Flutter Bootcamp

Responsive Layout, Testing, Semantics, i18n y l10n

Emanuel López – emanuel.lopez@globant.com

Globant

Repaso

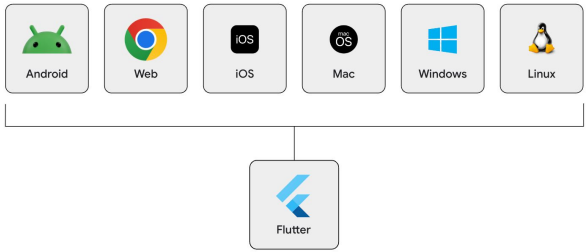
- Animaciones
  - basadas en código vs basadas en dibujos
- Animaciones Implícitas
  - `AnimatedContainer, AnimatedOpacity, AnimatedAlign, ...`
  - `TweenAnimationBuilder`
- Animaciones Explícitas
  - `FadeTransition, ScaleTransition, SlideTransition, ...`
  - `vsync`
  - `AnimatedWidget` y `AnimatedBuilder`
- Animaciones `Lottie`
- Animaciones `Rive`
- Implementando UI complejas

Repositorio del Bootcamp: bit.ly/FlutterBootcampGlobant

Globant

Diseño adaptativo y responsivo en Flutter

Uno de los objetivos principales de Flutter es crear un framework que nos permita **desarrollar aplicaciones a partir de un único código fuente tal que se vean y funcionen bien en cualquier plataforma.**



Esto significa que nuestra aplicación pueda usarse en pantalla de distintos tamaños, desde un reloj hasta un teléfono plegable con dos pantallas o un monitor de alta definición. Y el dispositivo de entrada puede ser un teclado físico o virtual, un mouse, una pantalla táctil o cualquier otro dispositivo.

Dos términos que describen estos conceptos de diseño son **adaptativo** y **responsivo**. Lo ideal sería que nuestra aplicación fuera ambas cosas, pero ¿qué significa esto exactamente?

¿Qué es responsivo y adaptativo?

- El diseño **responsivo** consiste en adaptar la interfaz de usuario al espacio
- El diseño **adaptativo** consiste en que la interfaz de usuario **se pueda utilizar** en ese espacio.

Por lo tanto:

- una aplicación **responsiva ajusta la ubicación de los elementos** de diseño para adaptarse al espacio disponible
- una aplicación **adaptable selecciona el diseño y los dispositivos de entrada adecