de ánimo almacena el nombre del estado de ánimo como `Satisficho`, `Neutral`, etc. La variable de nota almacena la descripción detallada del diario para la entrada.

PRUÉBALO Creación de la clase de modelo de diario

En esta sección, continúe editando el proyecto de diario que creó en el Capítulo 14. Agregará la clase de modelo `Diario` .

1. Cree un nuevo archivo Dart en la carpeta de modelos . Haga clic derecho en la carpeta de modelos , seleccione Nuevo Dardo Archivo, ingrese `journal.dart` y haga clic en el botón Aceptar para guardar.

2. Cree la estructura de la clase `Journal` .

```
diario de clase {  
}  
}
```

3. Dentro de la clase Journal , agregue las declaraciones para documentID, date, mood, note y uid Variables de cadena .

```
Cadena documentoID;  
Fecha de cadena;  
Estado de ánimo de cadena;  
Nota de cuerda;  
fluido de cadena;
```

4. Agregue una nueva línea e ingrese el constructor del Diario con los parámetros nombrados encerrándolos entre corchetes ({}). Haga referencia a cada variable documentID, date, mood, note y uid usando el azúcar sintáctico para acceder a los valores con la palabra clave this , refiriéndose al estado actual en la clase.

```
Diario({ this.documentID,  
        this.date,  
        this.mood,  
        this.note,  
        this.uid  
});
```

5. Agregue una nueva línea e ingrese el método de fábrica Journal.fromDoc() que es responsable de convertir y asignar un registro de documento de la base de datos de Cloud Firestore a una entrada de diario individual.

```
factory Journal.fromDoc(dynamic doc) => Journal( documentID:  
        doc.documentID, date: doc["date"],  
        mood: doc["mood"], note:  
        doc["note"], uid: doc["uid"]  
  
);
```

La siguiente es la clase de diario completa:

```
diario de clase {  
    Cadena documentoID;  
    Fecha de cadena;  
    Estado de ánimo de cadena;  
    Nota de cuerda;  
    fluido de cadena;  
  
    Diario({ this.documentID,  
            this.date,  
            this.mood,  
            this.note,  
            this.uid  
});  
  
fábrica Diario.fromDoc(documento dinámico) => Diario(documentID:  
        doc.documentID,
```

```
fecha: doc["fecha"], estado  
de ánimo: doc["estado de  
ánimo"], nota: doc["nota"],  
uid: doc["uid"]  
);  
}
```

CÓMO FUNCIONA

Creó el archivo journal.dart que contiene la clase Journal que es responsable de realizar un seguimiento de las entradas de diario individuales con las variables documentID, date, mood, note y uid String . Creó el método Journal.fromDoc() que asigna un registro de documento de base de datos de Cloud Firestore a una entrada de diario individual .

Adición de las clases de servicio

Se denominan clases de servicio porque envían y reciben llamadas a un servicio. La aplicación de diario tiene dos clases de servicio para manejar las llamadas a la API de la base de datos de Firebase Authentication y Cloud Firestore.

La clase AuthenticationService implementa la clase abstracta AuthenticationApi . La clase DbFirestoreService implementa la clase abstracta DbApi . La siguiente es una llamada de muestra a la base de datos de Cloud Firestore para consultar registros; La Tabla 15.2 describe los detalles:

```
Firestore.instance .collection("diarios") .where('uid',  
isEqualTo: uid) .snapshots()
```

TABLA 15.2: Cómo consultar la base de datos

LLAMAR	DESCRIPCIÓN
Firestore.instancia	Obtenga la referencia de Firestore.instance .
.colección('revistas')	Especifique el nombre de la colección.
.where('uid', esIgual a: uid)	El método where() filtra por el campo especificado.
.instantáneas()	El método snapshots() devuelve un Stream de QuerySnapshot que contiene los registros.

Cloud Firestore admite el uso de transacciones. Uno de los beneficios de usar transacciones es agrupar varias operaciones (agregar, actualizar, eliminar) en una sola transacción. Otro caso es la edición concurrente: cuando varios usuarios están editando el mismo registro, la transacción se ejecuta nuevamente, asegurándose de que se utilicen los datos más recientes antes de la actualización. Si una de las operaciones falla, la transacción no realizará una actualización parcial. Sin embargo, si la transacción es exitosa, se ejecutan todas las actualizaciones.

La siguiente es una transacción de muestra que toma la referencia del documento _docRef y llama a la ejecución Método Transaction() para actualizar los datos del documento:

```
DocumentReference _docRef = _firestore.collection('journals').document('Cf409us32'); var journalData = { 'fecha': diario.fecha,  
'estado de ánimo':  
    diario.estado de ánimo, 'nota':  
    diario.nota, };  
  
_firestore.runTransaction((transacción) asíncrono {  
    esperar  
        transacción .update(_docRef,  
            journalData) .catchError((error) => print('Error al actualizar: $error'));  
});
```

PRUÉBELO Creación de clases de servicio de autenticación En esta sección,

continúe editando el proyecto de diario . Agregará las clases AuthenticationApi y Authentication para manejar la API de autenticación de Firebase.

1. Cree un nuevo archivo Dart en la carpeta de servicios . Haga clic con el botón derecho en la carpeta de servicios , seleccione Nuevo Archivo Dart, ingrese autenticación_api.dart y haga clic en el botón Aceptar para guardar.

2. Cree la clase AuthenticationApi y use el modificador abstracto antes de la declaración de clase.

```
clase abstracta AuthenticationApi {  
  
}
```

3. Dentro de la clase AuthenticationApi , agregue los siguientes métodos de interfaz:

```
getFirebaseAuth();  
Future<String> currentUserId();  
Future<void> signOut();  
Future<String> signInWithEmailAndPassword({String email, String contraseña});  
Future<String> createUserWithEmailAndPassword({String email, String contraseña});  
Future<void> sendEmailVerification();  
Future<bool> isEmailVerified();
```

4. Cree un nuevo archivo Dart en la carpeta de servicios . Haga clic derecho en la carpeta de servicios , seleccione Nuevo Archivo Dart, ingrese autenticación.dart y haga clic en el botón Aceptar para guardar. Importe el paquete firebase_auth .dart y la clase abstracta authentication_api.dart .

5. Agregue una nueva línea y cree la clase AuthenticationService que implementa la clase abstracta AuthenticationApi .

6. Declare la variable FirebaseAuth _firebaseAuth final que tiene una referencia al FirebaseAuth.instancia.

7. Agregue el método `_getFirebaseAuth()` y devuelva la variable `_firebaseAuth`. `FirebaseAuth` es el punto de entrada del SDK de Firebase Authentication.

```
importar 'paquete:firebase_auth/firebase_auth.dart'; import 'paquete:journal/
services/authentication_api.dart';

clase AuthenticationService implementa autenticación {
    final FirebaseAuth _firebaseAuth = FirebaseAuth.instance;

    FirebaseAuth getFirebaseAuth() { return
        _firebaseAuth;

    }
}
```

8. Agregue el método asíncrono `Future<String> currentUserUid()` responsable de recuperar el usuario actualmente conectado.`uid` .

```
Future<String> currentUserUid() asíncrono { FirebaseUser
    user = await _firebaseAuth.currentUser(); devolver usuario.uid;

}
```

9. Agregue el método asincrónico `Future<void> signOut()` responsable de cerrar la sesión del usuario actual.

```
Future<void> signOut() asíncrono {
    devolver _firebaseAuth.signOut();
}
```

10. Agregue el método `Future<String> signInWithEmailAndPassword` aceptando el correo electrónico y la contraseña de los parámetros nombrados como `String`. Este método es responsable de iniciar sesión en un usuario mediante un proveedor de autenticación de correo electrónico/contraseña llamando al método `_firebaseAuth.signInWithEmailAndPassword` y devuelve el `user.uid`.

```
Future<String> signInWithEmailAndPassword({String email, String password}) async { FirebaseUser user = await
    _firebaseAuth.signInWithEmailAndPassword(email: email,
    contraseña: contraseña);
    devolver usuario.uid;
}
```

11. Agregue el método `Future<String> createUserWithEmailAndPassword` aceptando el correo electrónico y la contraseña de los parámetros nombrados como `String`. Este método es responsable de crear un usuario por proveedor de autenticación de correo electrónico/contraseña llamando al método `_firebaseAuth.createUserWithEmailAndPassword` y devuelve el `user.uid` recién creado.

```
Future<String> createUserWithEmailAndPassword({String email, String password}) async { FirebaseUser user = await
    _firebaseAuth.createUserWithEmailAndPassword(email:
    correo electrónico, contraseña:
    contraseña); devolver usuario.uid;
}
```

-
12. Agregue el método Future<void> sendEmailVerification que recupera el usuario que inició sesión actualmente llamando al método _firebaseAuth.currentUser . Una vez que se recupera el usuario , llama al usuario. método sendEmailVerification para enviar un correo electrónico al usuario para verificar que fue él quien creó la cuenta.

```
Future<void> sendEmailVerification() asíncrono {  
    FirebaseUser usuario = esperar _firebaseAuth.currentUser();  
    usuario.sendEmailVerification();  
}
```

13. Agregue el método Future<bool> isEmailVerified que llama al método _firebaseAuth.currentUser para recuperar el usuario conectado actual. Una vez que se recupera el usuario , lo devuelve . isEmailVerified valor booleano para verificar que el usuario haya verificado su correo electrónico.

```
Future<bool> isEmailVerified() async { FirebaseUser  
    user = await _firebaseAuth.currentUser(); volver usuario.isEmailVerified;  
}
```

CÓMO FUNCIONA

Creó el archivo authentication_api.dart con la clase abstracta AuthenticationApi que contiene los métodos de interfaz.

Para implementar la lógica del código que llama a la API de autenticación de Firebase, creó el archivo .dart de autenticación que contiene la clase AuthenticationService que implementa la clase abstracta AuthenticationApi . Cada método de la clase AuthenticationService llama a la API de autenticación de Firebase.

PRUÉBELO Creación de las clases de servicio DbFirestoreService

En esta sección, continúe editando el proyecto de diario . Agregará las clases de servicio DbApi y DbFirestore para manejar la API de la base de datos de Cloud Firestore.

1. Cree un nuevo archivo Dart en la carpeta de servicios . Haga clic derecho en la carpeta de servicios , seleccione Nuevo Dart Archivo, ingrese db_firestore_api.dart y haga clic en el botón Aceptar para guardar.
2. Cree la clase DbApi y use el modificador abstracto antes de la declaración de clase. Importe la clase journal.dart .

```
import 'paquete: diario/modelos/diario.dart';  
  
clase abstracta DbApi {  
}
```

3. Dentro de la clase DbApi , agregue los siguientes métodos de interfaz:

```
Stream<Lista<Diario>> getJournalList(String uid);  
Future<Journal> getJournal(String documentID);
```

```
Future<bool> addJournal(Diario diario); void  
updateJournal(diario diario); void  
updateJournalWithTransaction(Diario diario); void deleteJournal(diario  
diario);
```

4. Cree un nuevo archivo Dart en la carpeta de servicios . Haga clic con el botón derecho en la carpeta de servicios , seleccione Nuevo Archivo Dart, ingrese db_firestore.dart y haga clic en el botón Aceptar para guardar. Importe el paquete cloud_firestore .dart , la clase journal.dart y la clase db_firestore_api.dart .
5. Agregue una nueva línea y cree la clase DbFirestoreService que implementa DbApi clase abstracta .
6. Declare la variable Firestore _firestore final que tiene una referencia a Firestore .instancia. Agregue la variable final _collectionJournals String que contiene el nombre de la colección de Firestore inicializado en las revistas.
7. Agregue una nueva línea y agregue el constructor DbFirestoreService() que usa el _firestore instancia para llamar al método settings() para habilitar timestampsInSnapshotsEnabled estableciendo el valor en verdadero. En el futuro, Cloud Firestore habilitará el valor como verdadero de forma predeterminada, y es bueno optar por este nuevo comportamiento.

```
importar 'paquete:cloud_firestore/cloud_firestore.dart'; import 'paquete:  
diario/modelos/diario.dart'; importar 'paquete: diario/servicios/  
db_firestore_api.dart';  
  
clase DbFirestoreService implementa DbApi { Firestore  
_firestore = Firestore.instance; String _colecciónDiarios =  
'diarios';  
  
DbFirestoreService()  
{ _firestore.settings(timestampsInSnapshotsEnabled: true); } }
```

8. Agregue el método Stream<List<Journal>> getJournalList , tomando el parámetro uid String . Tenga en cuenta que el valor de uid es el ID de usuario que ha iniciado sesión. Este método es responsable de recuperar las entradas del diario. Use la instancia de _firestore con el operador de punto (.) para establecer los valores de propiedad del miembro de _firestore . Consulte la Tabla 15.2 para ver cómo consultar una base de datos de Cloud Firestore.
9. Establezca el valor de la colección en la variable _collectionJournals y configure el método where() para filtrar por el campo uid . El método snapshots() devuelve un QuerySnapshot Stream.
10. Continúe con el operador de punto y agregue el método map() pasando la instantánea recibida. Dentro del método map() , convierte los documentos de la instantánea y los asigna a las clases de Journal utilizando el método fromDoc(doc) de la clase Journal y convirtiéndolo en List() mediante el método toList() .
11. Tome la variable _journalDocs que se completa con la lista de clases de diario y use el método sort() para ordenar las fechas en orden descendente. Para la última línea, devuelva la lista de clases de diario _journalDocs .

```
Corriente<Lista<Diario>> getJournalList(String uid) {  
    volver  
    _firestore .colección(_colecciónJournals)
```

```

    .where('uid', isEqualTo:
        uid) .snapshots() .map((QuerySnapshot
            snapshot) { List<Journal> _journalDocs = snapshot.documents.map((doc) => Journal.
            fromDoc(doc)).toList();
            _journalDocs.sort((comp1, comp2) => comp2.date.compareTo(comp1.date)); volver _journalDocs; });
}

```

12. Agregue el método asincrónico Future<bool> addJournal(Journal journal) que toma el parámetro de clase Journal . Este método es responsable de agregar una nueva entrada de diario.
13. Declare la variable DocumentReference _documentReference que contiene el nuevo documento agregado. Use la palabra clave await con la instancia de _firestore , especifique la colección con la variable _collectionJournals y llame al método add() .
14. El método add() toma las variables de fecha, estado de ánimo, nota y uid de los campos de entrada de diario.
15. Agregue una nueva línea para devolver un valor booleano si el registro se creó correctamente comprobando si _documentReference.documentID != null . Si documentID no es nulo, el registro se creó y devuelve un valor verdadero ; de lo contrario, devuelve un valor falso .

```

Future<bool> addJournal(Diario diario) asíncrono {
    DocumentReference _documentReference =
        await _firestore.collection(_collectionJournals).add({ 'date': journal.date, 'mood':
            journal.mood, 'note':
            journal.note, 'uid': journal.uid, });
    volver
    _documentReference.documentID != null;
}

```

16. Agregue el método asíncrono updateJournal(Journal journal) tomando el parámetro de clase Journal . Este método es responsable de actualizar una entrada de diario existente.
17. Use la instancia de _firestore con el operador de punto (.) para establecer los valores de propiedad del miembro de _firestore . Establezca el valor de la colección en la variable _collectionJournals y establezca la variable del documento en journal.documentID , que es el ID del documento de Cloud Firestore.
18. Continúe con el operador punto y agregue el método updateData() , pasando la fecha, el estado de ánimo y anote los valores de la variable del diario . Agregue el método catchError() para interceptar cualquier error e imprimirlo en la consola.

```
void updateJournal (Diario diario) async { await _firestore
```

```

    .collection(_colecciónJournals) .document(journal.documentID) .updateData({ 'date': journal.date, 'mood': journal.
    })
}
```

19. Agregue el método asíncrono `deleteJournal(Journal journal)` , tomando el parámetro de clase `Journal` .

Este método es responsable de eliminar una entrada de diario.

20. Use la instancia de `_firestore` con el operador de punto `(.)` para establecer los valores de propiedad del miembro `_firestore` . Establezca el valor de la colección en la variable `_collectionJournals` y establezca la variable del documento en `journal.documentID` .

21. Continúe con el operador punto y agregue el método `delete()` . Agregue el método `catchError()` a intercepte cualquier error e imprímalo en la consola.

```
void deleteJournal(Diario diario) async { await _firestore  
  
    .collection(_colecciónJournals) .document(journal.documentID) .delete() .catchError((error) => print('Error al eliminar: $error'));  
}
```

CÓMO FUNCIONA

Creó el archivo `db_firestore_api.dart` con la clase abstracta `DbApi` que contiene los métodos de interfaz.

Creó el archivo `db_firestore.dart` que contiene la clase `DbFirestoreService` que implementa la clase abstracta `DbApi` . Cada método en la clase `DbFirestoreService` llama a la API de la base de datos de Cloud Firestore.

Agregar la clase de validadores

La clase `Validators` usa `StreamTransformer` para validar si el correo electrónico tiene el formato correcto usando al menos un signo @ y un punto. El validador de contraseña verifica un mínimo de seis caracteres ingresados. La clase `Validators` se utiliza con las clases `BLoC`.

El `StreamTransformer` transforma un `Stream` que se usa para validar y procesar valores dentro de un `Stream`. Los datos entrantes son un `Stream` y los datos salientes después del procesamiento son un `Stream`. Por ejemplo, una vez que se procesan los datos entrantes, puede usar el método `fregadero.add()` para agregar datos a la secuencia o usar el método `fregadero.addError()` para devolver un error de validación. El `StreamTransformer`. El constructor `fromHandlers` se usa para delegar eventos a una función determinada.

El siguiente es un ejemplo que muestra cómo usar `StreamTransformer` usando el constructor `fromHandlers` para validar si el correo electrónico está en el formato correcto:

```
StreamTransformer<String, String>.fromHandlers(handleData: (correo electrónico, sumidero) {  
    if (email.contains('@') && email.contains('.')) {  
        fregadero.add(correo electrónico);  
    } else if ((correo electrónico.longitud > 0)  
    { fregadero.addError('Ingrese un correo electrónico válido');  
  
});});
```

PRUÉBALO Creación de la clase de validadores

En esta sección, continúe editando el proyecto del diario agregando la clase Validators .

1. Cree un nuevo archivo Dart en la carpeta de clases . Haga clic con el botón derecho en la carpeta de clases , seleccione Nuevo Archivo Dart, ingrese validators.dart y haga clic en el botón Aceptar para guardar.

2. Importe la biblioteca async.dart y cree la clase Validators .

```
importar 'dardo: asíncrono';  
  
validadores de clase {  
  
}
```

3. Agregue la variable validateEmail y use la palabra clave final . Este método es responsable de verificar si el correo electrónico tiene el formato correcto al tener al menos un signo @ y un punto.

4. Inicialice la variable validateEmail llamando a StreamTransformer.fromHandlers constructor.

5. Dentro del controlador, agregue una declaración if para verificar si email.contains('@') && email . contiene('.'), y cuando ambas expresiones se validen como verdaderas, agregue el método sink.add(email) .

6. Agregue una declaración else if para verificar si email.length > 0, es decir, si el usuario ha escrito al menos un carácter, luego agregue el fregadero.addError('Ingrese un correo electrónico válido').

```
correo electrónico de validación final =  
StreamTransformer<String, String>.fromHandlers(handleData: (correo electrónico, sumidero) {  
    if (email.contains('@') && email.contains('.')) {  
        fregadero.add(correo electrónico);  
    } else if (correo electrónico.longitud > 0)  
        {fregadero.addError('Ingrese un correo electrónico válido');  
  
} });
```

7. Agregue la variable validatePassword y use la palabra clave final . Este método se encarga de verificar si la longitud de la contraseña es de al menos 6 caracteres.

8. Inicialice la variable validatePassword llamando a StreamTransformer.fromHandlers constructor.

9. Dentro del controlador, agregue una declaración if para verificar si password.length >= 6, y si la expresión se valida como verdadera, agregue el método sink.add (contraseña) .

10. Agregue una declaración else if para verificar si la contraseña.longitud > 0, lo que significa que si el usuario ha escrito al menos un carácter, luego agregue el fregadero.addError(La contraseña debe tener al menos 6 caracteres).

```
contraseña de validación final = StreamTransformer<String, String>.fromHandlers(  
handleData: (contraseña, dispositivo) { if  
(contraseña.longitud >= 6) {
```

```
fregadero.add(contraseña); } else if  
(contraseña.longitud > 0) { fregadero.addError('La contraseña debe tener  
al  
menos 6 caracteres'); }}); }
```

CÓMO FUNCIONA

Creó el archivo validators.dart que contiene la clase Validators . El StreamTransformer se utiliza para validar que los correos electrónicos y las contraseñas superen un estándar mínimo. Si los valores de las expresiones son verdaderos, el correo electrónico o la contraseña se agregan a la secuencia mediante el método sink.add() . Si las expresiones se validan como falsas, el método sink.addError() devuelve la descripción del error.

Agregar el patrón BLoC

En esta sección, creará el BLoC de autenticación, el proveedor de BLoC de autenticación, el BLoC de inicio de sesión, el BLoC de inicio, el proveedor de BLoC de inicio, el BLoC de edición de diario y el proveedor de BLoC de edición de diario. El BLoC de inicio de sesión no necesita una clase de proveedor porque no depende de recibir datos de otras páginas.

Quiero recordarle este importante concepto: las clases de BLoC son independientes de la plataforma y no dependen de paquetes o clases específicos de la plataforma. Por ejemplo, la página de inicio de sesión inyecta la clase AuthenticationService específica de la plataforma (Flutter) en el constructor de la clase LoginBloc . La clase BLoC receptora tiene la clase AuthenticationApi abstracta que recibe la clase AuthenticationService inyectada , lo que hace que la clase BLoC sea independiente de la plataforma.

Agregar el bloque de autenticación

El AuthenticationBloc es responsable de identificar las credenciales de los usuarios que han iniciado sesión y monitorear el estado de inicio de sesión de autenticación de los usuarios. Cuando se crea una instancia de AuthenticationBloc , inicia un agente de escucha de Stream Controller que supervisa las credenciales de autenticación del usuario y, cuando se producen cambios, el agente de escucha actualiza el estado de la credencial llamando a un evento del método sink.add() . Si el usuario ha iniciado sesión, los eventos del receptor envían el valor de uid del usuario , y si el usuario cierra la sesión, los eventos del receptor envían un valor nulo , lo que significa que ningún usuario ha iniciado sesión.

PRUÉBALO Creando el AuthenticationBloc

En esta sección, continúe editando el proyecto de diario . Agregará la clase AuthenticationBloc para manejar la llamada al servicio de autenticación de Firebase para iniciar o cerrar la sesión de un usuario.

1. Cree un nuevo archivo Dart en la carpeta de bloques . Haga clic con el botón derecho en la carpeta de bloques , seleccione Nuevo Archivo Dart, ingrese autenticación_bloc.dart y haga clic en el botón Aceptar para guardar.

2. Importe la biblioteca async.dart y la clase authentication_api.dart y cree el Clase de bloque de autenticación .

```
importar 'dardo:  
asíncrono'; import 'paquete:journal/services/authentication_api.dart';
```

```
clase bloque de autenticación {  
}
```

3. Dentro de la clase AuthenticationBloc , declare la variable API de autenticación AuthenticationApi final . El patrón BLoC requiere que inyecte las clases específicas de la plataforma y pasará la clase AuthenticationService en el constructor de BLoC. La variable authenticationApi recibe la clase AuthenticationService inyectada .

API de autenticación final API de autenticación;

4. Agregue la variable _authenticationController como String StreamController, agregue el addUser variable getter como String Sink, y agregue el usuario getter como String Stream. Cada vez que el usuario inicia o cierra sesión, _authenticationController StreamController se actualiza con el captador addUser .

```
final StreamController<String> _authenticationController = StreamController<String>(); Sink<String> get  
addUser => _authenticationController.sink; Stream<String> get usuario =>  
_authenticationController.stream;
```

5. Agregue la variable _logoutController como bool StreamController, la variable logoutUser getter como un Sink bool, y el getter listLogoutUser como un Stream bool. Cada vez que el usuario cierra la sesión, _logoutController StreamController se actualiza con el captador logoutUser .

```
final StreamController<bool> _logoutController = StreamController<bool>(); Sink<bool> get  
logoutUser => _logoutController.sink; Stream<bool> get  
listLogoutUser => _logoutController.stream;
```

6. Agregue una nueva línea e ingrese el constructor AuthenticationBloc que recibe el this inyectado . parámetro de autenticaciónApi . Tenga en cuenta que el parámetro inyectado es la clase AuthenticationService . Dentro del constructor, agregue una llamada al método onAuthChanged() que creó en el pas

```
AuthenticationBloc(this.authenticationApi)  
{ onAuthChanged();  
}
```

7. Agregue el método dispose() y llame a los métodos _authenticationController.close() y _logout Controller.close() para cerrar la secuencia de StreamController cuando no sea necesario. Tenga en cuenta que para la clase AuthenticationBloc no se llamará al método close() porque la autenticación debe ser accesible durante toda la vida útil de la aplicación.

```
void dispose()  
{ _authenticationController.close();  
_logoutController.close(); }
```

8. Agregue el método onAuthChanged() que es responsable de configurar un oyente para verificar cuándo el usuario inicia y cierra sesión. Dentro del método, llame a authenticationApi.getFirebaseAuth() para obtener FirebaseAuth.instance de la clase de servicio de autenticación. Continúe usando el operador de punto para llamar a onAuthStateChanged.listen((usuario)) para configurar el oyente. cuando el usuario

inicia sesión, la variable de usuario devuelve la clase FirebaseAuther con la información del usuario. Cuando el usuario cierra la sesión, la variable de usuario devuelve un valor nulo .

9. Dentro del oyente, agregue la variable String uid final inicializada mediante el operador ternario para comprobar si user != null y recuperar el valor de user.uid ; de lo contrario, devuelve un valor nulo .
10. Agregue una nueva línea y llame al método _addUser.add(uid) para agregar el valor al sumidero con el uid de usuario o el valor nulo .
11. Agregue una nueva línea y llame al oyente _logoutController.stream.listen((logout)) que está se llama cuando el usuario cierra la sesión.
12. Dentro del oyente, agregue una declaración if para verificar si logout == true y llame al signOut() que creará en el paso 13.

```
void onAuthChanged()
```

```
{ autenticaciónApi .getFirebaseAuth() .onAuthStateChanged .listen((usuario) {  
    cadena final uid = usuario! = nulo? usuario.uid: nulo;  
  
    agregarUsuario.agregar(uid);}); _logoutController.stream.listen((cerrar sesión) {  
    if (cerrar sesión == verdadero)  
        { _signOut(); } });}
```

13. Agregue el método void _signOut() que llama al método authenticationService.signOut() del servicio de autenticación para cerrar la sesión del usuario.

```
void _signOut()  
{authenticationApi.signOut();}  
}
```

CÓMO FUNCIONA

Para identificar las credenciales del usuario que inició sesión y monitorear el estado de inicio de sesión, creó el archivo authentication_bloc.dart que contiene la clase AuthenticationBloc . Declaró una referencia a la clase AuthenticationApi para obtener acceso a la API de autenticación de Firebase. La variable authenticationApi recibe la clase AuthenticationService inyectada . Para agregar datos a la propiedad de flujo de StreamController , usó el método sink.add() y la propiedad de flujo emite los últimos eventos de flujo . Agregó métodos que llaman al servicio de autenticación para iniciar sesión, cerrar sesión y crear usuarios nuevos.

Adición de AuthenticationBlocProvider La clase

AuthenticationBlocProvider es responsable de pasar el estado entre widgets y páginas mediante el uso de la clase InheritedWidget como proveedor. El constructor AuthenticationBlocProvider toma una clave, un widget y la variable this.authenticationBloc , que es la clase AuthenticationBloc .

PRUÉBALO Crear el AuthenticationBlocProvider

En esta sección, continúe editando el proyecto de diario . Agregará la clase de vider AuthenticationBlocPro como proveedor de la clase AuthenticationBloc para manejar el inicio y cierre de sesión y para monitorear las credenciales de un usuario. La clase AuthenticationBloc llama a la API de autenticación de Firebase de la clase de servicio AuthenticationService .

1. Cree un nuevo archivo Dart en la carpeta de bloques . Haga clic con el botón derecho en la carpeta de bloques , seleccione Nuevo Archivo Dart, ingrese authentication_bloc_provider.dart y haga clic en el botón Aceptar para guardar.

2. Importe el paquete material.dart y el paquete authentication_bloc.dart y cree el Clase AuthenticationBlocProvider que amplía la clase InheritedWidget .

```
importar 'paquete: flutter/material.dart'; import  
'paquete:journal/blocs/authentication_bloc.dart';  
  
clase AuthenticationBlocProvider extiende InheritedWidget {  
  
}
```

3. Dentro de la clase AuthenticationBlocProvider , declare el AuthenticationBloc final variable de bloque de autenticación .

```
bloque de autenticación final bloc de autenticación;
```

4. Agregue el constructor AuthenticationBlocProvider con la palabra clave const . Agregue al constructor los parámetros key, child y this.authenticationBloc .

```
const AuthenticationBlocProvider( {Clave clave,  
Widget hijo, this.authenticationBloc} ) : super(clave: clave, hijo: hijo);
```

5. Agregue el método AuthenticationBlocProvider of (BuildContext context) con el palabra clave estática .

6. Dentro del método, devuelva el AuthenticationBlocProvider usando el método HeredarFromWidget OfExactType que permite que los widgets secundarios obtengan la instancia del proveedor del Authentication BlocProvider .

```
AuthenticationBlocProvider estatico de (contexto BuildContext) {  
    volver (context.inheritFromWidgetOfExactType(AuthenticationBlocProvider)  
        como AuthenticationBlocProvider);  
}
```

7. Agregue y reemplace el método updateShouldNotify para comprobar si el bloque de autenticación no es igual al antiguo bloque de autenticación AuthenticationBlocProvider. Si la expresión devuelve verdadero, el marco notifica a los widgets que contienen los datos heredados que necesitan reconstruir.

```
@override  
bool updateShouldNotify(AuthenticationBlocProvider old) => authenticationBloc !=  
    old.authenticationBloc;
```

CÓMO FUNCIONA

Para pasar el estado entre widgets y páginas, creó el archivo authentication_bloc_provider.dart que contiene la clase AuthenticationBlocProvider como proveedor de la clase AuthenticationBloc . El constructor de la clase AuthenticationBlocProvider toma la clave, el widget y los parámetros this.authenticationBloc . El método of() devuelve el resultado del métodoHeredarFromWidgetOfExactType que permite que los widgets secundarios obtengan la instancia del proveedor AuthenticationBlocProvider .

El método updateShouldNotify comprueba si el valor ha cambiado y el marco notifica a los widgets para que se reconstruyan.

Agregar el LoginBloc El LoginBloc

es responsable de monitorear la página de inicio de sesión para verificar un formato de correo electrónico válido y la longitud de la contraseña. Cuando se crea una instancia de LoginBloc , inicia los oyentes de StreamController que monitorean el correo electrónico y la contraseña del usuario, y una vez que pasan la validación, se habilitan los botones de inicio de sesión y creación de cuenta. Una vez que los valores de inicio de sesión y contraseña pasan la validación, se llama al servicio de autenticación para iniciar sesión o crear un nuevo usuario. La clase Validators es responsable de validar los valores de correo electrónico y contraseña.

PRUÉBALO Creando el LoginBloc

En esta sección, continúe editando el proyecto de diario . Agregará la clase LoginBloc para manejar los botones de correo electrónico, contraseña, inicio de sesión y creación de cuenta de la página de inicio de sesión. LoginBloc también es responsable de llamar al servicio de autenticación de Firebase para iniciar sesión o crear un nuevo usuario .

1. Cree un nuevo archivo Dart en la carpeta de bloques . Haga clic derecho en la carpeta de bloques , seleccione Nuevo → Archivo Dart, ingrese login_bloc.dart y haga clic en el botón Aceptar para guardar.
2. Importe la biblioteca async.dart , importe las clases validators.dart y authentication_api.dart , y cree la clase LoginBloc usando la palabra clave with y la clase Validators .

```
import 'dart:async';  
import 'package:diario/clases/validadores.dart'; import 'package:journal/  
services/authentication_api.dart';  
  
clase LoginBloc con validadores {  
  
}
```

3. Dentro de la clase LoginBloc , declare la variable final AuthenticationApi authenticationApi . Agregue las variables privadas String _email y _password y las variables privadas bool _emailValid y _passwordValid .

API de autenticación final API de autenticación;

```
Cadena _correo electrónico;
Cadena _contraseña;
bool_emailValid; bool
_contraseña válida;
```

4. Agregue la variable _emailController como String StreamController, agregue el captador de variable emailChanged como String Sink y agregue el captador de correo electrónico como String Stream. Tenga en cuenta que StreamController se inicializa con el flujo de transmisión () , ya que tendremos múltiples oyentes. Después de la propiedad stream , agregue el método transform(validateEmail) que llama a la clase StreamTransformer de Validators y valida la dirección de correo electrónico. StreamTransformer agrega a la propiedad del receptor el valor de la dirección de correo electrónico si pasa la validación o un error si falla.

```
final StreamController<String> _emailController = StreamController<String> .broadcast();
Sink<String>
get emailChanged => _emailController.sink; Stream<String> get email
=> _emailController.stream.transform(validateEmail);
```

5. Siguiendo los pasos anteriores, agregue los StreamControllers adicionales para manejar la contraseña, habilite los botones de inicio de sesión o creación de cuenta, y agregue las llamadas de inicio de sesión o creación de cuenta al servicio Cloud Firestore.

```
final StreamController<String> _passwordController = StreamController<String> .broadcast();
Sink<String>
get passwordChanged => _passwordController.sink; Stream<String> obtener
contraseña => _passwordController.stream.transform (validatePassword);
```

```
final StreamController<bool> _enableLoginCreateButtonController =
StreamController<bool>.broadcast();
Sink<bool> get enableLoginCreateButtonChanged => _enableLoginCreateButton Controller.sink;
Stream<bool> get
enableLoginCreateButton => _enableLoginCreateButtonCon troller.stream;
```

```
final StreamController<String> _loginOrCreateButtonController =
StreamController<String>();
Sink<String> get loginOrCreateButtonChanged => _loginOrCreateButtonController.sink; Stream<String> get
loginOrCreateButton => _loginOrCreateButtonController.stream;
```

```
final StreamController<String> _loginOrCreateController =
StreamController<String>();
Sink<String> get loginOrCreateChanged => _loginOrCreateController.sink; Stream<String>
get loginOrCreate => _loginOrCreateController.stream;
```

6. Agregue una nueva línea e ingrese el constructor LoginBloc que recibe el `this.authentica` inyectado parámetro `tionApi`. Tenga en cuenta que el parámetro inyectado es la clase `AuthenticationService`. Dentro del constructor, llama al método `_startListenersIfEmailPasswordAreValid()` que creaste en el paso 8.

```
LoginBloc(this.authenticationApi) {  
    _startListenersIfEmailPasswordAreValid(); }
```

7. Agregue el método `dispose()` y llame a `_passwordController`, `_emailController`, `_enableLoginCreateButtonController` y `_loginOrCreateButtonController` `close()` para cerrar la secuencia de `StreamController` cuando no se necesitan. void

```
dispose()  
{ _passwordController.close();  
_emailController.close();  
_enableLoginCreateButtonController.close();  
_loginOrCreateButtonController.close();  
_loginOrCreateController.close(); }
```

8. Agregue el método `_startListenersIfEmailPasswordAreValid()` que es responsable de configurar tres oyentes que verifican el correo electrónico, la contraseña y el inicio de sesión o crean secuencias de botones.

9. Dentro del método, agregue el oyente `email.listen((email))`. Dentro del oyente, establezca los valores `_email = email` y `_emailValid = true`.

10. Mediante el uso del operador punto, agregue el controlador de eventos `onError((error))` y establezca " y los valores `_email = _emailValid = false`.

11. Para ambos escenarios, llame al método `_updateEnableLoginCreateButtonStream()` que crear en el paso 14.

12. Agregue una nueva línea y, siguiendo los pasos anteriores, ingrese la contraseña.`listen((contraseña))` oyente.

13. Agregue el oyente `loginOrCreate.listen((acción))` y, mediante el uso de un operador ternario, establezca la variable de acción en `_login()` o `_createAccount()`, dependiendo de si el usuario ha elegido iniciar sesión o crear una nueva cuenta.

```
void _startListenersIfEmailPasswordAreValid()  
{ email.listen((email))  
{ _email = email;  
_emailValid = true;  
_updateEnableLoginCreateButtonStream();  
}).onError((error)  
{ _email = ";  
_emailValid = false;  
_updateEnableLoginCreateButtonStream();});  
  
contraseña.listen((contraseña))  
{ _contraseña =  
contraseña; _contraseñaValid = verdadero;
```

```
        _updateEnableLoginCreateButtonStream();
    }).onError((error) {
        { _password = "";
        _passwordValid = false;
        _updateEnableLoginCreateButtonStream(); });

    loginOrCreate.listen((acción) {
        acción == 'Iniciar sesión'? _Login Crear cuenta(); });
}
```

14. Agregue el método `_updateEnableLoginCreateButtonStream()` que verifica si las variables `_emailValid` y `_passwordValid` se evalúan como verdaderas y llame a `enableLoginCreateButtonChanged.add(true)` para agregar un valor verdadero a la propiedad del receptor . De lo contrario, agregue un valor falso a la propiedad del receptor . Los resultados del valor que se agrega a la propiedad del receptor habilitan o deshabilitan los botones de inicio de sesión o creación de cuenta.

```
vacío _updateEnableLoginCreateButtonStream() {
    if (_emailValid == verdadero && _passwordValid == verdadero)
        { enableLoginCreateButtonChanged.add(true);

    } más
    { enableLoginCreateButtonChanged.add(false);

}
```

15. Agregue el método asíncrono `Future<String> _login()` que es responsable de iniciar sesión en un usuario con las credenciales de correo electrónico/contraseña.
16. Dentro del método, agregue la variable `String _result = "` que rastrea si el inicio de sesión es exitoso o falla.
17. Agregue una declaración `if` para verificar si las variables `_emailValid` y `_passwordValid` se evalúan como un valor verdadero . Si la expresión se evalúa como verdadera, agregue la llamada `await authenticationApi.signInWithEmailAndPassword()` pasando los valores `_email` y `_password` .
18. Usando el operador de punto, agregue la devolución de llamada `then((user))` que establece el `_result = Variable 'éxito'` .
19. Agregue la devolución de llamada `catchError((error))` que establece la variable `_result = error` .
20. Agregue la declaración `return _result` para devolver un estado de que el inicio de sesión ha sido exitoso o devolver el error de inicio de sesión.
21. Agregue una declaración `else` para verificar si las variables `_emailValid` y `_passwordValid` evalúa a un valor falso y devuelve 'Correo electrónico y contraseña no son válidos' ya que la validación falló.

```
Future<String> _login() asíncrono {
    Cadena _resultado = "";
    if(_emailValid && _passwordValid) {
        aguardar autenticaciónApi.signInWithEmailAndPassword(correo electrónico: _correo electrónico,
        contraseña:
            _contraseña).then((usuario) { _resultado = 'Éxito';

```

```
        }).catchError((error) { print('Error de
            inicio de sesión: $error'); _result = error; });
        devolver _resultado;
    }
    else
    {
        return 'El correo
electrónico
y la contraseña no son válidos';
    }
}
```

22. Agregue el método asíncrono Future<String> `_createAccount()` que es responsable de crear una nueva cuenta y, si tiene éxito, iniciar sesión automáticamente en el nuevo usuario. Cuando se crea una nueva cuenta, es una buena práctica iniciar sesión automáticamente con el nuevo usuario.
Siga los pasos anteriores y agregue los métodos `createUserWithEmailAndPassword` y `signInWithEmailAndPassword`.

```
Future<String> _createAccount() asíncrono {
    Cadena _resultado = "";
    if(_emailValid && _passwordValid) {
        esperar autenticaciónApi.createUserWithEmailAndPassword(email: _email, contraseña: _password).then((usuario)
        { print('Usuario creado: $usuario');
            _result = 'Usuario creado: $usuario';
            authenticationApi.signInWithEmailAndPassword(email:
                _email, contraseña : _contraseña).then((usuario) {}).catchError((error) asíncrono {

                print('Error de inicio de sesión: $error');
                _resultado = error; });

            }).catchError((error) async { print('Creando
                error de usuario: $error'); }); devolver _resultado; } else
    {
        return 'Error al crear
usuario';
    }
}
```

CÓMO FUNCIONA

Para monitorear el inicio de sesión para verificar un formato de correo electrónico y una longitud de contraseña válidos, creó el archivo `login_bloc.dart` que contiene la clase `LoginBloc` que funciona con la clase `Validators`. Declaró una referencia a la clase `AuthenticationApi` para obtener acceso a la API de autenticación de Firebase. La variable `authenticationApi` recibe la clase `AuthenticationService` inyectada. Para agregar datos a la propiedad de flujo de `StreamController`, usó el método `sink.add()` y la propiedad de flujo emite los últimos eventos de flujo. Agregó métodos que llaman al servicio de autenticación de Firebase para iniciar sesión o crear una nueva cuenta mediante el proveedor de autenticación de correo electrónico o contraseña.

Agregando el HomeBloc

El HomeBloc es responsable de identificar las credenciales de los usuarios registrados y monitorear el estado de inicio de sesión de la autenticación del usuario. Cuando se crea una instancia de HomeBloc , inicia un agente de escucha StreamController que supervisa las credenciales de autenticación del usuario y, cuando se producen cambios, el agente de escucha actualiza el estado de la credencial llamando a un evento del método sink.add() . Si el usuario inició sesión, los eventos de receptor envían el valor de uid del usuario y, si el usuario cierra sesión, los eventos de receptor envían un valor nulo , lo que significa que ningún usuario ha iniciado sesión.

PRUÉBALO Creando el HomeBloc

En esta sección, continúe editando el proyecto de diario . Agregará la clase HomeBloc para manejar las llamadas al servicio de base de datos de Cloud Firestore.

1. Cree un nuevo archivo Dart en la carpeta de bloques . Haga clic derecho en la carpeta de bloques , seleccione Nuevo Archivo Dart, ingrese home_bloc.dart y haga clic en el botón Aceptar para guardar.

2. Importe la biblioteca async.dart ; importar la clase authentication.dart , la clase db_firestore_api .dart y la clase journal.dart ; y crea la clase HomeBloc .

```
importar 'dardo: asíncrono';
import 'paquete:jurnal/services/authentication_api.dart'; importar 'paquete:
diario/servicios/db_firestore_api.dart'; import 'paquete: diario/modelos/
diario.dart';

clase HomeBloc {
```

```
}
```

3. Dentro de la clase HomeBloc , declare la variable final DbApi dbApi y la variable final AuthenticationApi authenticationApi .

```
DbApi final dbApi; API
de autenticación final API de autenticación;
```

4. Agregue la variable _journalController como String StreamController, agregue el captador de variables _addListJournal (privado) como List<Journal> Sink y agregue el captador listJournal como Flujo de lista <Diario>.

```
final StreamController<Lista<Diario>> _journalController =
StreamController<Lista<Diario>>.broadcast();
Sink<List<Journal>> get _addListJournal => _journalController.sink; Stream<List<Journal>>
get listJournal => _journalController.stream;
```

5. Agregue la variable _journalDeleteController como Journal StreamController y agregue la variable getter deleteJournal como Journal Sink. Dado que este StreamController es responsable de eliminar diarios, no necesitará una lista de diarios eliminados Stream.

```
Final StreamController<Diario> _journalDeleteController =
StreamController<Diario>.broadcast(); Sink<Diario>
get eliminarDiario => _journalDeleteController.sink;
```

6. Agregue una nueva línea e ingrese al constructor HomeBloc , tomando los parámetros this.dbApi y this.authenticationApi . Tenga en cuenta que los parámetros inyectados son, respectivamente, las clases DbFirestoreService y AuthenticationService .

7. Dentro del constructor de HomeBloc , llame al método _startListeners() que creará en el paso 9.

```
HomeBloc(this.dbApi , this.authenticationApi)
{ _startListeners(); }
```

8. Agregue el método dispose() y llame a los métodos _journalController.close() y _journalDeleteController.close() para cerrar la transmisión de StreamController cuando no se necesiten.

```
void dispose()
{ _journalController.close();
_journalDeleteController.close(); }
```

9. Agregue el método _startListeners() responsable de configurar dos oyentes para recuperar una lista de diarios y para eliminar una entrada de diario individual.

10. Antes de iniciar los oyentes, debe recuperar el uid de usuario que ha iniciado sesión actualmente. Dentro de llame al método authenticationApi.getFirebaseAuth().currentUser() y, con el operador de punto, agregue la devolución de llamada then((user)) que devuelve el valor de user.uid .

11. Dentro del método currentUser() , llame al método dbApi.getJournalList() para obtener la lista de diarios filtrados por el uid del usuario de la clase de servicio de Cloud Firestore.

12. Continúe usando el operador de punto para llamar a listen((journalDocs)) para configurar el oyente.

13. Dentro del oyente, llame al receptor _addListJournal.add(journalDocs) para agregar la lista de diarios a la secuencia _journalController .

14. Agregue una nueva línea e ingrese el oyente _journalDeleteController.stream.listen((journal)) que devuelve un diario para ser eliminado. Dentro del oyente, llame a la clase DbApi dbApi. método deleteJournal(diario) para eliminar el diario de la base de datos.

```
void _startListeners() {
// Recuperar registros de diarios de Firestore como Lista<Diario> no DocumentSnapshot
autenticaciónApi.getFirebaseAuth().currentUser().then((usuario)
{ dbApi.getJournalList(usuario.uid).listen((journalDocs)
{ _addListJournal.add(journalDocs) ; });

_journalDeleteController.stream.listen((diario) {
dbApi.deleteJournal(diario);});});}
```

CÓMO FUNCIONA

Para identificar las credenciales del usuario que inició sesión y monitorear el estado de inicio de sesión de autenticación del usuario, creó el archivo `home_bloc.dart` que contiene la clase `HomeBloc`. Declaró una referencia a la clase `DbApi` para obtener acceso a la API de la base de datos de Cloud Firestore. También declaró una referencia a la clase `AuthenticationApi` para obtener acceso a la API de autenticación de Firebase. La variable `dbApi` recibe la clase `DbFirestoreService` inyectada. La variable `authenticationApi` recibe la clase `AuthenticationService` inyectada. Para agregar datos a la propiedad de flujo de `StreamController`, usó el método `sink.add()` y la propiedad de flujo emite los últimos eventos de flujo. Agregó métodos que llaman al servicio Cloud Firestore para recuperar una lista de diarios filtrados por el uid del usuario y para eliminar entradas de diario individuales.

Agregar el HomeBlocProvider

La clase `HomeBlocProvider` es responsable de pasar el Estado entre widgets y páginas usando la clase `InheritedWidget` como proveedor. El constructor `HomeBlocProvider` toma una clave, un widget y la variable `this.authenticationBloc`, que es la clase `HomeBloc`.

PRUÉBALO Creando el HomeBlocProvider

En esta sección, continúe editando el proyecto de diario. Agregará la clase `HomeBlocProvider` como proveedor de la clase `HomeBloc` para recuperar la lista de diarios y eliminar entradas individuales. La clase `Home Bloc` llama a la API de la base de datos de Cloud Firestore de la clase de servicio `DbFirestoreService`.

1. Cree un nuevo archivo Dart en la carpeta de bloques. Haga clic derecho en la carpeta de bloques, seleccione Nuevo Archivo Dart, ingrese `home_bloc_provider.dart` y haga clic en el botón Aceptar para guardar.
2. Importe el paquete `material.dart` y el paquete `home_bloc.dart` y cree la clase `HomeBlocProvider` que amplía la clase `InheritedWidget`.

```
importar 'paquete: flutter/material.dart'; import  
'paquete:diario/blocs/home_bloc.dart';  
  
clase HomeBlocProvider extiende InheritedWidget {  
  
}
```

3. Dentro de la clase `HomeBlocProvider`, declare el `HomeBloc` final `homeBloc` y el final Cadena de variables `uid`.

```
final HomeBloc homeBloc; uid  
de cadena final;
```

4. Agregue el constructor `HomeBlocProvider` con la palabra clave `const`.

5. Agregue al constructor los parámetros `key`, `child`, `this.homeBloc` y `this.uid`.

```
const HomeBlocProvider( {Clave  
clave, Widget hijo, este.homeBloc, este.uid}) : super(clave:  
clave, hijo: hijo);
```

6. Agregue el método HomeBlocProvider of (BuildContext context) con la palabra clave estática .
7. Dentro del método, devuelve HomeBlocProvider mediante el métodoHeredarFromWidgetOfExactType que permite que los widgets secundarios obtengan la instancia del proveedor HomeBlocProvider . HomeBlocProvider

```
estático de (contexto BuildContext) {  
    volver (context.inheritFromWidgetOfExactType(HomeBlocProvider)  
        como HomeBlocProvider);  
}
```

8. Agregue y anule el método updateShouldNotify para comprobar si homeBloc no es igual al antiguo HomeBlocProvider homeBloc. Si la expresión devuelve verdadero, el marco notifica a los widgets que contienen los datos heredados que necesitan reconstruir. Si la expresión devuelve un valor falso , el marco no envía una notificación ya que no es necesario reconstruir los widgets.

```
@override  
bool updateShouldNotify(HomeBlocProvider old) =>  
    homeBloc != antiguo.homeBloc;
```

CÓMO FUNCIONA

Para pasar el estado entre widgets y páginas, creó el archivo home_bloc_provider.dart que contiene la clase HomeBlocProvider como proveedor de la clase HomeBloc . El constructor de la clase HomeBlocProvider toma los parámetros key, widget, this.homeBloc y this.uid . El método of() devuelve el resultado del métodoHeredarFromWidgetOfExactType que permite que los widgets secundarios obtengan la instancia del proveedor HomeBlocProvider . El método updateShouldNotify comprueba si el valor ha cambiado y el marco notifica a los widgets para que se reconstruyan.

Agregar el JournalEditBloc El JournalEditBloc

es responsable de monitorear la página de edición del diario para agregar una entrada nueva o guardar una existente. Cuando se crea una instancia de JournalEditBloc , inicia los oyentes de StreamController que monitorean las secuencias de fecha, estado de ánimo, nota y botón de guardar.

PRUÉBALO Creando el JournalEditBloc

En esta sección, continúe editando el proyecto de diario . Agregará la clase JournalEditBloc para manejar la fecha, el estado de ánimo, la nota y el botón de guardar de la página de entrada del diario. JournalEditBloc también es responsable de llamar al servicio de base de datos de Cloud Firestore para guardar la entrada .

1. Cree un nuevo archivo Dart en la carpeta de bloques . Haga clic derecho en la carpeta de bloques , seleccione Nuevo → Archivo Dart, ingrese journal_entry_bloc.dart y haga clic en el botón Aceptar para guardar.
2. Importe la biblioteca async.dart ; importar las clases journal.dart, db_firestore_api.dart ; y cree la clase JournalEditBloc que acepte this.add, this.selectedJournal y this. parámetros dbApi . Cuando la variable add es verdadera, se crea una nueva entrada y cuando el valor es falso, se edita una entrada existente. El parámetro selectedJournal contiene la clase Journal

variables que contienen los valores de entrada seleccionados. this.dbApi recibe la clase DbFirestoreService inyectada .

```
importar 'dardo: asíncrono';
import 'paquete: diario/modelos/diario.dart'; importar 'paquete:
diario/servicios/db_firestore_api.dart';

JournalEditBloc(this.add, this.selectedJournal, this.dbApi) {

}
```

3. Dentro de la clase JournalEditBloc , declare la variable final DbApi dbApi . Añadir el bool final add variable y la variable Journal selectedJournal .

```
DbApi final dbApi; bool
final añadir;
Diario seleccionadoDiario;
```

4. Agregue la variable _dateController como String StreamController, agregue el captador de variable dateEditChanged como String Sink y agregue el captador dateEdit como String Stream. Tenga en cuenta que StreamController se inicializa con el flujo de transmisión () ya que tendremos múltiples oyentes.

```
final StreamController<String> _dateController = StreamController<String> .broadcast(); Sink<String> get
dateEditChanged
=> _dateController.sink; Stream<String> get dateEdit => _dateController.stream;
```

5. Siguiendo los pasos anteriores, agregue StreamControllers adicionales para manejar el estado de ánimo, nota, y guarde la llamada del diario al servicio Cloud Firestore.

```
final StreamController<String> _moodController = StreamController<String> .broadcast(); Sink<String> get
moodEditChanged => _moodController.sink; Stream<String> get moodEdit =>
_moodController.stream;

final StreamController<String> _noteController = StreamController<String> .broadcast(); Sink<String> get
noteEditChanged
=> _noteController.sink; Stream<String> get noteEdit => _noteController.stream;
```

```
final StreamController<String> _saveJournalController = StreamController<String> .broadcast(); Sink<String> get
saveJournalChanged
=> _saveJournalController.sink; Stream<String> get saveJournal => _saveJournalController.stream;
```

6. Agregue una nueva línea e ingrese el constructor JournalEditBloc que recibe el this.add inyectado, parámetros this.selectedJournal y this.dbApi . Tenga en cuenta que el parámetro inyectado this.dbApi recibe la clase DbFirestoreService .

7. Dentro del constructor JournalEditBloc , llame al método _startEditListeners() que crear en el paso 10.

8. Mediante el uso del operador punto, agregue la devolución de llamada `then((finished))` que llama al método `_getJournal()` que creó en el paso 12 y pase las variables `add` y `selectedJournal`.

```
JournalEditBloc(this.add, this.selectedJournal, this.dbApi) {  
    _startEditListeners().then((finished) => _getJournal(add, selectedJournal)); }
```

9. Agregue el método `dispose()` y llame a `_dateController`, `_moodController`, `_noteController` y `_saveJournalController close()` para cerrar el flujo de StreamController cuando no se necesitan.

```
anular disponer()  
{ _dateController.close();  
_moodController.close();  
_noteController.close();  
_saveJournalController.close(); }
```

10. Agregue el método asíncrono `Future<bool> _startEditListeners()` que es responsable de configurar cuatro oyentes para monitorear la fecha, el estado de ánimo, la nota y guardar secuencias.

11. Dentro de cada oyente `listen()`, se mantienen la fecha, el estado de ánimo y los valores de nota del diario seleccionado. Cuando se llama al oyente `_saveJournalController`, verifica si la acción == 'Guardar' y llama al método `_saveJournal()` que creará en el paso 10. La última línea del método agrega la declaración de retorno verdadero que indica que todos los oyentes se han iniciado. .

```
Future<bool> _startEditListeners() async  
{ _dateController.stream.listen((date)  
{ selectedJournal.date = date; });  
  
_moodController.stream.listen((mood)  
{ selectedJournal.mood =  
  
mood; }); _noteController.stream.listen((nota)  
{ selectedJournal.note = nota; });  
  
_saveJournalController.stream.listen((acción) { if (acción  
== 'Guardar')  
{ _saveJournal(); } });  
  
devolver verdadero;  
}
```

12. Agregue el método `_getJournal` que toma los parámetros `bool add` y `Journal`.

13. Agregue una declaración `if` para verificar si el valor agregado es verdadero, lo que significa que se va a crear una nueva entrada, y establezca los valores predeterminados para la variable `Journal` seleccionada .

14. Agregue una declaración `else` para manejar la edición de una entrada existente y establezca los valores predeterminados de la variable de diario existente pasada .

15. Después de la declaración `if-else`, notifique a los StreamControllers agregando los valores de fecha, estado de ánimo y nota a través de cada método `sink.add()`.

```
void _getJournal(bool add, Journal journal) { if (add)  
{ selectedJournal = Journal();  
    DiarioSeleccionado.fecha = FechaHora.ahora().toString();  
    selectedJournal.mood = 'Muy Satisfecho';  
    DiarioSeleccionado.nota = '';  
    diarioseleccionado.uid = diario.uid; } else  
{ diario  
    seleccionado.fecha = diario.fecha;  
    diarioseleccionado.estado de ánimo = diario.estado  
de ánimo; diarioseleccionado.nota = diario.nota;  
}  
} dateEditChanged.add(selectedJournal.date);  
moodEditChanged.add(selectedJournal.mood);  
noteEditChanged.add(selectedJournal.note);  
}
```

16. Agregue el método `_saveJournal()` para crear una variable de diario que contenga el valores de entrada.

17. Llame al constructor `DateTime.parse()` para convertir la fecha al estándar ISO 8601.

18. Agregue un operador ternario para verificar si la variable de agregar es igual a verdadero y llame al método `dbApi .addJournal(journal)` para crear una nueva entrada de diario. De lo contrario, llame al método `dbApi .updateJournal(journal)` para actualizar la entrada actual.

```
vacío _saveJournal() {  
    diario diario = diario(  
        ID de documento: diario seleccionado. ID de  
        documento, fecha: fecha y hora. parse (diario seleccionado. fecha). a Iso8601String  
    (), estado de ánimo: diario  
    seleccionado. estado de ánimo, nota:  
    diario seleccionado. nota, uid:  
  
    diario seleccionado. uid,); agregar ? dbApi.addJournal(diario) : dbApi.updateJournal(diario);  
}
```

CÓMO FUNCIONA

Para monitorear la página de edición del diario para agregar una nueva entrada o guardar una entrada existente, creó el archivo `journal_edit_bloc.dart` que contiene la clase `JournalEditBloc`. Declaró una referencia a la clase `DbApi` para obtener acceso a la API de la base de datos de Cloud Firestore. La variable `dbApi` recibe la clase `DbFire storeService` inyectada . Para agregar datos a la propiedad de flujo de `StreamController` , usó el sumidero. `add()` y la propiedad de flujo emite los últimos eventos de flujo . Agregó métodos que llaman al servicio Cloud Firestore para crear una nueva entrada o guardar una entrada existente.

Adición de `JournalEditBlocProvider`

La clase `JournalEditBlocProvider` es responsable de pasar el estado entre widgets y páginas utilizando la clase `InheritedWidget` como proveedor. El constructor `JournalEditBlocProvider` toma las variables `Key`, `Widget` y `this.journalEditBloc` . Tenga en cuenta que la variable `this.journalEditBloc` es la clase `JournalEditBloc` .

PRUÉBALO Creando JournalEditBlocProvider

En esta sección, continúe editando el proyecto de diario . Agregará la clase JournalEditBlocProvider como proveedor de la clase JournalEditBloc para agregar o editar entradas individuales. La clase JournalEdit Bloc llama a la API de la base de datos de Cloud Firestore de la clase de servicio DbFirestoreService .

1. Cree un nuevo archivo Dart en la carpeta de bloques . Haga clic con el botón derecho en la carpeta de bloques , seleccione Nuevo Archivo Dart, ingrese journal_edit_bloc_provider.dart y haga clic en el botón Aceptar para guardar.
2. Importe el paquete material.dart , el paquete journal_edit_bloc.dart y el diario .dart y cree la clase JournalEditBlocProvider que amplía la clase InheritedWidget .

```
importar 'paquete: flutter/material.dart'; import  
'paquete:journal/blocs/journal_edit_bloc.dart';  
  
clase JournalEditBlocProvider extiende InheritedWidget {  
  
}
```

3. Dentro de la clase JournalEditBlocProvider , declare el JournalEditBloc final journalEdit Variables de bloque, agregado bool final y diario final .

```
final JournalEditBloc journalEditBloc;
```

4. Agregue el constructor JournalEditBlocProvider con la palabra clave const .

5. Agregue al constructor la clave, child, this.journalEditBloc, this.add y this.journal parámetros

```
const JournalEditBlocProvider( {Clave  
clave, Widget hijo, this.journalEditBloc}) : super(clave: clave,  
hijo: hijo);
```

6. Agregue el método JournalEditBlocProvider of (BuildContext context) con la palabra clave estática .

7. Dentro del método, devuelva el JournalEditBlocProvider usando la propiedad inheritFromWidgetOf Método ExactType que permite que los widgets secundarios obtengan la instancia del proveedor JournalEditBlocProvider .

```
static JournalEditBlocProvider of(BuildContext context) { return  
(context.inheritFromWidgetOfExactType(JournalEditBlocProvider) as  
JournalEditBlocProvider); }
```

8. Agregue y anule el método updateShouldNotify para comprobar si journalEditBloc no es igual al antiguo JournalEditBlocProvider journalEditBloc. Si la expresión devuelve verdadero, el marco notifica a los widgets que contienen los datos heredados que necesitan reconstruir.

```
@override  
bool updateShouldNotify(JournalEditBlocProvider old) => false;
```

CÓMO FUNCIONA

Para pasar el estado entre widgets y páginas, creó el archivo `journal_edit_bloc_provider.dart` que contiene la clase `JournalEditBlocProvider` como proveedor de la clase `JournalEditBloc`.

El constructor de la clase `JournalEditBlocProvider` toma la clave, el widget y los parámetros `this.journalEditBloc`. El método `of()` devuelve el resultado del método `HeredarFromWidgetOfExactType` que permite que los widgets secundarios obtengan la instancia del proveedor `JournalEditBlocProvider`. El método de actualización `ShouldNotify` verifica si el valor ha cambiado y el marco notifica a los widgets para que se reconstruyan.

RESUMEN

En este capítulo, implementó la administración del estado local y de toda la aplicación del cliente. Aprendió a implementar el patrón BLoC para separar la lógica empresarial de las páginas de la interfaz de usuario. Creó una clase abstracta para definir la interfaz de autenticación y la clase de servicio de autenticación para implementar la clase abstracta. Al usar la clase abstracta, puede imponer restricciones de implementación y diseño. Usó la clase abstracta con la clase BLoC para injectar en tiempo de ejecución la clase adecuada dependiente de la plataforma, lo que resultó en que la clase BLoC fuera independiente de la plataforma.

Implementó una clase `InheritedWidget` como proveedor para pasar el estado entre los widgets y las páginas. Usó el método `of()` para acceder a la referencia al proveedor. Creó la clase de modelo de diario para estructurar los registros de diario individuales y usó el método `fromDoc()` para convertir y asignar un documento de base de datos de Cloud Firestore a una entrada de diario individual. Creó clases de servicio para administrar el envío y la recepción de llamadas a la API del servicio. Creó la clase `AuthenticationService` implementando la clase abstracta `AuthenticationApi` para acceder a la API de autenticación de Firebase.

Creó la clase `DbFirestoreService` implementando la clase abstracta `DbApi` para acceder a la API de la base de datos de Cloud Firestore.

Implementó el patrón BLoC para maximizar la separación de los widgets de la interfaz de usuario y los componentes de la lógica empresarial. Aprendió que el patrón expone sumideros a datos de entrada y expone flujos a datos de salida. Aprendió a injectar las clases de servicio compatibles con la plataforma en el constructor de BLoC, lo que hace que la plataforma de clases de BLoC sea independiente. Al separar la lógica comercial de la interfaz de usuario, no importa si usa Flutter para las aplicaciones móviles, AngularDart para las aplicaciones web o cualquier otra plataforma. Implementó `StreamController` para enviar datos, eventos realizados y errores en la propiedad de flujo. Implementó la clase `Sink` para agregar datos con la propiedad `Sink` y la clase `Stream` para enviar datos con la propiedad `Stream` de `StreamController`.

En el próximo capítulo, aprenderá cómo implementar páginas reactivas para comunicarse con los BLoC. Modificará la página principal para implementar la gestión de estado de toda la aplicación mediante el proveedor de autenticación BLoC. Creará la página de inicio de sesión e implementará el BLoC para validar el correo electrónico y la contraseña, iniciar sesión y crear una nueva cuenta de usuario. Modificará la página de inicio para implementar la base de datos BLoC y usar el constructor `ListView.separated`. Creará la página de entrada de edición de diario e implementará el BLoC para crear y actualizar entradas existentes.

LO QUE APRENDISTE EN ESTE CAPÍTULO

TEMA	CONCEPTOS CLAVE
Administración de toda la aplicación y del estado local	Aprendió a aplicar la gestión de estado creando la clase InheritedWidget como proveedor para pasar el estado entre widgets y páginas.
clase abstracta	Aprendió a implementar una clase abstracta y métodos abstractos para definir la interfaz de autenticación. Creó la clase AuthenticationService que implementa la clase abstracta .
clase de modelo	Aprendió a crear la clase de modelo Journal responsable de modelar la estructura de datos y mapear la base de datos QuerySnapshot de Cloud Firestore a entradas individuales de Journal .
clases de servicio	Aprendió a crear clases de servicio que llaman a diferentes API de servicios. Ha creado el servicio de autenticación. clase para llamar a la API de autenticación de Firebase y al Clase DbFirestoreService para llamar a Cloud Firestore API de base de datos.
Clase de validador	Aprendió a crear la clase Validators que usa StreamTransformer para validar el correo electrónico y la contraseña para cumplir con los requisitos mínimos.
StreamController, Corrientes, Fregaderos y StreamBuilder	Aprendió a usar StreamController para enviar datos, eventos realizados y errores en la propiedad de flujo . StreamController tiene una propiedad de receptor (entrada) y una propiedad de flujo (salida).
patrón BLoC	El acrónimo BLoC significa Business Logic Component y se creó para definir una interfaz independiente de la plataforma para la lógica empresarial. En otras palabras, separa la lógica comercial de los widgets/componentes de la interfaz de usuario.
Clases BLoC	Aprendió a crear las clases AuthenticationBloc, LoginBloc, HomeBloc y JournalEditBloc BLoC.
Inyección de dependencia BLoC	Aprendiste a usar clases abstractas con el BLoC clases y cómo inyectar clases dependientes de la plataforma a las clases BLoC, haciendo que las clases BLoC sean independientes de la plataforma.

TEMA	CONCEPTOS CLAVE
Clase InheritedWidget como proveedor	Aprendiste a crear el Clases AuthenticationBlocProvider, HomeBlocProvider y JournalEditBlocProvider que amplían la clase InheritedWidget para actuar como proveedores de las clases AuthenticationBloc, HomeBloc y JournalEditBloc .

16 dieciséis

Agregar BLoC a Firestore

Páginas de la aplicación del cliente

LO QUE APRENDERÁS EN ESTE CAPÍTULO

- Cómo pasar la gestión de estado de toda la aplicación entre páginas
- Cómo aplicar la gestión de estado local en el árbol de widgets
- Cómo aplicar InheritedWidget como proveedor para pasar el estado entre widgets y páginas
- Cómo usar la inyección de dependencia para injectar clases de servicio al Clases BLoC para lograr la independencia de la plataforma
- Cómo aplicar la clase LoginBloc a la página de inicio de sesión
- Cómo aplicar la clase AuthenticationBloc para administrar las credenciales de usuario para la administración del estado de toda la aplicación
- Cómo aplicar la clase HomeBloc a la página de inicio para enumerar, agregar y eliminar entradas de diario
- Cómo aplicar JournalEditBloc a la página de edición de la revista para agregar o modificar una entrada existente
- Cómo crear widgets reactivos implementando StreamBuilder artilugio
- Cómo usar el constructor ListView.separated para crear una lista de entradas de diario con una línea divisoria usando el widget Divider()
- Cómo usar el widget Descartable para deslizar y eliminar una entrada
- Cómo usar la propiedad ConfirmDismiss del widget Descartable para solicitar un cuadro de diálogo de confirmación de eliminación

Cómo usar el widget DropdownButton() para presentar una lista de estados de ánimo con el título, el color y la rotación de iconos

Cómo aplicar la clase MoodIcons para recuperar el título, el color, la rotación y el ícono del estado de ánimo

Cómo aplicar el método RotateZ() de Matrix4 para rotar íconos según el estado de ánimo junto con la clase MoodIcons

Cómo aplicar la clase FormatDates para formatear fechas

En este capítulo, continuará editando y completando la aplicación de diario de estado de ánimo en la que ha trabajado en los Capítulos 14 y 15. Para su comodidad, puede usar el proyecto ch15_final_journal como punto de partida y asegúrese de agregar su GoogleService-Info.plist al proyecto Xcode y el archivo google-services.json al proyecto de Android que descargó en el Capítulo 14 desde su consola Firebase.

Aprenderá a aplicar las clases BLoC, servicio, proveedor, modelo y utilidad a las páginas de widgets de la interfaz de usuario. La ventaja de utilizar el patrón BLoC permite la separación de los widgets de la interfaz de usuario de la lógica empresarial. Aprenderá a usar la inyección de dependencia para inyectar clases de servicio en las clases de BLoC.

Mediante el uso de inyección de dependencia, los BLoC siguen siendo independientes de la plataforma.

También aprenderá a aplicar la administración de estado de toda la aplicación mediante la implementación de la clase Authentication BlocProvider en la página principal. Aprenderá a pasar el estado entre las páginas y el árbol de widgets mediante la implementación de las clases HomeBlocProvider y JournalEditBlocProvider . Aprenderá a crear una página de inicio de sesión que implemente la clase LoginBloc para validar correos electrónicos, contraseñas y credenciales de usuario. Modificará la página de inicio y aprenderá a implementar la clase HomeBloc para manejar la lista de entradas del diario y agregar y eliminar entradas individuales. Aprenderá a crear la página de edición del diario que implementa la clase JournalEditBloc para agregar, modificar y guardar entradas existentes.

AGREGAR LA PÁGINA DE INICIO DE SESIÓN

La página de inicio de sesión contiene un campo de texto para ingresar la dirección de correo electrónico y un campo de texto para ingresar la contraseña oscureciendo los caracteres por motivos de privacidad. También agregará un botón para iniciar sesión en el usuario y un botón para crear una nueva cuenta de usuario. Aprenderá a implementar la clase LoginBloc mediante el widget StreamBuilder . Aprenderá a usar la inyección de dependencia para inyectar la clase AuthenticationService() en el constructor de la clase LoginBloc , lo que da como resultado que LoginBloc sea independiente de la plataforma. Consulte la Figura 16.1 para tener una idea de cómo se verá la página de inicio de sesión final.

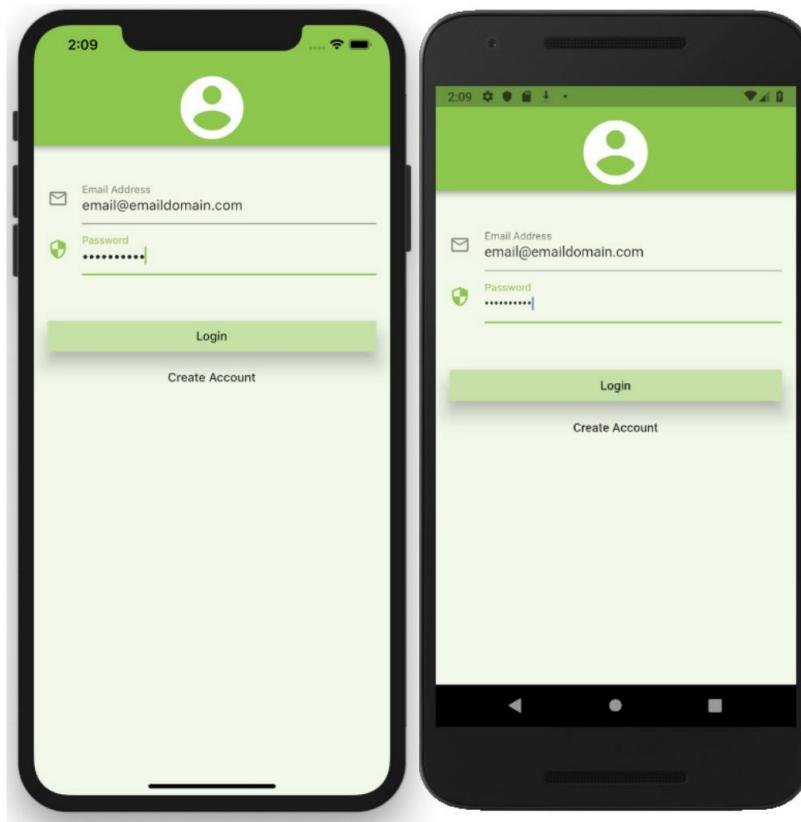


FIGURA 16.1: Página de inicio de sesión final

PRUÉBALO Creación de la página de inicio de sesión

En esta sección, continuará editando el proyecto del diario . Agregará la clase LoginBloc para manejar la validación de los valores de correo electrónico y contraseña. Utilizará la clase LoginBloc para iniciar sesión en el usuario o crear una nueva cuenta de usuario e iniciar sesión con las nuevas credenciales de autenticación.

1. Cree un nuevo archivo Dart en la carpeta de páginas . Haga clic derecho en la carpeta de páginas , seleccione Nuevo Archivo Dart, ingrese login.dart y haga clic en el botón Aceptar para guardar.
2. Importe las clases material.dart, login_bloc.dart y authentication.dart . Agregar un nuevo línea y cree la clase de inicio de sesión que extiende un StatefulWidget.

```
importar 'paquete: flutter/material.dart'; importar  
'paquete: diario/blocs/login_bloc.dart'; import 'paquete:  
diario/servicios/autenticación.dart';
```

```
El inicio de sesión de clase extiende StatefulWidget
{ @override
    _LoginState createState() => _LoginState(); }
```

```
class _LoginState extends StatelessWidget<Inicio de sesión>
{ @override
    void initState() {
        // Compilación del widget (contexto BuildContext) {
        // devolver Contenedor();
    }
}
```

3. Modifique la clase `_LoginState` y agregue la variable privada `LoginBloc _loginBloc`.
4. Anule `initState()` e inicialice la variable `_loginBloc` con la clase `LoginBloc(AuthenticationService())` inyectando `AuthenticationService()` en el constructor.

Tenga en cuenta que la razón por la que inicializó la variable `_loginBloc` desde `initState()` y no desde `didChangeDependencies()` es porque `LoginBloc` no necesita un proveedor (`InheritedWidget`).

```
class _LoginState extends StatelessWidget<Inicio de sesión> {
    LoginBloc _loginBloc;

    @anular
    void initState() {
        super.initState();
        _loginBloc = LoginBloc(AuthenticationService());
    }

    @anular
    void didChangeDependencies() {
        // devolver Contenedor();
    }
}
```

5. Anule el método `dispose()` y elimine la variable `_loginBloc`. Se llama al método `dispose()` de la clase `LoginBloc` y se cierran todos los `StreamControllers`.

```
@anular
anular disponer()
{
    _loginBloc.dispose();
    super.dispose();
}
```

6. En el método `Widget build()`, reemplace `Container()` con los widgets UI `Scaffold` y `AppBar` y para la propiedad del cuerpo agregue `SafeArea()` y `SingleChildScrollView()` con la propiedad secundaria como `Column()`. Establezca la propiedad `Column crossAxisAlignment` para estirar.
7. Para la propiedad inferior de `AppBar`, agregue un widget `PreferredSize` con la propiedad secundaria establecida en `Icons.account_circle` y establezca la propiedad de tamaño en 88,0 píxeles y la propiedad de color en `Colores.blanco`.

```
@anular
void crearWidgets(BuildContext context) {
    Scaffold(
```

```
abajo: PreferredSize( child:

Icon( Icons.account_circle, size:
88.0, color:
Colors.white, ),ferredSize:
Size.fromHeight(40.0),

), ), cuerpo: SafeArea(
hijo: SingleChildScrollView( relleno:
EdgeInsets.all(16.0, 32.0, 16.0, 16.0), hijo: Columna( crossAxisAlignment:
CrossAxisAlignment.stretch, children: <Widget>[

], ), ), ), );

}

8. Agregue a la propiedad Column children dos widgets de StreamBuilder . El primer StreamBuilder maneja el TextField de correo electrónico y establece la propiedad de flujo en el flujo _loginBloc.email. Agregue a la propiedad del constructor el widget TextField y, para la propiedad InputDecoration errorText , configúrelo en snapshot.error. La propiedad onChanged llama al receptor _loginBloc .emailChanged.add que pasa la dirección de correo electrónico actual.
```

Agregue el segundo StreamBuilder siguiendo los pasos anteriores para manejar la contraseña TextField. Agregue un SizedBox con la propiedad de altura establecida en 48,0 píxeles y agregue una llamada al método _buildLoginAndCreateButtons() que creará en el paso 9.

```
Columna( crossAxisAlignment: CrossAxisAlignment.stretch, children:
<Widget>[ StreamBuilder(
flujo: _loginBloc.email, constructor:
(contexto BuildContext, instantánea AsyncSnapshot) => TextField(tipo de teclado: TextInputType.emailAddress,
decoration: InputDecoration(
labelText: 'Dirección de correo electrónico',
icon: Icon(Icons.mail_outline), errorText:
snapshot.error),
onChanged: _loginBloc.emailChanged.add, ),

),
StreamBuilder(
flujo: _loginBloc.password, constructor:
(contexto BuildContext, instantánea AsyncSnapshot) => TextField(texto oscuro: verdadero,
decoration:
InputDecoration(textoetiqueta:
'Contraseña',
```

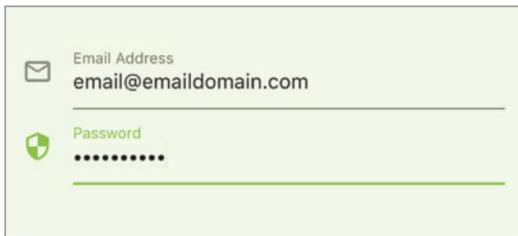
```

icono: Icon(Icons.security),
errorText: snapshot.error),
onChanged: _loginBloc.passwordChanged.add, ),

),

SizedBox (altura: 48,0),
_buildLoginAndCreateButtons(), ], ),

```



9. Agregue una nueva línea después del método Widget build (BuildContext context) y agregue _buildLoginAndCreateButtons() que devuelve un Widget. Agregue StreamBuilder para manejar qué conjunto de botones está activo, por ejemplo, el botón Iniciar sesión y luego Crear cuenta, o en orden inverso. StreamBuilder comprueba si snapshot.data == 'Iniciar sesión' y llama al método _buttonsLogin() ; de lo contrario, llama al método _buttonsCreateAccount() . Creará cada método, respectivamente, en los pasos 10 y 11.

```

Widget _buildLoginAndCreateButtons() {
devuelve StreamBuilder(
initialData: 'Iniciar sesión',
flujo: _loginBloc.loginOrCreateButton, constructor:
((Contexto BuildContext, instantánea AsyncSnapshot) {
if (instantánea.datos == 'Iniciar sesión')
{ return _buttonsLogin(); } else
if (instantánea.datos == 'Crear cuenta') {
volver _botonesCrearCuenta();

} }}, );
}

```

10. Agregue el método _buttonsLogin() que maneja la devolución de la combinación del botón con Iniciar sesión primero y Crear cuenta después. Dado que el botón Iniciar sesión es el predeterminado, agregue el widget RaisedButton y, para el botón Crear cuenta , agregue el widget FlatButton .

```

Columna _buttonsLogin()
{ return
Column( crossAxisAlignment: CrossAxisAlignment.stretch,
children:

```

```

<Widget>[ StreamBuilder( initialData: false, stream: _loginBloc.enableLoginCreateButton,

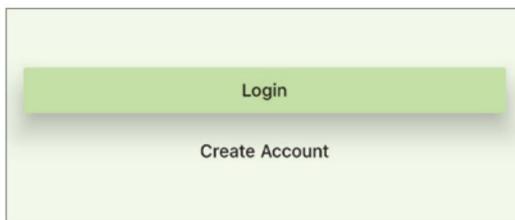
```

```

constructor: (contexto BuildContext, instantánea AsyncSnapshot) => RaisedButton
(elevación: 16.0,
niño: Texto ('Iniciar
sesión'), color:
Colors.lightGreen.shade200, disabledColor:
Colors.grey.shade100, onPressed: snapshot.data ? ()
=>
_loginBloc.loginOrCreateChanged.add('Iniciar sesión') : nulo,
),
),
FlatButton( child:
Texto('Crear Cuenta'), onPressed: () {
>Login Bloc.loginOrCreateButtonChanged.add('Crear cuenta'); }, ), ], );
}

}

```



11. Agregue el método `_buttonsCreateAccount()` que maneja la devolución de la combinación del botón con Create Account primero e Login segundo. Dado que el botón Crear cuenta es el predeterminado, agregue el widget RaisedButton y, para el botón Iniciar sesión , agregue el widget FlatButton .

```

Columna _buttonsCreateAccount() { return
Column( crossAxisAlignment: CrossAxisAlignment.stretch, children:
<Widget>[ StreamBuilder( initialData:
false, stream: _loginBloc.enableLoginCreateButton, builder:
(contexto BuildContext, instantánea AsyncSnapshot) => RaisedButton( elevación: 16.0,
child : Text('Crear
cuenta'), color:
Colors.lightGreen.shade200, disabledColor:
Colors.grey.shade100, onPressed: snapshot.data ?
() => _loginBloc.loginOrCreateChanged.add('Crear
cuenta'): nulo,
),
),
),

```

```
FlatButton( child:  
    Text('Iniciar sesión'), onPressed:  
    ()  
        { _loginBloc.loginOrCreateButtonChanged.add('Iniciar sesión'); }, ], );  
  
}
```



CÓMO FUNCIONA

El archivo login.dart contiene la clase de inicio de sesión que amplía un StatefulWidget para manejar el inicio de sesión de un usuario o la creación de una nueva cuenta de usuario. Crea una instancia de la clase LoginBloc inyectando la clase AuthenticationService() anulando el método initState(). Tenga en cuenta que la razón por la que inicializó la variable _loginBloc desde initState() y no desde didChangeDependencies() es porque LoginBloc no necesita un proveedor (InheritedWidget). Cerró los oyentes de los controladores de transmisión de LoginBloc anulando el método dispose() y llamando al método _loginBloc.dispose(). Es una buena práctica cerrar los oyentes de StreamController cuando no se necesitan.

El widget StreamBuilder se usa para monitorear los valores de correo electrónico y contraseña. Usó la propiedad onChanged del widget TextField para llamar al receptor _loginBloc.emailChanged.add y al _loginBloc.passwordChanged.add sumidero para enviar los valores a la clase LoginBloc Validators para validar que se cumplen los requisitos mínimos de formato. Aprendió en la sección "Agregar la clase de validadores" del Capítulo 15 cómo el StreamTransformer transforma un Stream que se usa para validar y procesar valores dentro de un Stream.

El widget StreamBuilder se usa para escuchar el flujo _loginBloc.loginOrCreateButton para alternar entre mostrar Iniciar sesión o Crear cuenta como el botón predeterminado.

MODIFICACIÓN DE LA PÁGINA PRINCIPAL

La página principal es el centro de control responsable de monitorear la administración del estado de toda la aplicación. Aprenderá a implementar la clase AuthenticationBlocProvider como proveedor principal de la clase AuthenticationBloc. Aprenderá a implementar la clase HomeBlocProvider como proveedor de la clase HomeBloc con la propiedad secundaria como clase Home y también a mantener el estado.

para el usuario uid. Aprenderá a aplicar el widget StreamBuilder para monitorear el estado de autenticación del usuario. Cuando el usuario inicia sesión, el widget dirige al usuario a la página de inicio, y cuando el usuario cierra sesión, lo dirige a la página de inicio de sesión. Consulte la Figura 16.2 para ver el flujo de la página principal de BLoC.

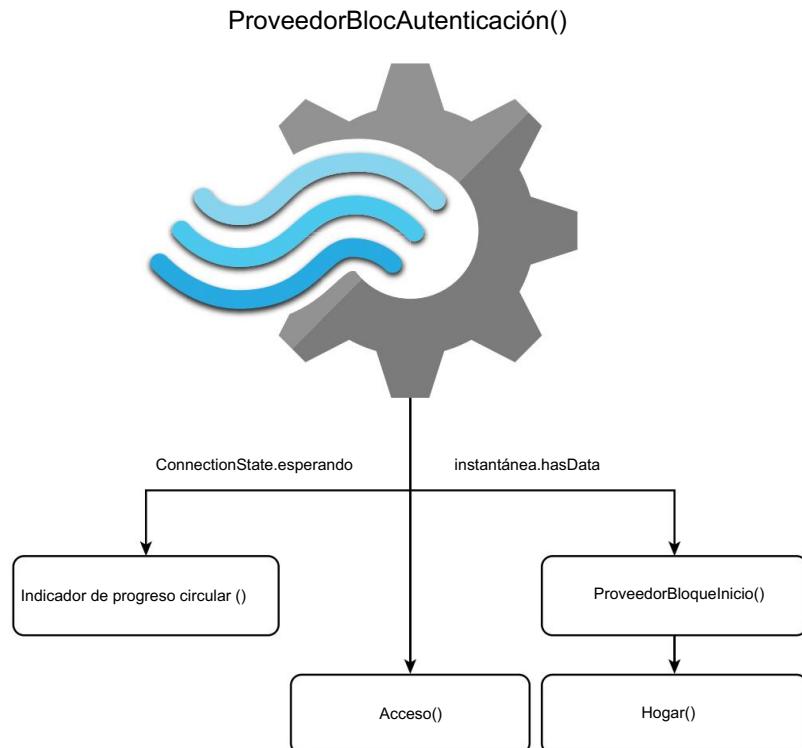


FIGURA 16.2: Flujo BLoC de la página principal

PRUÉBELO Modificación de la página principal En esta

sección, continuará editando el proyecto del diario . Modificará el archivo main.dart para manejar la administración de estado de toda la aplicación mediante StreamBuilder y BLoC.

1. Abra el archivo main.dart y agregue los siguientes paquetes y clases:

```
importar 'paquete: flutter/material.dart'; import
'paquete:jurnal/blocs/authentication_bloc.dart'; import 'paquete:jurnal/
blocs/authentication_bloc_provider.dart'; import 'paquete:diario/blocs/
home_bloc.dart'; import 'paquete:jurnal/blocs/
home_bloc_provider.dart'; import 'paquete: diario/servicios/
autenticación.dart'; import 'paquete:jurnal/services/db_firestore.dart';
import 'paquete:diario/páginas/inicio.dart'; importar 'paquete:
diario/páginas/login.dart';
```

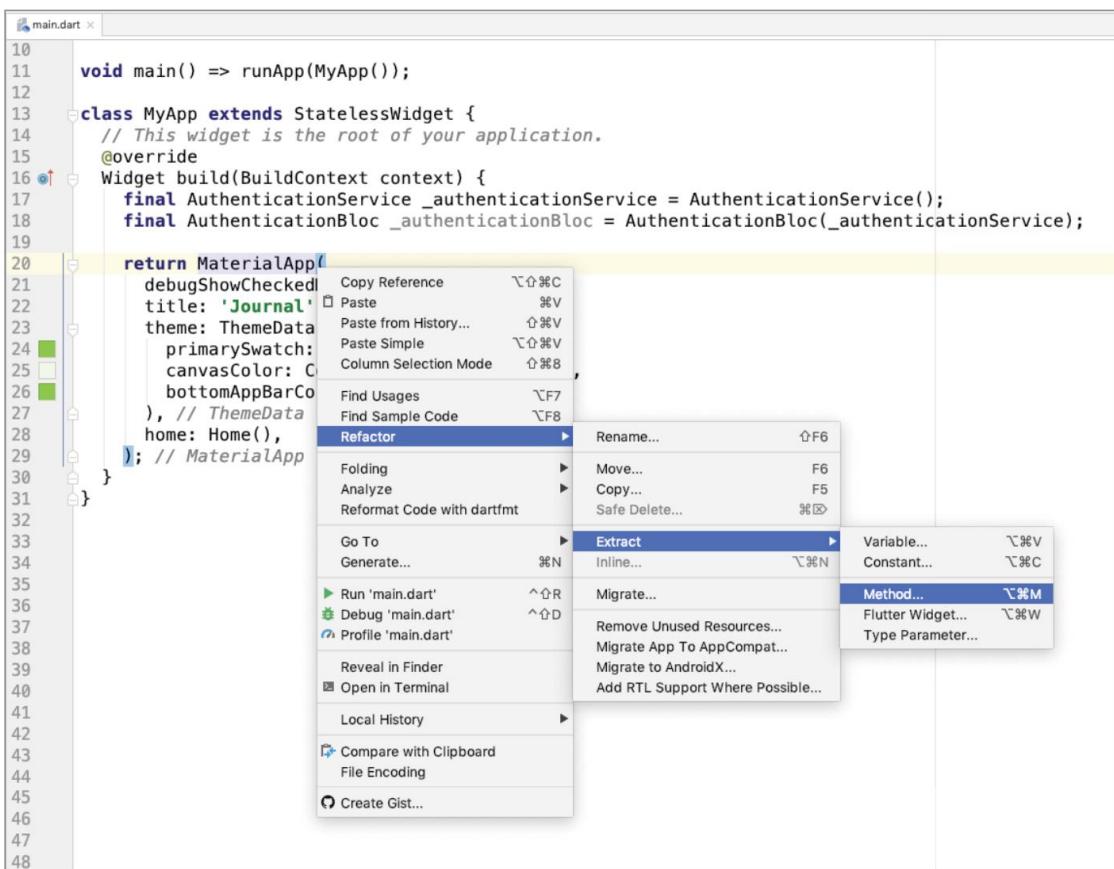
2. Dentro del método Widget build() , agregue la variable final _authenticationService inicializada por el Servicio de Autenticación().

```
@anular
Compilación del widget (contexto BuildContext) {
    Servicio de autenticación final _servicio de autenticación = Servicio de autenticación ();
```

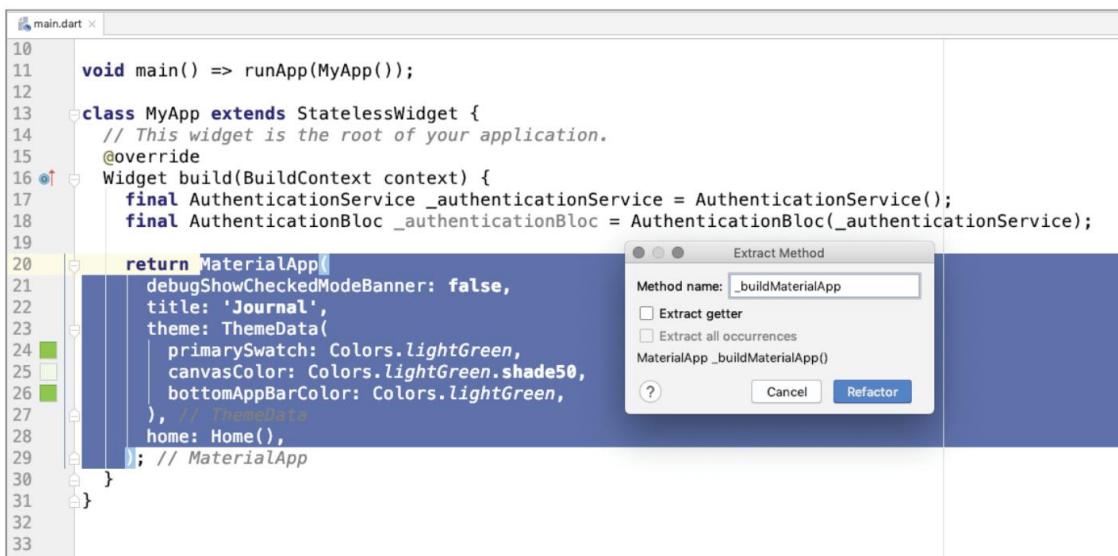
3. Agregue la variable _authenticationBloc final inicializada por AuthenticationBloc() e inyecte la dependencia _authenticationService . La clase AuthenticationBloc sigue siendo independiente de la plataforma mediante el uso de inyección de dependencia de la clase AuthenticationService() .

```
bloque de autenticación final _authenticationBloc = bloque de autenticación
(_servicio de autenticación);
```

4. Refactorice el widget MaterialApp() a un método colocando el mouse sobre la palabra MaterialApp y haciendo clic con el botón derecho. Luego seleccione Refactorizar Extraer Método Método.



5. En el cuadro de diálogo Extraer método, ingrese `_buildMaterialApp` como nombre del método y haga clic en el Botón de refactorización.



6. Agregue al constructor `_buildMaterialApp()` el parámetro Página de inicio del widget . Cuando esto se llama al método, el widget de página se pasa al parámetro `homePage` , por ejemplo, la página de clase `Login()` o `Home()` . Cambie la propiedad de inicio al widget de página de inicio .

```

MaterialApp _buildMaterialApp(Widget HomePage) { return
  MaterialApp(
    debugShowCheckedModeBanner: false,
    title: 'Security Inherited', theme: ThemeData(
      PrimarySwatch: Colors.lightGreen, canvasColor:
      Colors.lightGreen.shade50, bottomAppBarColor:
      Colors.lightGreen, ), home: HomePage, );
}

```

7. Vuelva al método de compilación Widget y reemplace la llamada `return _buildMaterialApp()` con la clase `AuthenticationBlocProvider()` .

8. Para la propiedad `authenticationBloc` , pase la variable `_authenticationBloc` y para la propiedad secundaria , pase el widget `StreamBuilder()` .

```

devolver ProveedorBlocAutenticación(
  authenticationBloc: _authenticationBloc, child: StreamBuilder(), );

```

9. Establezca la propiedad `initialData` del widget `StreamBuilder()` en un valor nulo , lo que significa que ningún usuario ha iniciado sesión. Establezca la propiedad de secuencia en la secuencia `_authenticationBloc.user`.

10. Mientras espera el estado de conexión de la instantánea, es una buena práctica mostrar un indicador de progreso.

De lo contrario, el usuario cree que la aplicación dejó de funcionar. Dentro de la propiedad del constructor , agregue una declaración `if` que verifique que `snapshot.connectionState == ConnectionState.waiting` y devuelva un widget Container con la propiedad de color establecida en `Colors.lightGreen` y la propiedad secundaria en el widget `CircularProgressIndicator()` .

11. Cuando el usuario inicia sesión con las credenciales correctas, debe navegar a la página de la clase `Home()` .

Agregue una declaración `else if` que verifique si `snapshot.hasData` es igual a verdadero y agregue la clase `return HomeBlocProvider()` .

12. Establezca la propiedad `homeBloc` de la clase `HomeBlocProvider()` en la clase `HomeBloc(DbFirestoreService(), _authenticationService)` ; tenga en cuenta que está inyectando las clases de servicio de la plataforma Flutter.

13. Para la propiedad secundaria , llame al método `_buildMaterialApp(Home())` para navegar a `Home()` página de clase.

14. Agregue la última declaración `else` y agregue el método `return _buildMaterialApp(Login())` para navegar a la página de la clase `Login()` .

```
StreamBuilder( initialValue:  
    null, stream: _authenticationBloc.user,  
    builder: (contexto BuildContext, instantánea AsyncSnapshot) {  
        if (snapshot.connectionState == ConnectionState.waiting) { return  
            Container( color:  
                Colors.lightGreen, child:  
                CircularProgressIndicator(), );  
  
        } else if (snapshot.hasData) { return  
            HomeBlocProvider( homeBloc:  
                HomeBloc(DbFirestoreService(), _authenticationService), uid: snapshot.data,  
                child:  
                _buildMaterialApp(Home()), ); } else  
  
        { return  
            _buildMaterialApp(Iniciar sesión());  
  
    } }, ),
```

CÓMO FUNCIONA

Editó el archivo `main.dart` que contiene la clase `MyApp` que extiende un `StatelessWidget` para manejar la administración del estado de autenticación en toda la aplicación. La raíz de la clase `MyApp` es la clase `Authentication BlocProvider` con la propiedad secundaria establecida en el widget `StreamBuilder` . Este widget de `StreamBuilder` supervisa el flujo `_authenticationBloc.user` y el constructor vuelve a generar cada vez que cambia el estado de autenticación del usuario y lleva al usuario al inicio de sesión o página de inicio correspondiente.

MODIFICAR LA PÁGINA DE INICIO

La página de inicio es responsable de mostrar una lista de entradas de diario filtradas por el uid del usuario que inició sesión y la capacidad de agregar, modificar y eliminar entradas individuales. En esta sección, aprenderá a implementar AuthenticationBlocProvider como proveedor de la clase AuthenticationBloc . Aprenderá a implementar la clase HomeBlocProvider como proveedor de la clase HomeBloc . Aprenderá a aplicar el widget StreamBuilder para crear la lista de entradas de diario llamando al constructor ListView .separated .

El widget Descartable se usa para deslizar el dedo sobre un elemento para eliminar la entrada. Aprenderá a usar la propiedad confirmDismiss del widget Descartable para solicitar al usuario un cuadro de diálogo de confirmación de eliminación. Aprenderá cómo llamar a la clase MoodIcons para colorear y rotar iconos según el estado de ánimo seleccionado. Aprenderá cómo llamar a la clase FormatDates para dar formato a las fechas.

Tenga en cuenta que no inyecté deliberadamente las clases MoodIcons y FormatDates en las clases HomeBloc y EditJournalBloc para mostrar que puede decidir no incluir ciertas clases de utilidad en el BLoC. Si los incluyera en estos BLoC, duplicaría los métodos MoodIcons y Format Dates en dos BLoC diferentes. En la sección "Implementación del patrón BLoC" del Capítulo 15, aprendió en las pautas de diseño de la interfaz de usuario cómo crear un BLoC para cada componente lo suficientemente complejo.

Otra opción es crear una clase MoodAndDatesBloc para manejar las clases MoodIcons y FormatDates (como clases abstractas) y usar la clase MoodAndDatesBloc en las páginas de inicio y de edición. Vea la Figura 16.3 para tener una idea de cómo se verá la página de inicio final.



FIGURA 16.3: Página de inicio final

PRUÉBALO Modificación de la página de inicio

En esta sección, continuará editando el proyecto del diario . Estará modificando el archivo home.dart para manejar la lista de las entradas del diario, además de agregarlas, modificarlas y eliminarlas. Utilizará la clase MoodIcons para recuperar el título, el color, la rotación y el ícono correctos del estado de ánimo. Utilizará la clase FormatDates para dar formato a las fechas.

1. Abra el archivo home.dart y agregue los siguientes paquetes y clases:

```
importar 'paquete: flutter/material.dart'; import
'paquete:journal/blocs/authentication_bloc.dart'; import 'paquete:journal/
blocs/authentication_bloc_provider.dart'; import 'paquete:diario/blocs/home_bloc.dart';
import 'paquete:journal/blocs/home_bloc_provider.dart';
import 'paquete:journal/blocs/journal_edit_bloc.dart'; import 'paquete:journal/
blocs/journal_edit_bloc_provider.dart'; import 'paquete:journal/classes/
format_dates.dart'; importar 'paquete: diario/clases/mood_icons.dart'; import
'paquete: diario/modelos/diario.dart'; importar 'paquete: diario/
páginas/edit_entry.dart'; import 'paquete:journal/services/
db_firestore.dart';
```

2. Modifique la clase _HomeState y agregue el _authenticationBloc privado, _homeBloc, _uid, variables _moodIcons y _formatDates . Anule initState() e inicialice la variable _login Bloc con la clase LoginBloc(AuthenticationService()) inyectando AuthenticationService() en el constructor.

```
bloque de autenticación _ bloque de autenticación;
BloqueInicio _bloqueInicio;
Cadena _uid;
MoodIcons _moodIcons = MoodIcons();
FormatDates _formatDates = FormatDates();
```

3. Anule el método didChangeDependencies() e inicialice las variables de las clases de proveedor. Tenga en cuenta que es necesario acceder a las clases InheritedWidget desde el método didChangeDependencies() y no desde el método initState() . Tenga en cuenta que el estado de la variable _uid se inicializa desde HomeBlocProvider y el valor se pasa a la clase Journal para agregar entradas de diario.

```
@anular
void hizoCambiarDependencias() {
    super.didChangeDependencies();
    _authenticationBloc = AuthenticationBlocProvider.of(contexto).
    bloque de autenticación;
    _homeBloc = HomeBlocProvider.of(contexto).homeBloc; _uid =
    HomeBlocProvider.of(contexto).uid; }
```

4. Anule el método dispose() y elimine la variable _homeBloc . La clase HomeBloc Se llama al método dispose() y se cierran todos los StreamControllers.

```
@anular
anular disponer()
{ _homeBloc.dispose();}
```

```
super.dispose();
}
```

5. Agregue el método `_addOrEditJournal()` que recibe los parámetros con nombre `add` y `journal`.
6. Dentro del método, agregue el método `Navigator.push()` que navega a la página `EditEntry()` como una propiedad secundaria de la clase `JournalEditBlocProvider`.
7. Para la propiedad `journalEditBloc`, inyecte la clase `JournalEditBloc` con la clase `add`, `journal` y `DbFirestoreService()`.
8. Agregue la propiedad `fullscreenDialog` y establezcala en verdadero.

Tenga en cuenta que ingresa valores al BLoC a través de sumideros, pero para este caso, tiene sentido evitar agregar StreamControllers y listeners innecesarios, ya que no requiere que el usuario ingrese estos datos. En su lugar, pasa las variables `add` y `journal` a través de `JournalEditBloc()` constructor.

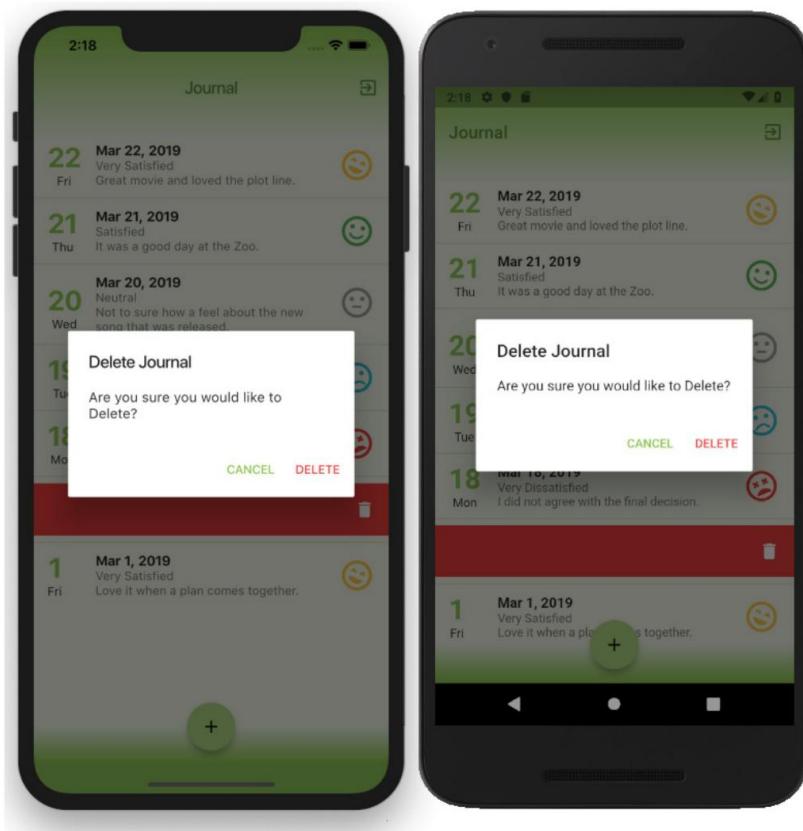
```
// Agregue o edite la entrada del diario y llame al cuadro de diálogo Mostrar entrada void
_addOrEditJournal({bool add, Journal journal}) { Navigator.push(
    contexto,
    MaterialPageRoute(builder:
        (Contexto BuildContext) => JournalEditBlocProvider(
            journalEditBloc: JournalEditBloc(add, journal, DbFirestoreService()), child: EditEntry(), ), fullscreenDialog: true
),
),
);
}
```

9. Agregue el método `_confirmDeleteJournal()` que devuelve el método `showDialog()`. El cuadro de diálogo muestra una advertencia de confirmación de eliminación que le da al usuario la opción de eliminar la entrada o cancelarla.

```
// Confirmar la eliminación de una entrada de diario
Future<bool> _confirmDeleteJournal() asíncrono {
    return await showDialog( context:
        context, barrierDismissible:
        false, builder: (BuildContext context)
    { return AlertDialog( title: Text("Eliminar diario"),
        content: Text("¿Está seguro
        de que desea eliminar?"), acciones :
        <Widget>[ Botón Plano(
            niño: Text("CANCELAR"),
            onPressed: () {
                Navegador.pop(contexto, falso); },
            ),
            botón plano (
                child: Text('DELETE', style: TextStyle(color: Colors.red),), onPressed: () {

```

```
        Navigator.pop(contexto, verdadero); }, ), ], ); }, );  
    }  
}
```



10. Modifique la propiedad de acciones de AppBar y reemplace el comentario TODO con la llamada a la autenticaciónBloc.logoutUser.add (verdadero) sumidero. El AuthenticationBloc recibe el valor del sumidero para cerrar la sesión del usuario, y la página principal (que contiene la administración del estado de toda la aplicación) lleva automáticamente al usuario a la página de inicio de sesión.

```
acciones: <Widget>[  
    BotónIcono(  
        icono: Icono (Icons.exit_to_app, color: Colors.lightGreen.shade800,), onPressed: () {  
            _authenticationBloc.logoutUser.add(verdadero); }, ), ],
```

11. Edite la propiedad del cuerpo y reemplace el widget Container() con el widget StreamBuilder , y establezca la propiedad de secuencia en la secuencia _homeBloc.listJournal.
12. Dentro de la propiedad del constructor , agregue una instrucción if para verificar ConnectionState.waiting y devolver un widget Center() con la propiedad secundaria establecida en el widget CircularProgressIndicator() .
13. Agregue una instrucción else if que verifique snapshot.hasData y devuelva la llamada al método _build ListViewSeparated(snapshot) pasando la variable de instantánea . Creará el método _buildListViewSeparated() en el paso 17.
14. Agregue una instrucción else y devuelva un widget Center() con la propiedad secundaria establecida en un widget Container() .
15. Agregue un widget de texto a la propiedad secundaria Container() con el mensaje 'Add Journals'.

```
cuerpo: StreamBuilder( stream:  
    _homeBloc.listJournal, builder: ((Contexto  
BuildContext, instantánea AsyncSnapshot) {  
    if (instantánea.connectionState == ConnectionState.esperando) {  
        centro de retorno  
        (hijo: CircularProgressIndicator(),);  
  
    } else if (instantánea.hasData) { return  
        _buildListViewSeparated(instantánea); } else { return  
  
        Center( child:  
            Container( child:  
                Text('Add Journals.'), ), );  
  
    } }, ),
```

16. Modifique la propiedad FloatingActionButton onPressed y reemplace el comentario TODO con la llamada al método _addOrEditJournal(add: true, journal: Journal(uid: _uid)) .
botón de acción flotante: botón de acción flotante (/...
onPressed: () async
 { _addOrEditJournal(add: true, journal: Journal(uid: _uid)); },)
17. Después del método Widget build(BuildContext context) , agregue el _buildListViewSeparated(instantánea AsyncSnapshot) .
18. Regrese y llame al constructor ListView.separated() con la propiedad itemCount establecida en snapshot.data.length, lo que significa el recuento de elementos del diario.
19. Para la propiedad itemBuilder , agregue la variable _titleData y establezca el valor llamando al _formatDates.dateFormatShortMonthDayYear pasando el valor de la fecha de la instantánea .

20. Agregue la variable `_subtitle` y establezca el valor concatenando el estado de ánimo de la instantánea y valores de nota .

21. Agregue la propiedad `separatorBuilder` y devuelva un widget `Divider()` con el conjunto de propiedades de color a `Colores.gris`.

```
Widget _buildListViewSeparated(AsyncSnapshot snapshot) { return
    ListView.separated( itemCount:
        snapshot.data.length, itemBuilder: (BuildContext
        context, int index) { String _titleDate =
            _formatDates.dateFormatShortMonthDayYear(snapshot .data[index].date); String _subtitle =
        instantánea.datos[índice].estado
            de ánimo + "\n" + instantánea .datos[índice].nota;
            volver Desechable(); },
        separatorBuilder: (contexto BuildContext, int index) { return Divider( color:
            Colors.grey, ); }, );
}
```

22. Agregue al widget `Dismissible()` la propiedad clave establecida en el ID de documento de la instantánea mediante la clase `Key()` . La clase `Key()` crea un identificador único para un widget para asegurarse de que se elimine la entrada de diario correcta.

```
volver descartable(
    clave: Clave(instantánea.datos[índice].documentID),
```

23. Establezca la propiedad de fondo en un widget `Container()` con el ícono establecido en `Icons.delete`. Siga los mismos pasos para la propiedad `secondBackground` . Cuando el usuario desliza el dedo sobre una fila para eliminar la entrada, se muestra el fondo rojo con los iconos de eliminación que notifican al usuario que está a punto de eliminar una entrada del diario.

```
fondo: Contenedor (color:
    Colors.red, alineación:
    Alignment.centerLeft, padding: EdgeInsets.only
    (left: 16.0), child: Icon (Icons.delete, color: Colors.white, ), ),
secundarioBackground:
    Contenedor (color : Colors.red,
```

```
alineación: Alignment.centerRight, padding:
    EdgeInsets.only(right: 16.0),
    child: Icon( Icons.delete, color: Colors.white, ), ),
```

24. Establezca la propiedad secundaria en el widget ListTile() con la propiedad principal establecida en el widget Column(). Agregue a la propiedad de los niños dos widgets de texto que llamen a la clase _formatDates . La propiedad principal muestra el día y el día de la semana del asiento de diario.

```
hijo: ListTile(  
    inicial: Columna (hijos:  
  
        <Widget>[ Text(_formatDates.dateFormatDayNumber(snapshot.data[index].date),  
            estilo: TextStyle  
                (fontWeight: FontWeight.bold, fontSize:  
                    32.0, color:  
                        Colors.lightGreen),  
        ),  
        Texto(_formatDates.dateFormatShortDayName(instantánea.datos[índice].fecha)), ], ),
```

25. Establezca la propiedad final en el widget Transform() con el método Matrix4 giratorioZ() .
26. Pase la rotación del ícono de estado de ánimo al método giratorioZ() llamando al método _moodIcons.getMoodRotation de la clase MoodIcons . Para el estado de ánimo más feliz, el ícono se gira hacia la izquierda y para el estado de ánimo más triste, el ícono se gira hacia la derecha.

```
final: Transformar (transformar:  
    Matrix4.identity().rotateZ(_moodIcons .getMoodRotation(snapshot.data[index].mood)),  
  
    alineación: Alignment.center, child:  
        Icon(_moodIcons.getMoodIcon(snapshot.data[index].mood), color:  
            _moodIcons.getMoodColor(instantánea.datos[índice].estado de ánimo), tamaño: 42.0,),
```

27. Establezca la propiedad de título en el widget de texto que muestra la variable _titleDate .
28. Establezca la propiedad de subtítulos en el widget de texto que muestra la variable _subtítulo .
29. Para la propiedad onTap , llame al método _addOrEditJournal y pase un valor falso para la propiedad add y el valor snapshot.data[index] para la propiedad journal . La propiedad onTap maneja al usuario tocando una entrada de diario y llama al método _addOrEditJournal() para editar la entrada.

```
título:  
    Texto( _titleDate,  
        estilo: TextStyle(fontWeight: FontWeight.bold), ), subtítulo: Texto(_subtitle),  
  
    onTap: () { _addOrEditJournal( add:  
        false, journal:  
            snapshot.data[index], ); }, ),
```

30. Para la propiedad confirmDismiss , agregue una llamada al método _confirmDeleteJournal() que muestra un cuadro de diálogo para confirmar la eliminación de la entrada del diario.

31. Agregue la declaración if para verificar que el valor de la variable confirmDelete sea verdadero y llame al _homeBloc.deleteJournal.add(snapshot.data[index]) sumidero para indicar a la clase HomeBloc que elimine la entrada del diario.

```
confirmDismiss: (dirección) async { bool  
  confirmDelete = await _confirmDeleteJournal(); if  
  (confirmDelete)  
    { _homeBloc.deleteJournal.add(snapshot.data[index]); } }, );
```

CÓMO FUNCIONA

Editó el archivo home.dart para que contuviera la clase Home que amplía un StatefulWidget para manejar la visualización de una lista de entradas de diario con la capacidad de agregar, modificar y eliminar registros individuales. Accedió a las clases AuthenticationBlocProvider y HomeBlocProvider utilizando el método of() del proveedor desde el método didChangeDependencies para recibir el estado de la página principal. Implementó las clases AuthenticationBloc y HomeBloc usando el widget StreamBuilder para monitorear los cambios de entrada y autenticación. Implementó el widget StreamBuilder para monitorear el flujo _homeBloc.listJournal, y el generador se reconstruye cada vez que cambia una entrada de diario.

El método _addOrEditJournal() maneja agregar o modificar una entrada de diario. El constructor toma los parámetros add y journal nombrados para ayudar si está agregando o modificando una entrada. Para mostrar la página de entrada de edición, llame al método Navigator.push() pasando JournalEditBlocProvider e inyectando los parámetros recibidos en la clase JournalEditBloc .

El método _buildListViewSeparated(snapshot) utiliza el constructor ListView.separated() para crear la lista de entradas de diario. El itemBuilder devuelve un widget Dismissible() que maneja la eliminación de entradas de diario deslizando el dedo hacia la izquierda o hacia la derecha en la entrada. La propiedad secundaria Dismissible() usa List Tile() para dar formato a cada entrada de diario en ListView. SeparatorBuilder devuelve el widget Divider() para mostrar una línea divisoria gris entre las entradas del diario.

AÑADIR LA PÁGINA DEL DIARIO DE EDICIÓN

La página de edición de diario es responsable de agregar y editar una entrada de diario. En esta sección, aprenderá a crear la página de edición de diario que implementa JournalEditBlocProvider como proveedor de la clase JournalEditBloc . Aprenderá a usar el widget StreamBuilder con los botones de fecha, estado de ánimo, nota y Cancelar y Guardar. Implementará la función showTimePicker() para presentar al usuario un calendario para elegir una fecha. Aprenderá a utilizar el widget DropdownButton() para presentar al usuario una lista de estados de ánimo seleccionables con el ícono, el color, la descripción y la rotación del ícono de estado de ánimo. Utilizará el método Matrix4 giratorioZ() para implementar la rotación del ícono de estado de ánimo. Utilizará el constructor TextEditingController() con el widget de nota TextField() . Aprenderá a llamar a la clase MoodIcons para colorear y girar iconos en la lista de selección DropdownButton DropdownMenuItem . Aprenderá cómo llamar a la clase FormatDates para dar formato a la fecha seleccionada. Consulte la Figura 16.4 para conocer el aspecto de la página de edición final del diario.

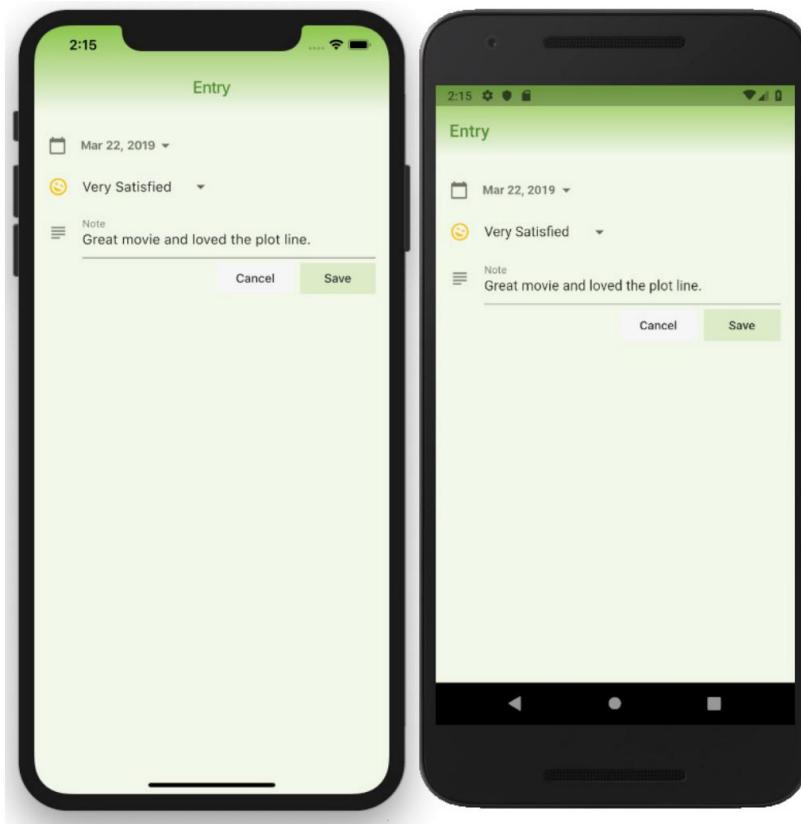


FIGURA 16.4: Página de edición final del diario

PRUÉBALO Creación de la página Editar diario

En esta sección, continuará editando el proyecto del diario . Agregará la clase JournalEditBloc para crear o modificar y guardar una entrada existente.

1. Cree un nuevo archivo Dart en la carpeta de páginas . Haga clic derecho en la carpeta de páginas , seleccione Nuevo Archivo Dart, ingrese edit_entry.dart y haga clic en el botón Aceptar para guardar.
2. Importe las clases material.dart, journal_edit_bloc.dart, journal_edit_bloc_provider.dart, format_dates.dart y mood_icons.dart . Agregue una nueva línea y cree la clase EditEntry que extiende un StatefulWidget.

```
importar 'paquete: flutter/material.dart'; import
'paquete:journal/blocs/journal_edit_bloc.dart'; import 'paquete:journal/
blocs/journal_edit_bloc_provider.dart'; import 'paquete:journal/classes/
format_dates.dart'; importar 'paquete: diario/clases/
mood_icons.dart';
```

```
clase EditEntry extiende StatefulWidget { @override

    _EditEntryState createState() => _EditEntryState(); }

clase _EditEntryState extiende Estado<EditEntry> {
    @anular
    Compilación del widget (contexto BuildContext) {
        devolver Contenedor();

    }
}
```

3. Modifique la clase `_EditEntryState` y agregue las variables privadas `JournalEditBloc _journalEditBloc`, `FormatDates _formatDates`, `MoodIcons _moodIcons` y `TextEditingController _noteController`.

El campo de nota usa el widget `TextField` , que requiere `_noteController` para acceder y modificar los valores.

```
clase _EditEntryState extiende Estado<EditEntry> {
    JournalEditBloc _journalEditBloc;
    FormatDates _formatDates;
    Icons de estado de ánimo _iconos de estado de ánimo;
    Controlador de edición de texto _noteController;

    @anular
    Compilación del widget (contexto BuildContext) {
        devolver Contenedor();
    }
}
```

4. Anule `initState()` e inicialicemos las variables `_formatDates`, `_moodIcons` y `_noteController`. Asegúrese de agregar `super.initState ()`.

```
@anular
void initState() {
    super.initState();
    _formatoFechas = FormatoFechas();
    _moodIcons = MoodIcons();
    _noteController = TextEditingController(); _noteController.text =
    "";
}
```

5. Anule el método `didChangeDependencies()` e inicialice la variable `_journalEditBloc` de la clase `JournalEditBlocProvider` . Tenga en cuenta que es necesario acceder a las clases `InheritedWidget` desde el método `didChangeDependencies()` y no desde el método `initState()` .

```
@anular
void hizoCambiarDependencias() {
    super.didChangeDependencies();
    _journalEditBloc = JournalEditBloc.of(context).journalEditBloc;
}
```

6. Anule el método dispose() y elimine las variables _noteController y _journalEditBloc . Se llama al método dispose() de la clase JournalEditBloc y se cierran todos los Controladores de flujo.

```
@anular  
disponer ()  
{ _noteController.dispose ();  
_journalEditBloc.dispose ();  
super.dispose ();  
}
```

7. Agregue el método asíncrono _selectDate(DateTime selectedDate) que devuelve un Future<FechaHora>. Este método es responsable de llamar al showDatePicker() incorporado de Flutter que presenta al usuario un cuadro de diálogo emergente que muestra un calendario de Material Design para elegir fechas.
8. Agregue la variable DateTime _initialDate e inicialícela con la variable selectedDate pasada en el constructor.
9. Agregue una variable final DateTime _pickedDate (fecha que el usuario elige del calendario) e inicialícela llamando al constructor await showDatePicker() . Pase las propiedades context, initialDate, firstDate y lastDate . Tenga en cuenta que para la primera fecha, usa la fecha de hoy y resta 365 días, y para la última fecha, agrega 365 días, lo que le indica al calendario los intervalos de fechas seleccionables.
10. Agregue una declaración if que verifique que la variable _pickedDate (la fecha que el usuario seleccionó del calendario) no es igual a nulo, lo que significa que el usuario tocó el botón Cancelar del calendario. Si el usuario eligió una fecha, modifique la variable de fecha seleccionada utilizando el constructor DateTime() y pase el año, mes y día de _pickedDate.

11. Para la hora, pase _initialDate hora, minuto, segundo, milisegundo y microsegundo.

Tenga en cuenta que dado que está cambiando solo la fecha y no la hora, usa la fecha y hora de creación del original.

12. Agregue una declaración de devolución para devolver la fecha seleccionada.

```
// Selector de  
fecha Future<String> _selectDate(String selectedDate) async { DateTime  
_initialDate = DateTime.parse(selectedDate);  
  
fecha y hora final _pickedDate = esperar showDatePicker (  
context: context,  
initialDate: _initialDate, firstDate:  
DateTime.now().subtract(Duration(days: 365)), lastDate:  
DateTime.now().add(Duration(days: 365)), ); if (_fechaelegida!=  
nulo) { fechaseleccionada =  
fechahora( _fechaeleccionada.año,  
_fechaeleccionada.mes,  
_fechaeleccionada.día,  
_fechainicial.hora,  
_fechainicial.minuto,
```

```
        _fechainicial.segundo,  
        _fechainicial.milisegundo,  
        _fechainicial.microsegundo).toString();  
  
    } return fechaseleccionada;  
}
```



13. Agregue el método `_addOrUpdateJournal()` que llama al `_journalEditBloc.saveJournal`

Sumidero modificado.add('Save') para guardar la entrada del diario. JournalEditBloc recibe esta solicitud y llama a la API de la base de datos de Cloud Firestore .

```
void _addOrUpdateJournal()  
{ _journalEditBloc.saveJournalChanged.add('Save'); Navigator.pop(context);  
}
```

14. En el método Widget build() , reemplace Container() con los widgets UI Scaffold y AppBar, y para la propiedad del cuerpo agregue SafeArea() y SingleChildScrollView() con la propiedad secundaria como Column().

15. Configure la propiedad Column crossAxisAlignment para comenzar.

16. Modifique el widget de texto de la propiedad de título de AppBar a Entrada y establezca el color de TextStyle propiedad a Colors.lightGreen.shade800.

17. Establezca la propiedad automaticImplyLeading en un valor falso para eliminar el icono de navegación predeterminado para volver a la página anterior. Desea que el usuario solo descarte esta página tocando el botón Cancelar o Guardar.

18. Para personalizar el color de fondo de la barra de aplicaciones con un degradado, elimine la sombra del widget de la barra de aplicaciones configurando la propiedad de elevación en 0.0.

19. Para aumentar la altura de AppBar , establezca la propiedad inferior en un widget PreferredSize con la propiedad secundaria como un widget Contenedor y la propiedad PreferredSize en Size.from Height(32.0).

20. Establezca la propiedad `flexibleSpace` en un widget Container con la propiedad `decoration` en un widget `BoxDecoration`.
21. Establezca la propiedad de gradiente BoxDecoration en un LinearGradient con la propiedad de colores establecida en una lista de [Colors.lightGreen, Colors.lightGreen.shade50].
22. Establezca la propiedad de inicio en Alignment.topCenter y la propiedad final en Alignment.bottom Center. El efecto LinearGradient dibuja el color AppBar de un color verde claro y se desvanece gradualmente a un color verde claro.shade50 .

```
@anular
Widget compilación (contexto BuildContext) { return
Scaffold (appBar:
    AppBar (
        título: Texto('Entrada', estilo: EstiloTexto(color: Colores.verde claro .sombra800),),
```

```
automáticamente implicarLeading: falso, elevación: 0.0, espacio
flexible: Contenedor(
```

```
decoration: BoxDecoration(degradado:
    LinearGradient(
        colors: [Colors.lightGreen, Colors.lightGreen.shade50], comenzar:
        Alignment.topCenter, end:
        Alignment.bottomCenter, ), ), ), ), cuerpo:
```

```
SafeArea(
    mínimo: EdgeInsets.all(16.0), hijo:
    SingleChildScrollView(
        niño: Columna
            (crossAxisAlignment: CrossAxisAlignment.start, niños: <Widget>[
```

```
], ), ), ), );
}
```



CÓMO FUNCIONA

Creó el archivo edit_entry.dart que contiene la clase EditEntry que amplía un StatefulWidget para agregar o editar una entrada de diario. Implementó JournalEditBlocProvider para recibir el estado de la página de inicio mediante el uso del método of() del proveedor del método didChangeDependencies . La clase JournalEditBloc es responsable de agregar, editar y guardar entradas de diario. Usó la inyección de dependencia pasando add, journal y el servicio DbFirestoreService() a la clase JournalEditBloc . Implementó showDatePicker() para seleccionar fechas con un calendario e implementó el widget DropdownButton() para seleccionar una lista de estados de ánimo y usó el método Matrix4 giratorioZ() para rotar íconos de estados de ánimo. Usó la clase de utilidad MoodIcons con la lista de selección DropdownButton Drop downMenuItem . La clase de utilidad FormatDates da formato a la fecha seleccionada. Para darle a AppBar un efecto de degradado de color, usó el widget BoxDecoration para personalizar la propiedad de degradado con la clase LinearGradient .

El siguiente ejercicio continúa editando la página de entrada para agregar widgets de StreamBuilder individuales para manejar cada campo de entrada y acciones.

PRUÉBELO Adición de StreamBuilders a la página Editar diario

En esta sección, continuará editando el archivo edit_entry.dart . Agregará los widgets de StreamBuilder para controlar la fecha, el estado de ánimo, la nota y los botones Cancelar y Guardar.

1. Continúe editando la Columna creada en el paso 13 del ejercicio anterior. Agregue a los elementos secundarios Column el widget StreamBuilder() con la propiedad de flujo establecida en _journalEditBloc . flujo de edición de fecha. Este widget StreamBuilder() es responsable de manejar el campo de fecha.
2. Agregue a la propiedad del constructor la instrucción if para comprobar si la instantánea no tiene datos mediante la expresión !snapshot.hasData . Dentro de la declaración if , devuelva un Container() , lo que significa un espacio vacío ya que los datos aún no han llegado.

```
if (!snapshot.hasData) { return  
    Container();  
}
```

3. Agregue una nueva línea y devuelva el widget FlatButton que se usa para mostrar la fecha seleccionada con formato, y cuando el usuario toca el botón, presenta el calendario.
4. Establezca la propiedad de relleno FlatButton en EdgeInsets.all(0.0) para eliminar el relleno para una mejor estética y agregue a la propiedad secundaria un widget Row() .



5. Agregue a la propiedad Row children el ícono Icons.calendar_day con un tamaño de 22.0 y color propiedad a Colors.black54.
6. Agregue un SizedBox con la propiedad de ancho establecida en 16.0 para agregar un espaciador.
7. Agregue un widget de texto y formatee _selectedDate con el constructor DateFormat.yMMEd(). Agregue el ícono Icons.arrow_drop_down con la propiedad de color establecida en Colors.black54.
8. Agregue la devolución de llamada onPressed() y márquela asíncrona ya que llamar al calendario es un evento futuro .
9. Agregue a onPressed() la llamada al método FocusScope.of().requestFocus() para descartar el teclado si alguno de los widgets de TextField tiene foco. Este paso es opcional, pero quería mostrarte cómo se logra.

```
FocusScope.of(context).requestFocus(FocusNode());
```

10. Agregue una variable DateTime _pickerDate inicializada llamando al método futuro await _selectDate(_select edDate) , razón por la cual agrega la palabra clave await . Agregó este método en el paso 7 en el ejercicio "Crear la página Editar diario".
11. Agregue la llamada al receptor _journalEditBloc.dateEditChanged.add(_pickerDate) que pasa a JournalEditBloc la variable _pickerDate seleccionada .

```
String _pickerDate = esperar _selectDate(snapshot.data);
_journalEditBloc.dateEditChanged.add(_pickerDate);
```

El siguiente es el código completo del widget FlatButton :

```
StreamBuilder( stream:
    _journalEditBloc.dateEdit, builder: (BuildContext
    context, AsyncSnapshot snapshot) { if (!snapshot.hasData) { return Container();

} return FlatButton( relleno:
    EdgeInsets.all(0.0), child: Row( children:

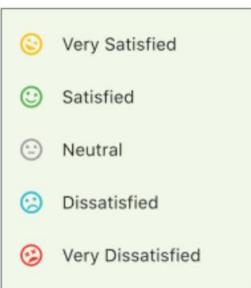
<Widget>[ Icon( Icons.calendar_today,
size:
    22.0, color: Colors.black54,

),
SizedBox (ancho: 16.0), Texto
    _formatDates.dateFormatShortMonthDayYear (snapshot.data), estilo: TextStyle (color:
    Colors.black54,
    fontWeight: FontWeight.bold),

),
Icono
    (Icons.arrow_drop_down, color:
    Colors.black54,
),
],
),
);}
```

```
onPressed: () asíncrono {
    FocusScope.of(contexto).requestFocus(FocusNode()); String
    _pickerDate = esperar _selectDate(snapshot.data);
    _journalEditBloc.dateEditChanged.add(_pickerDate); }, ), ),
```

12. Agregue una nueva línea y agregue el widget `StreamBuilder()` con la propiedad de flujo establecida en `_journalEditBloc.moodEdit stream`. Este widget `StreamBuilder()` es responsable de manejar el campo de estado de ánimo.



`StreamBuilder(flujo:`
 `_journalEditBloc.moodEdit,`

13. Agregue a la propiedad del constructor la instrucción `if` para comprobar si la instantánea no tiene datos mediante la expresión `!snapshot.hasData`. Dentro de la instrucción `if` devuelve un `Container()`.

```
constructor: (contexto BuildContext, instantánea AsyncSnapshot) { if (!
    snapshot.hasData) { return
        Container();
    }
```

14. Agregue una nueva línea y devuelva el widget `DropdownButtonHideUnderline` con la propiedad secundaria establecida en el widget `DropdownButton<MoodIcons>` declarando el tipo como la clase `MoodIcons`.

`return DropdownButtonHideUnderline(child:
 DropdownButton<MoodIcons>(`

15. Agregue la propiedad de valor y llame al método `_moodIcons.getMoodIconsList()` utilizando el método `indexWhere` para buscar por el valor de `icon.title`. El valor devuelto es la posición de índice del ícono de estado de ánimo en la lista.

`valor: _moodIcons.getMoodIconsList()`

```
[ _moodIcons .getMoodIconsList() .indexWhere((icono) => icon.title
== snapshot.data) ],
```

16. Agregue la propiedad `onChanged` y llame al sumidero `_journalEditBloc.moodEditChanged.add(selected. title)` pasando el valor `selected.title`. Este evento ocurre cuando el usuario selecciona un estado de ánimo del widget `DropdownButton`.

- ```
onChanged: (seleccionado) {
 _journalEditBloc.moodEditChanged.add(seleccionado.título); }, 17. Esta
sección crea la lista emergente para elegir un estado de ánimo. Agregue la propiedad items llamando a la
_moodIcons.getMoodIconsList(), y usando el operador de punto, llame al método map() que recibe el ícono de
estado de ánimo seleccionado .
```
18. Agregue una nueva línea, devuelva el widget DropdownMenuItem<MoodIcons> y establezca la propiedad de valor en la variable seleccionada .
19. Para la propiedad secundaria , agregue una Fila con la propiedad secundaria establecida en el widget Transform() , SizedBox y el widget de texto .
20. Siga los pasos desde el paso 22 de la página de inicio para configurar el método de rotación de Matrix4() .
21. Al final de este método, asegúrese de agregar la llamada a toList() usando el operador punto. elementos:

```
_moodIcons.getMoodIconsList().map((MoodIcons seleccionados) {
 return DropdownMenuItem<MoodIcons>(valor:
 seleccionado, hijo:
 Fila(hijos:
 <Widget>[Transform(transform:
 Matrix4.identity()..rotateZ(
 _moodIcons.getMoodRotation(selected.title)), alineación:
 Alignment.center, child:
 Icon(_moodIcons.getMoodIcon(selected.title), color:
 _moodIcons.getMoodColor(selected.title)),
),
 SizedBox(ancho: 16.0,),

 Texto(título.seleccionado)],),); }).Listar(),),);};

},
```

El siguiente es el código completo del widget DropdownButton :

```
StreamBuilder(flujo:
 _journalEditBloc.moodEdit, constructor:
 (contexto BuildContext, instantánea AsyncSnapshot) { if (!snapshot.hasData)
 { return Container();

 } return DropdownButtonHideUnderline(child:
 DropdownButton<MoodIcons>(
 valor: _moodIcons.getMoodIconsList()

 [_moodIcons .getMoodIconsList()
```

```
.indexWhere((icono) => icon.title == snapshot.data)], onChanged:

(seleccionado) {
 _journalEditBloc.moodEditChanged.add(seleccionado.título); }, artículos:

_moodIcons.getMoodIconsList().map((MoodIcons seleccionados) {
 return DropdownMenuItem<MoodIcons>(
 valor:
 seleccionado, hijo:
 Fila(hijos:
 <Widget>[Transform(transform:

 Matrix4.identity().rotateZ(
 _moodIcons.getMoodRotation(selected.title)), alineación:
 Alignment.center, child:

 Icon(_moodIcons.getMoodIcon(selected.title), color:
 _moodIcons.getMoodColor(selected.title)),
),
 SizedBox(ancho: 16.0,),

 Texto(título.seleccionado)],),); }).Listar(),),);
},
});
```

22. Agregue una nueva línea y agregue el widget StreamBuilder() con la propiedad de flujo establecida en  
\_journalEditBloc.noteEditar flujo. Este widget StreamBuilder() es responsable de manejar el campo de nota.

23. Agregue a la propiedad del constructor la instrucción if para comprobar si la instantánea no tiene datos mediante la  
expresión !snapshot.hasData .

24. Dentro de la instrucción if , devuelve un Container(). if (!

```
snapshot.hasData) { return
 Container();
}
```

25. Cuando usa un TextField y el StreamBuilder mientras el usuario escribe su nota, el cursor sigue saltando al principio del  
TextField. Para solucionar esto, agregue \_noteController.value para que sea igual al método  
\_noteController.value.copyWith(text: snapshot.data) . El cursor que rebota ocurre porque cada vez que se escribe un  
carácter, el sumidero actualiza la secuencia y StreamBuilder recupera el nuevo valor y vuelve a llenar el  
TextField.

26. Agregue una nueva línea, devuelva el widget TextField y establezca la propiedad del controlador en \_note  
Controlador.

27. Agregue la propiedad maxLines y configúrela en el valor nulo , haciendo que TextField se expanda automáticamente a  
múltiples líneas según sea necesario, lo que lo convierte en una excelente característica de UX.

28. Agregue la propiedad onChanged y llame a \_journalEditBloc.noteEditChanged.add(note) fregadero pasando el valor de la nota . Este evento ocurre cuando el usuario escribe en el widget TextField .

```
StreamBuilder(flujo: _journalEditBloc.noteEdit,
constructor: (contexto BuildContext, instantánea AsyncSnapshot)
{ if (!snapshot.hasData)
 { return Container();
}
// Use copyWith para asegurarse de que cuando edite TextField, el cursor no salte al
primer carácter _noteController.value =
_noteController.value.copyWith(text: snapshot.data); return

TextField(controlador:
 _noteController,textInputAction:
 TextInputAction.newline, textCapitalization:
 TextCapitalization.sentences, decoration:
 InputDecoration(labelText: 'Note',
 icon:
 Icon(Icons.subject),),
maxLines: null,
onChanged: (nota) => _journalEditBloc.noteEditChanged.add(nota),);),
```

29. Agregue una nueva línea, agregue un widget Row() y establezca la propiedad mainAxisAlignment en MainAxisSize.end. Esta fila () es responsable de alinear a la derecha los botones Cancelar y Guardar en el formulario.

30. Agregue a la propiedad de los niños el widget FlatButton y establezca el niño en el widget Texto con el valor 'Cancelar' . Para la propiedad onPressed , llama al método Navigator.pop(context) para cerrar la página sin guardar.

31. Agregue un SizedBox (ancho: 8.0) para separar el siguiente widget FlatButton .

32. Agregue un segundo widget FlatButton y configure el niño en el widget Texto con el valor 'Guardar' .

33. Para la propiedad onPressed , llame al método \_addOrUpdateJournal() para guardar la entrada del diario.

```
Row(mainAxisAlignment: MainAxisSize.end,
children:
<Widget>[FlatButton(child:
Text('Cancel'), color:
Colors.grey.shade100,
onPressed: () {
{ Navigator.pop(context); },),
SizedBox (ancho: 8.0), FlatButton (hijo: Text ('Guardar'), color: Colors.lightGreen.shade100,
```

```
onPressed: ()

{ _addOrUpdateJournal(); },),],).
```

## CÓMO FUNCIONA

Creó el archivo `edit_entry.dart` que contiene la clase `EditEntry` que amplía un `StatefulWidget` para manejar la adición y edición de entradas de diario. `JournalEditBlocProvider` se utiliza para recibir el estado de la página de inicio . Implementó la clase `JournalEditBloc` sin usar un proveedor para agregar o editar y guardar entradas de diario. El widget `StreamBuilder` supervisa la secuencia `_journalEditBloc.dateEdit`, la secuencia `_journalEditBloc.moodEdit`, la secuencia `_journalEditBloc.noteEdit` y cada constructor se reconstruye cada vez que cambia un valor.

La función `showTimePicker()` presenta al usuario un calendario y el widget `DropdownButton()` presenta al usuario una lista de estados de ánimo seleccionables. El método `Matrix4 rotateZ()` se utiliza para implementar la rotación de iconos de estado de ánimo. El constructor `TextEditingController()` maneja el widget de nota `TextField()` . La clase `MoodIcons` es responsable de colorear y rotar iconos en la lista de selección `DropdownButton` `DropdownMenuItem` . La clase `FormatDates` da formato a la fecha seleccionada. Creó un efecto de degradado de color personalizado para el widget `AppBar` usando la clase `LinearGradient` .

---

## RESUMEN

En este capítulo, completó la aplicación de diario que comenzó en el Capítulo 14. Aplicó el patrón BLoC para separar los widgets de la interfaz de usuario de la lógica comercial. Implementó clases de BLoC, proveedores de BLoC, clases de servicio, clases de utilidad, clases de modelo y administración de estado local y de toda la aplicación.

Pasó la administración de estado de toda la aplicación entre páginas y la administración de estado local en el árbol de widgets mediante el uso de proveedores (`InheritedWidget`) y BLoC. Usó la inyección de dependencia para inyectar clases de servicio en las clases de BLoC. Aprendió que el beneficio de usar la inyección de dependencia mantiene las clases de BLoC independientes de la plataforma, lo que brinda la capacidad de compartir clases de BLoC entre diferentes plataformas como Flutter, AngularDart u otras.

Aplicó la administración de estado de toda la aplicación mediante la implementación de las clases `AuthenticationBlocProvider` y `AuthenticationBloc` en la página principal de la aplicación. Usó el widget `StreamBuilder` para monitorear la secuencia `_authenticationBloc.user` en busca de cambios de credenciales de usuario. Cuando la credencial del usuario cambia, el generador de `StreamBuilder` vuelve a compilar y lleva al usuario de forma adecuada a la página de inicio de sesión o de inicio.

Aplicó la clase `LoginBloc` para validar las credenciales del usuario y los requisitos de correo electrónico y contraseña. Reemplazaste el método `initState()` para inicializar la variable `_loginBloc` con la clase `LoginBloc` al inyectar la clase `AuthenticationService()` sin usar un proveedor.

Tenga en cuenta que la razón por la que inicializó la variable `_loginBloc` desde `initState()` y no desde `didChangeDependencies()` es porque `LoginBloc` no necesita un proveedor (`InheritedWidget`).

Usó el widget StreamBuilder con el widget TextField para validar los valores de correo electrónico y contraseña. La propiedad onChanged del widget TextField llama al sumidero \_loginBloc.emailChanged.add y al sumidero \_loginBloc.passwordChanged.add para enviar los valores a la clase LoginBloc Valida tors . Usó el widget StreamBuilder para escuchar la secuencia \_loginBloc.loginOrCreateButton para alternar Iniciar sesión o Crear cuenta como botón predeterminado.

Aplicó la clase HomeBloc para crear una lista de entradas de diario filtradas por el uid del usuario con la capacidad de agregar, modificar y eliminar entradas. Accedió a las clases AuthenticationBlocProvider y HomeBlocProvider utilizando el método of() del proveedor desde el método didChangeDependencies . Usó el widget StreamBuilder llamando al constructor ListView.separated para crear la lista de entradas de diario. Usó la propiedad confirmDismiss del widget Descartable para mostrar al usuario el cuadro de diálogo de confirmación de eliminación. Llamó a la clase de utilidad MoodIcons para dar formato a los iconos de estado de ánimo ya la clase FormatDates para dar formato a las fechas.

Accedió a la clase JournalEditBlocProvider usando el método of() del proveedor desde el método didChangeDependencies . Aplicó la clase JournalEditBloc para agregar, editar y guardar entradas de diario. Usó el widget StreamBuilder para manejar la fecha, el estado de ánimo, la nota y los botones Cancelar y Guardar. Implementó la función showTimePicker() para mostrar un calendario y usar el widget DropdownButton() para presentar al usuario una lista de estados de ánimo seleccionables. Usó el método Matrix4 giratorioZ() para rotar el ícono de estado de ánimo según el estado de ánimo. Usó la clase MoodIcons con DropdownButton DropdownMenuItem para presentar la lista de selección de estado de ánimo. Usó la clase Format Dates para formatear la fecha seleccionada.

## LO QUE APRENDISTE EN ESTE CAPÍTULO

| TEMA                             | CONCEPTOS CLAVE                                                                                                                                                                                                                                                |
|----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Clase LoginBloc                  | Aprendió a implementar la clase LoginBloc sin un proveedor para manejar las credenciales de autenticación y validar los requisitos de correo electrónico y contraseña.                                                                                         |
| Clase de bloque de autenticación | Aprendió a implementar la clase AuthenticationBloc con la clase AuthenticationBlocProvider para manejar la administración de estado de toda la aplicación en la página principal.                                                                              |
| Clase HomeBloc                   | Aprendió a implementar la clase HomeBloc con la clase HomeBlocProvider para llenar el ListView con entradas de diario mediante el separador() constructor. Aprendió a usar la clase HomeBloc para agregar, modificar o eliminar entradas de diario existentes. |
| Clase JournalEditBloc            | Aprendió a implementar la clase JournalEditBloc sin un proveedor para agregar o editar y guardar entradas de diario.                                                                                                                                           |
| Clase de widget heredado         | Las clases AuthenticationBlocProvider, HomeBlocProvider y JournalEditBlocProvider se extienden desde la clase InheritedWidget , y aprendió a acceder a ellas desde el método didChangeDependencies() y no desde el método initState() .                        |
| Clase AuthenticationBlocProvider | Aprendió a implementar AuthenticationBlocProvider como proveedor de la clase AuthenticationBloc para la administración del estado de autenticación en toda la aplicación.                                                                                      |
| Clase HomeBlocProvider           | Aprendió a implementar la clase HomeBlocProvider como proveedor de la clase HomeBloc .                                                                                                                                                                         |
| Inyección de dependencia         | Aprendió a usar la inyección de dependencia inyectando clases de servicio a las clases BLoC. Aprendió que al inyectar servicios, las clases de BLoC siguen siendo independientes de la plataforma.                                                             |
| Widget de StreamBuilder          | Aprendió a implementar el widget StreamBuilder para monitorear una transmisión y, cuando se produce un cambio, el constructor vuelve a generar el widget con los datos más recientes.                                                                          |

| TEMA                                              | CONCEPTOS CLAVE                                                                                                                                                                                                                                                  |
|---------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Constructor ListView.separated y widget Divider() | Aprendió a implementar ListView. constructor separado para crear una lista de entradas de diario separadas por un widget Divider() .                                                                                                                             |
| Widget descartable y propiedad confirmDismiss     | Aprendió a implementar el widget Descartable para deslizar y eliminar una entrada de diario. Aprendió a implementar la propiedad confirmDismiss del widget Descartable para generar un cuadro de diálogo para confirmar la eliminación de una entrada de diario. |
| Widget de botón desplegable ()                    | Aprendió a implementar el widget DropdownButton() para presentar una lista de estados de ánimo con el título, el color y la rotación de iconos.                                                                                                                  |
| clase MoodIcons                                   | Aprendió a implementar la clase MoodIcons para recuperar las propiedades del estado de ánimo, como el título, el color, la rotación y el ícono.                                                                                                                  |
| Función showTimePicker()                          | Aprendió a implementar la función showTimePicker() para presentar un calendario para elegir una fecha.                                                                                                                                                           |
| Matrix4 método de rotación Z ()                   | Aprendió a implementar el método Matrix4 giratorioZ() para girar un ícono de acuerdo con el estado de ánimo seleccionado.                                                                                                                                        |
| Clase FormatDates                                 | Aprendió a implementar la clase FormatDates para formatear fechas.                                                                                                                                                                                               |

# ÍNDICE

## A

clase abstracta , 414–415, 424, 427, 450  
 accediendo a la función main() , 64  
 propiedad de acciones , 104

agregando animación a aplicaciones, 151–175  
 Widgets de AppBar , 105–107  
 autenticación en la aplicación de diario del cliente, 395–403  
 Clase AuthenticationBloc , 432–434  
 Clase AuthenticationBlocProvider , 435–436

diseño básico de la aplicación de diario del cliente, 403–406  
 patrón BLOC, 432–449  
 BLoC a las páginas de la aplicación cliente de Firestore, 453–487  
 Clases a la aplicación de diario del cliente, 406–408  
 Base de datos de Cloud Firestore para aplicación de diario, 391–395  
 Paquetes de Cloud Firestore para la aplicación de diario del cliente, 395–403  
 arrastrar y soltar, 275–277 editar  
 página del diario, 472–484  
 Aplicación de Firebase a diario, 375–410  
 Aplicación de Firestore a diario, 375–410  
 Clase HomeBloc , 441–443  
 Clase HomeBlocProvider , 443–444  
 Clase de modelo de diario , 422–424  
 Clase JournalEditBloc , 444–447  
 Clase JournalEditBlocProvider , 447–449  
     página de inicio de sesión, 454–460  
 Clase LoginBloc , 436–440 Clases de servicio, 424–430  
 administración de estado a la aplicación cliente de Firestore, 411–451  
 Clase de validadores , 430–432

método addOrEditJournal() , 338–339, 362–364, 368, 371, 467, 471, 472 método addOrUpdateJournal() , 476 adelantado (AOT), 44

Emulador de Android  
 configurar en Linux, 19 configurar en macOS, 15 configurar en Windows, 17  
 Canal de la plataforma de host de Android, implementando, 318–321  
 Android Studio sobre, 23 editor de configuración para, 20 instalación en Linux, 19 instalación en macOS, 14–15 instalación en Windows, 16–17 método animationBalloon() , animaciones escalonadas y, 172, 174

Widget de AnimatedBuilder , 176  
 Widget de AnimatedContainer sobre, 176 usando, 152–155  
 Widget AnimatedCrossFade sobre, 176 usando, 155–159 método animationOpacity() , 160–163  
 Widget de AnimatedOpacity sobre, 176 usando, 160–163 animación agregar a aplicaciones, 151–175 usando el widget AnimatedContainer , 152–155 usando el widget AnimatedCrossFade , 155–159 usando el widget AnimatedOpacity , 160–163 usando la clase AnimationController , 164–170 usando animación escalonada, 170–175

## Clase AnimationController –clases

- Clase AnimationController sobre,  
176 usando,  
164–170  
AOT (antes de tiempo), 44  
AppBar widgets, 77, 78, 104, 105–107 aplicación  
de  
interactividad, 267–305  
temas, 42
- aplicaciones  
agregando animación a, 151–175  
creando, 26–30  
creando navegación para, 177–219  
ejecutando, 30–32  
diseñando usando temas, 33–36  
operadores aritméticos, 49  
matrices, declarado en JSON, 328–329  
método askToOpen(), 53  
afirmar declaración, 58  
Clase AssetBundle , 130–131  
operadores de asignación, 50  
programación asíncrona, implementación, 64 capacidades  
de autenticación agregadas a la  
aplicación de diario del cliente, 395–403  
Base de fuego, 380–381, 410
- Clase AuthenticationBloc , 432–434, 465–472,  
486  
Clase AuthenticationBlocProvider , 435–436, 460–464, 486
- Clase AuthenticationService , 414–415,  
424–430
- B**
- Clases BLoC, 415–416, 450, 453–487  
inyección de dependencia BLoC, 450  
patrón BLoC  
sobre, 450  
agregar, 432–449  
implementar, 417–419  
carpeta de bloques , 421  
Variables booleanas  
sobre, 47  
declaradas en JSON, 328–329  
Widget BottomAppBar , 199–203, 219
- Widget de barra de navegación inferior , 193–199, 219  
Clase BoxDecoration , 135  
instrucción break , 54  
método build() , 6, 7, 8, 11 método  
buildBody() , 259–260 método  
buildCompleteTrip() , 299, 300  
Objetos BuildContext , 9, 86 método  
buildDraggable() , 275–276 método  
buildDragTarget() , 276 método  
buildJournalEntry() , 260, 261 método  
buildJournalFooterImages() , 260,  
263–265  
método buildJournalHeaderImage() , 261 método  
buildJournalTags() , 260, 263 método  
buildJournalWeather() , 260, 262 método  
buildListViewSeparate() , 369, 371 método  
buildRemoveTrip() , 299, 301  
Widget de barra de botones , 126–130  
Widget de botones , 104, 119–130  
métodobuttonCreateAccount() , 458  
métodobuttonLogin() , 458–460
- C**
- Widget de tarjeta , 222–223, 251  
notación en cascada, 51  
cláusula de caso , 55  
Widget Center() , 469  
comprobar la orientación, 143–149, 150  
propiedad secundaria , 108  
Widget de fichas , 256, 266  
Widget CircleAvatar , 78, 266  
CircularProgressIndicator, herencia de 374  
clases, mezclas de 60  
clases, 60–61 clases  
resumen, 414–415, 424, 427, 450 agregar  
a la aplicación de diario del cliente, 406–408  
Controlador de animación, 164–170, 176  
Paquete de activos, 130–131  
Bloque de autenticación, 432–434, 465–472, 486  
Proveedor de bloque de autenticación, 435–436, 460–  
464, 486  
Servicio de autenticación, 414–415, 424–430

BloC, 415–416, 450, 453–487  
BoxDecoration, base de datos 135, 330–331, 373  
Base de datos, 330–331, 339–344, 373  
Rutinas de archivo de base de datos, 330–331, 339–344, 373  
Duración, 155  
EditJournalBloc, 465  
FormatoFechas, 407–408, 465, 487  
Bloque Hogar, 441–443, 486  
Proveedor de HomeBloc, 443–444, 460–464, 465–472, 486  
Widget heredado, 6, 415–416, 418, 421, 451, 486  
  
EntradaDecoración, 135–136  
Revista, 330–331, 339–344, 373, 422–424  
DiarioEditar, 330–331, 339–344, 373  
JournalEditBloc, 444–447, 486  
JournalEditBlocProvider, 447–449, 472–484  
LoginBloc, 436–440, 454–455, 486  
MaterialPageRoute, 178, 180–181 modelo, 416, 450  
MoodIcons, 407–408, 465, 471, 478, 487 servicio, 417, 424–430, 450  
Estado, 30, 37  
Widget con estado, 6–8, 22, 37  
ApátridaWidget, 6, 22  
Corriente, 61–62, 419–420, 450  
Controlador de corriente, 419–420, 450  
Teletipo, 164  
usando, 57–61, 64  
Validadores, 430–432, 436, 440, 450 carpeta de clases , aplicación de canal de plataforma de cliente 421, implementación, 309–313  
Cloud Firestore  
sobre, 410  
agregar base de datos a la aplicación de diario, 391–395 agregar paquetes a la aplicación de diario del cliente, 395–403  
colecciones, 410  
documentos, 410  
reglas de seguridad, 410  
estructuración y modelado de datos, 377–379  
visualización de reglas de seguridad, nube \_ 381–383 paquete firestore , 410 código, comentarios, 44–45, 64 colecciones, Cloud Firestore, 410 anidamiento de columnas, 115–117  
Widget Column() , 36, 78, 104, 114–119, 254, 256, 257, 266, 471  
código de comentarios, 44–45, 64  
composición, 257  
expresiones condicionales, 51  
configuración  
Editor de Android Studio, 20  
Firebase, 383–391  
método confirmDeleteJournal() , 467, 471–472  
  
confirmDismiss propiedad, 487  
constantes, refactorización con, 86  
Widget de contenedor , 78, 104, 108–112, 469, 470  
declaración de continuación , 54–55 método copyWith , 36  
método createElement , 9–10 creación de  
aplicaciones, 26–30  
Proyectos de Firebase, 383–391  
carpetas/archivos, 65–68 árboles completos de widgets, 79–85, 100 aplicación de diario, 335–371 diseños, 253–266, 257–265 navegación para aplicaciones, 177–219 listas de desplazamiento/efectos, 221 –251 árboles de widgets poco profundos, 85–98, 100 gestión de estado, 421–449 método crossFade() , 155–156  
Widget de CupertinoApp , 69  
Widget de barra de navegación de Cupertino , 78  
Widget CupertinoPageScaffold , 78  
Widget CupertinoTabScaffold , 78 llaves, 328, 340, 346, 347, 407 personalización del widget  
CustomScrollView con astillas, 243–250, 251  
  
Widget CustomScrollView sobre, 221  
personalización con astillas, 243–250, 251

## D

dar

vueltas, 4, 22, 23  
 Variables booleanas, 47  
 sentencia break , 54  
 cláusula case , 55  
 herencia de clase, 60  
 mixins de clase, 60–61  
 código de comentario, 44–45  
 sentencia continue , 54–55  
 declaración de variables, 46–49  
 bucle do-while , 53–54  
 sentencia else , 51–52  
 instrucción if , 51–52  
 implementar programación asíncrona, 61–62 importar paquetes, 57 instalar complemento para, 20 listas, 47–48 bucles for , 52–53 mapas, 48 variables numéricas, 47 razones para usar, 43– 44 variables de referencia, 45–46 runas, 48–49 ejecución del punto de entrada principal() , 45 variables de cadena, 47 declaración de cambio , 55 operador ternario, 52 uso de clases, 57–61 uso de declaraciones de flujo, 51–55 uso de funciones, 55–57 usando operadores, 49–51 while loop, 53–54 dart:convert library, 374 dart:io library, 374 dart:math, 374 recuperación de datos con el widget

FutureBuilder , 333–334 guardar con persistencia local, 327–  
 374 modelado de datos, Cloud Firestore, 377–379  
 Clase de base de datos , 330–331, 339–344, 373  
 clases de base de datos

sobre, 373  
 JSON y, 330–331  
 Clase DatabaseFileRoutines , 330–331, 339–344, 373  
 DateFormat, 373 formato de fechas, 331–332 clasificación de listas de, 332–333 DateTime.parse() , 373 método deactivate() , 8 interfaz de usuario (UI) declarativa, 22 declaración de variables, 46–49 decoradores, 135–139, 150 método deleteTrip() , 299–300, 303 comprobación de dependencias en Linux, 19 comprobando en Windows, 16 comprobando en macOS, 14 inyección de dependencia, 415, 450, 486 método didChangeDependencies() , 8, 474 método didFinishLaunchingWithOptions , 316 método didUpdateWidget() , 8 Widget descartable , 296–303, 305, 360, 365, 371, 373, 465–472, 470, 487 método dispose() , 7, 350, 433, 438, 442, 456, 460, 475 Widget divisor , 78, 261, 266, 470, 487 comentarios de documentación, 44–45 documentos, Cloud Firestore, 410 bucle do-while , 53–54 arrastrar y soltar, agregar, 275–277 Widget arrastrable , 271, 275–277, 305 Widget de arrastrar objetivo , 271, 275–277, 305 Widget de cajón , 207–217, 219 widget DropDownButton() , 487 Clase de duración , 155

## Y

editar página del diario, agregar, 472–484  
 Clase EditJournalBloc , editor 465, Android Studio, 20  
 Objetos de elementos , árboles de 9 elementos

about, 8–9, 22  
widget con estado y, 10–12 widget sin estado y, 9–10 elementos, 5, 22 sentencia else , 51–52 Configuración del emulador (Android) en Linux, 19 configuración en macOS, 15 configuración en Windows, 17 operadores de igualdad, 49 Widget ampliado , 78 paquetes externos, 38–41, 42

## F

diseño final, 257 Acerca de Firebase, 376–377, 410 adición a la aplicación de diario, 375–410 autenticación, 410 configuración, 383–391 sitio web del panel de la consola, 391 capacidades de autenticación de visualización, 380–381 paquete firebase\_auth , 410 Firestore. Consulte también Cloud Firestore sobre, 376–377 agregar clases BLoC a las páginas de la aplicación del cliente, 453–487 agregar administración de estado a, 411–451 agregar a la aplicación de diario, 375–410 widget FlatButton , 121 propiedad flexibleSpace , 104 widget FloatingActionButton , 119–120, 180 instrucciones de flujo, 51–55, 64 Flutter. Consulte Google Flutter Complemento de Flutter, 23 Acerca del SDK de Flutter, 23 instalación, 13–19 Método FlutterMethodChannel , 323 FocusNode, 373 carpetas/archivos, creación y organización, 65–68 diseño de imágenes de pie de página, 257 bucles for , 52–53 Widget de formulario , 139 –143

Clase FormatDates , 407–408, 465, 487 formato de fechas, 331–332 FormState método de validación, 139, 143 propiedad fullscreenDialog , 178–179 funciones, 55–57, 64 Objeto futuro , 61–62, 373 Widget de FutureBuilder , 333–334, 360, 371, 373

GRADO barra de estado de gestos, agregando InkWell y InkResponse a, 291–296

GestureDetector() widget Clase AnimationController y, 168 Hero widget y, 189 interactividad y, 267–274, 305 configuración, 267–274 animaciones escalonadas y, 173 uso para mover y escalar, 278–284 método getApplicationDocumentsDirectory() , 330–331 método getDeviceInfo() , 312, 320 método getFlutterView() , 321 método getIconList() , 234 método getJournal , 446–447 temas de aplicaciones globales, 33–35 Google Cloud Platform, 383 Google Flutter. Consulte también temas específicos sobre, 3–4, 22 sitio web, 13 archivo GoogleService-Info.plist , 410 archivo google-services.json , 410 widget GridView , 143, 230–236, 251

## H

Configuración del proyecto de la aplicación Hello World, 25–30 widgets sin estado y con estado, 37–38 usando paquetes externos, 38–41 usando recarga en caliente, 30–32 usando temas para diseñar su aplicación, 33–36 Widget de héroe , 188–192, 219 página de inicio, modificación, 465–472 Clase HomeBloc , 441–443, 486

## Clase HomeBlocProvider : aplicación de diario

Clase HomeBlocProvider , 443–444, 460–464, 465–472,  
486 home.dart,  
75 hot reload, 30–  
32, 42

## I

Widget de ícono , 132–135, 266  
Widget IconButton , 122–123 iconos,  
130–135 instrucción  
if , 51–52  
Widget de imagen , 131–132, 255, 266  
imágenes, 130–135, 150  
implementando  
  clase abstracta , 414–415  
  Canal de plataforma de host Android, 318–321  
  programación asíncrona, 61–62, 64  
  Patrón BLoC, 417–419  
  Aplicación de canal de plataforma de cliente, 309–313  
  Widget arrastrable , 275–277  
  Widget de arrastrar objetivo , 275–277  
  Método FlutterMethodChannel , 323  
  Clase InheritedWidget , 415–416 método  
  de método de invocación , 323  
  canal de plataforma de host iOS, 313–318  
  MethodChannel, 323 clase  
  de modelo, 416  
  paquetes, 40–41  
  clase de servicio,  
  417 método setMethodCallHandler , 323  
Propiedad del fregadero , 419–  
420 gestión estatal, 412–420  
Corriente de clase, 419–420  
Widget de StreamBuilder , 419–420  
  clase StreamController , 419–420 paquetes  
de importación, 57  
declaración de importación,  
57 paquetes de importación,  
64 método de aumento de ancho () , 152, 153  
Clase InheritedWidget , 6, 415–416, 418, 421, 451, 486  
  inicialización  
de paquetes, 40–41 método  
initState() about , 7

Clase AnimationController y, 167 aplicación de  
canal de plataforma de cliente y, 312

interactividad y, 298  
aplicación de diario y, 349–350, 456, 460, 466, 474  
Widget de navegador y, 185  
animaciones escalonadas y, 171–172  
Widget InkResponse , 289–296, 305  
Widget InkWell , 189, 289–296, 305  
Clase InputDecoration , instalación 135–  
136  
  Android Studio en Linux, 19  
  Android Studio en macOS, 14–15  
  Android Studio en Windows, 16–17  
  Complemento de dardo, 20  
  Complemento de aleteo, 20  
  SDK de Flutter, 13–19  
  Xcode en macOS,  
  aplicación de  
    interactividad 14, 267–305  
    Widget descartable , 296–303, 305  
    Widget arrastrable , 275–277, 305  
    Widget de arrastrar objetivo , 275–277, 305  
    widget GestureDetector , 267–274, 278–289,  
      305  
    Widget InkResponse , 289–296, 305  
    Widget de InkWell , 289–296, 305  
  Intervalo(), animaciones escalonadas y, 170–171 paquete  
  intl , 374 método de  
    método de invocación , 323 canal  
    de plataforma de host iOS, implementación, 313–318

## j

JIT (justo a tiempo), 44  
aplicación de  
  diario que agrega el patrón BLoC, 432–  
  449 agrega la base de datos de Cloud Firestore, 391–  
  395 agrega la página de edición del diario,  
  472–484 agrega Firebase y Firestore, 375–410  
  agrega las clases de la base de datos del diario, 339  
  –344 agregar página de entrada de diario,  
  344–359 agregar clase de modelo de diario , 422–  
  424 agregar página de inicio de  
  sesión, 454–460 agregar clases de  
  servicio, 424–430 agregar clase de  
  validadores , 430–432  
  construir, 335–371 construir aplicación de diario de cliente, 395 –408

terminando la página de inicio, 359–371  
poniendo los cimientos de, 337–339  
modificando la página de inicio, 465–472  
modificando la página principal, 460–464  
Clase de diario , 330–331, 339–344, 373, 422–424  
Clase JournalEdit , 330–331, 339–344, 373  
Clase JournalEditBloc , 444–447, 486  
Clase JournalEditBlocProvider , 447–449, 472–484  
método Journal.fromDoc() , 423–424  
formato JSON,  
328–330 clases  
de base de datos y, 330–331  
justo a tiempo (JIT), 44

## L

diseños  
sobre, 253–255  
agregar a la aplicación de diario del cliente, 403–406 construir, 253–266  
crear, 257–265 final, 257  
imágenes de pie de página, 257  
etiquetas, 256–257 sección meteorológica, 256 propiedad principal, 104 carpeta lib , 25–26 Linux, instalando el SDK de Flutter en, 17–19 método listen() , 419  
listas, variables para, 47–48  
List().sort, 373 Widget ListTile, 223–229, 251, 367, 471 Widget ListView , 207–217 , 219, 223–229, 251, 373, 487 método loadJournals() , 371

operadores lógicos, 50 página de inicio de sesión, adición, 454–460 clase LoginBloc , 436–440, 454–455, 486

macOS, instalación de Flutter SDK en, 13–15 punto de entrada principal() , ejecución, 44–45  
página principal, modificación, 460–464  
función principal() , 9, 62, 64

main.dart, 75  
mapas, variables para, 48 método markTripComplete() , 299–300, 303 widgets de componentes materiales, 26 widget MaterialApp , 69, 188, 462 clase MaterialPageRoute , 178, 180–181 método Matrix4 giratorioZ() , 487 MethodChannel , 309–313, 323 métodos addOrEditJournal(), 338–339, 362–364, 368, 371, 467, 471, 472 addOrUpdateJournal(), 476 animatedBalloon () , 172, 174 animatedOpacity() , 160–163 askToOpen() , 53 build() , 6 , 7 , 8 , 11 buildBody() , 259–260 buildCompleteTrip() , 299, 300 buildDraggable() , 275–276 buildDragTarget() , 276 buildJournalEntry() , 260, 261 buildJournalFooterImages() , 260, 263–265 buildJournalHeaderImage() \_ \_ \_ \_ \_ –472 copyWith(), 36 createElement, 9–10 crossFade() , 155–156 deactivate() , 8 deleteTrip() , 299–300, 303 didChangeDependencies() , 8, 474 didFinishLaunchingWithOptions() , 316 didUpdateWidget() , 8 dispose() , 7, 350, 433, 438, 442, 456, 460, 475

FlutterMethodChannel, 323  
FormState validar, 139, 143  
getApplicationDocumentsDirectory() , 330–331  
getDeviceInfo() , 312, 320  
getFlutterView() , 321  
getIconList() , 234

METRO

getJournal, 446–447  
 aumentarAncho(), 152, 153  
 initState(), 7, 167, 171–172, 185, 298, 312, 349–350,  
 456, 460, 466, 474 invocar Método,  
 323 Journal.fromDoc  
 ( ), 423–424 listen(), 419  
 loadJournals(),  
 371 markTripComplete(),  
 299–300, 303 Matrix4 rotateZ(), 487  
 Navigator.pop, 178, 187  
 notImplemented(), 320 of(),  
 416 onAuthChanged(), 433  
 onCreate,  
 320 onDoubleTap(), 287,  
 288 onLongPress(),  
 289, 293–294, 295 onPressed(),  
 153, 345, 353 onScaleStart(), 288  
 onScaleUpdate(), 288  
 openPageAbout(), 179–  
 181, 182 openPageGratitude  
 (), 181–182 radioOnChanged(), 184  
 readAsString(), 331 refactorización con,  
 86–91 resetToDefaultValues(),  
 287–289, 294 rotateZ(),  
 471, 472–484, 478–484  
 runApp(), 9 runTransaction (), 424–425  
 saveJournal(), 447 setMethodCallHandler,  
 323  
 setScaleBig, 293–294, 295  
 setScaleSmall, 293–294,  
 295 setState(), 6–7, 8, 11, 37, 152,  
 155–156,  
 160–163, 195, 199, 285–287, 294, 310–312, 362–  
 363, 412  
 showDialog(), 467  
 showTimePicker(), 345, 472–484, 487 sink.add(),  
 434, 441 tabChanged,  
 205–206 signOut(), 434  
 vacío \_ Widget build(),  
 456–457, 462, 463, 469, 476 writeAsString(), 331  
 métodos, refactorización  
 con, 86–91 comando mkdir , 75 clase  
 modelo, 416, 450

carpeta de modelos , 421  
 modificación  
 de la página de inicio, 465–  
 472 página principal, 460–464  
 Clase MoodIcons , 407–408, 465, 471, 478, 487 en  
 movimiento, usando el widget GestureDetector para,  
 278–284  
 comentarios de varias líneas, 44

norte

### navegación

Widget BottomAppBar , 199–203, 219  
 Widget de barra de navegación inferior , 193–199,  
 219  
 crear para aplicaciones, 177–  
 219 Widget de cajón , 207–217,  
 219 Widget de héroe , 188–192,  
 219 Widget de ListView , 207–217,  
 219 Widget de navegador , 178–188,  
 219 Widget de TabBar , 203–207,  
 219 Widget de TabBarView , 203–207,  
 219 Widget Navigator , 178–188, 219, 373  
 Método Navigator.pop , 178, 187 Widgets  
 anidados, 100 Método  
 notImplemented() , 320 Variables  
 numéricas, 47 números,  
 declarados en JSON, 328–329

o

objetos, declarados en JSON, 328–329  
 método of() , 416  
 método onAuthChanged() , 433  
 método onCreate , 320  
 método onDoubleTap() , 287, 288  
 método onLongPress() , 289, 293–294, 295 método  
 onPressed() , 153, 345, 353 método  
 onScaleStart() , 288 método  
 onScaleUpdate() , 288 método  
 openPageAbout() , 179–181, 182 método  
 openPageGratitude() , 181–182 operadores, 49–  
 51, 64 organización de  
 carpetas/archivos, 65–68 orientación,  
 comprobación, 143–149, 150  
 Widget de OrientationBuilder , 143

PAG

**paquetes**

- cloud\_firestore, 410
- firebase\_auth, 410 importar,
  - 64 intl, 374
- path\_provider,
  - 374 buscar, 39–40 usando,
    - 40–41

Widget de relleno , 78, 266 carpeta de páginas , 421

paquete path\_provider , 374 persistencia sobre, 327–

- 328 creación de aplicación de diario, 335–371 clases de base de datos y JSON, 330–331 fechas de formato, 331–332 formato JSON, 328–330 recuperación de datos con el widget FutureBuilder , 333–334

guardar datos con local, 327–374 ordenar listas de fechas, 332–333 canales de plataforma, 307–308 código nativo de plataforma implementar canal de plataforma de host Android, 318–321 implementar aplicación de canal de plataforma de cliente, 309–313

implementación de canal de plataforma de host iOS, 313–318

canales de plataforma, 307–308

escritura, 307–323

Widget de botón de menú emergente , 123–126

Widget posicionado , 79

declaración de impresión,

56 declaración de

impresión ,

- 53 acciones de

- propiedades ,

- 104 secundario, 108 confirmar

- descartar, 487 espacio

- flexible, 104 diálogo de pantalla completa,

- 178–179 principal, 104

Fregadero, 419–420, 450

título, 104

R

**método radioOnChanged() , 184**

Widget RaisedButton , 121–122 método

readAsString() , 331 refactorización con constantes, 86

- con métodos, 86–91 con

- clases de widget, 91–98

- variables de referencia, 45–46

operadores relacionales, 49 árbol de representación, 9, 23

RenderObject, 9, 22

reemplazo de texto con contenedores secundarios RichText , 113–114

método resetToDefaultValues() , 287–289, 294 recuperación de datos con el widget FutureBuilder , 333–334 declaración nula de retorno,

55–56

Widget de texto enriquecido , 104, 112–114

carpeta raíz, 421

método de rotarZ() , 471, 472–484, 478–484 nombre de ruta, para navegación, 188 anidamiento de filas, 115–117

Widget de fila , 78, 104, 115–119, 255, 266 método

runApp() , 9 runas, como

variables, 48–49 aplicaciones en ejecución,

- 30–32 punto de

- entrada principal() , 45 método

runTransaction() , 424–425

S

Widget SafeArea , 104, 107–108, 266 método

saveJournal() , 447 guardar datos con persistencia local, 327–374

Widget Scaffold , 77, 78, 104 escalas,

usando el widget GestureDetector para, 278–284 listas de

desplazamiento/efectos

- Widget de tarjeta , 222–223, 251

- crear, 221–251

- personalizar el widget CustomScrollView con astillas, 243–250, 251

- Widget GridView , 230–236, 251

Widget ListTile , 223–229, 251  
Widget ListView , 223–229, 251  
Widget de pila , 237–243, 251 reglas  
de seguridad, Cloud Firestore, 381–383, 410 clases de servicio sobre,  
450 adición,  
424–430  
implementación, 417  
carpeta de servicios, 421  
método setMethodCallHandler , 323 método  
setScaleBig , 293–294, 295 método setScaleSmall ,  
293–294, 295 método setState()  
  
about, 6–7, 8, 11, 37 widget  
AnimatedContainer y, 152, 155 widget AnimatedCrossFade  
y, 155–156 AnimatedOpacity y, 160–163 widget  
BottomNavigationBar y, 195, 199 aplicación  
de canal de plataforma cliente y, 311–312 widget  
GestureDetector y , 285–287 interactividad y, 294, 300  
aplicación de diario y, 362–363 administración de  
estado y, 412 configuración de  
Android Emulator en Linux, 19  
Android Emulator en macOS, 15  
Android  
Emulator en Windows, 17 GestureDetector  
widget, 267–274 de proyecto para Aplicación  
Hello World, 25–30 función showDatePicker() ,  
345, 359, 373, 478 método showDialog() , 467  
método showTimePicker() , 345, 472–484, 487  
widget SingleChildScrollView , 78, 266 comentarios de una sola  
línea, 44 Sink propiedad, 419–420,  
450 método sink.add() , 434, 441 widget SizedBox , 256, 257,  
266 Skia, 4, 22 widget SliverAppBar , 243–250 widget  
SliverGrid , 243–250 widget  
SliverList , 243–250 fragmentos,  
personalización del widget  
CustomScrollView con, 243–250, 251  
widgets  
slivers, 221 widget SliverSafeArea , 243–  
250 clasificación de listas de fechas,  
332–333  
  
corchetes, 328  
Apilar widget, 79, 237–243, 251 animación  
escalonada, 170–175 plantilla de  
proyecto de inicio creación de  
carpetas/archivos, 65–68 organización  
de carpetas/archivos, 65–68  
estructuración de widgets, 69–74  
State class, 30, 37 state  
management about,  
450 agregando  
a la aplicación cliente de Firestore, 411–451  
construyendo, 421–449  
implementando, 412–420  
Objeto de estado , 6, 11  
widget con estado  
sobre, 37–38  
árboles de elementos y, 10–12  
clase StatefulWidget , 6–8, 22, 37 widget sin  
estado sobre, 37–38  
árboles de  
elementos y, 9–10  
Clase StatelessWidget , 6, 22  
Corriente de clase, 61–62, 419–420, 450  
Widget de StreamBuilder , 419–420, 450, 460–484,  
486  
Clase StreamController , 419–420, 450 variables de  
cadena, 47 cadenas,  
declaradas en JSON, 328–329 estructuración  
  
Cloud Firestore, 377–379 widgets,  
69–74 diseñar  
aplicaciones usando temas, 33–36  
declaración de cambio , 55

## T

Widget de barra de pestañas , 203–207, 219  
Widget TabBarView , 203–207, 219 método  
tabChanged  
Widget de barra de pestañas y, 205–206  
Widget TabBarView y, 205–206 diseño de  
etiquetas, 256–257 operador  
ternario, 52 campos de  
texto usando  
formularios para validar, 150 validar con  
widget de formulario , 139–143

Widget de texto , 11–12, 78, 104, 112–114, 178, 255, 266  
Controlador de edición de texto, 373  
Widget de campo de texto , 373, 472–484  
TextInputAction, 373 temas aplicados,  
    42 aplicaciones  
    de estilo usando, 33–36  
Clase de teletipo , 164  
título de propiedad, 104  
Transform widget, 278–283, 471  
operadores de prueba de tipos, 50

EN

Unicode, 48–49

EN

validación de campos de texto usando el widget Formulario , 139–143  
Clase de validadores , 430–432, 436, 440, 450 variables  
  
Booleano, 47  
declaración, 46–49  
listas como, 47–48  
mapas como,  
48 número, 47  
referencias, 45–46 runas  
como, 48–49 cadena,  
47 usando,  
64  
visualización  
    Reglas de seguridad de Cloud Firestore, 381–383  
    Capacidades de autenticación de Firebase, 380–381 Método  
void\_signOut() , 434

EN

diseño de la sección meteorológica,  
256 sitios web  
    Panel de la consola Firebase, 391  
    Flutter de Google, 13  
bucles while , 53–54  
Método build() de widget , 456–457, 462, 463, 469, 476

árboles de  
widgets sobre, 8–9,  
22 construcción completa,  
79–85 construcción superficial,  
85–98 creación completa,  
100 creación superficial, 100  
refactorización con constantes, 86  
refactorización con métodos, 86–91  
refactorización con clases de widgets, 91–98 widgets  
descripción general, 77–79 widgets  
sobre, 5,  
22, 103  
Constructor animado, 176  
Contenedor animado, 152–155, 176  
Fundido cruzado animado, 155–159, 176  
Opacidad animada, 160–163, 176  
Barra de aplicaciones, 77, 78, 104, 105–107  
BottomAppBar, 199–203, 219  
Barra de navegación inferior, 193–199, 219  
Barra de botones, 126–130  
Botones, 104, 119–130  
Tarjeta, 222–223, 251  
Center(), 469  
comprobación de la orientación, 143–149  
Chip, 256, 266  
CírculoAvatar, 78, 266  
Columna, 104, 114–119  
Columna() , 36, 78, 104, 114–119, 254, 256, 257, 266, 471  
  
Contenedor, 78, 104, 108–112, 469, 470  
Aplicación de Cupertino, 69  
CupertinoNavigationBar, 78  
CupertinoPageAndamio, 78  
CupertinoTabAndamio, 78  
CustomScrollView, 221, 243–250, 251  
Desechable, 296–303, 305, 360, 365, 371, 373, 465–  
    472, 470, 487  
Divisor, 78, 261, 266, 470, 487  
Arrastrable, 271, 275–277, 305  
Objetivo de arrastre, 271, 275–277, 305  
Cajón, 207–217, 219  
Botón desplegable(), 487  
Ampliado, 78  
Botón plano, 121  
Botón de acción flotante, 119–120, 180

Formulario, 139–143  
Constructor del futuro, 333–334, 360, 371, 373  
Detector de gestos(), 168, 173, 189, 267–274, 278–284, 278–289, 305  
GridView, 143, 230–236, 251  
Héroe, 188–192, 219  
Icono, 132–135, 266  
Botón de ícono, 122–123  
Imagen, 131–132, 255, 266  
Respuesta de tinta, 289–296, 305  
InkWell, 189, 289–296, 305 eventos del ciclo de vida de, 5–8, 22  
ListTile, 223–229, 251, 367, 471  
ListView, 207–217, 219, 223–229, 251, 373, 487  
MaterialApp, 69, 188, 462  
Navegador, 178–188, 219, 373  
anidamiento, 100  
OrientationBuilder, 143 descripción general de, 77–79  
Acolchado, 78, 266  
Botón de menú emergente, 123–126  
Posicionado, 79  
Botón elevado, 121–122  
Texto enriquecido, 104, 112–114  
Fila, 78, 104, 115–119, 255, 266  
SafeArea, 104, 107–108, 266  
Andamio, 77, 78, 104  
SingleChildScrollView, 78, 266  
Caja de tamaño, 256, 257, 266  
SliverAppBar, 243–250  
SliverGrid, 243–250  
AstillaLista, 243–250 astillas, 221  
SliverSafeArea, 243–250  
Pila, 79, 237–243, 251 con estado, 10–12, 37–38 sin estado, 9–10, 37–38  
StreamBuilder, 419–420, 450, 460–484, 486 estructuración, 69–74  
TabBar, 203–207, 219  
TabBarView, 203–207, 219  
Texto, 11–12, 78, 104, 112–114, 178, 255, 266  
Campo de texto, 373, 472–484  
Transform, 278–283, 471 usando basic, 103–130, 150 usando decoradores, 135–139 usando formulario para validar campos de texto, 139–143 usando widget de formulario para validar campos de texto, 139–143 uso de imágenes e iconos, 130–135 Envolver, 266  
Windows, instalando Flutter SDK en, 15–17  
Widget de ajuste , 266  
método writeAsString() , 331 escritura de código nativo de la plataforma, 307–323

## X

Acerca de Xcode, 23 instalación en macOS, 14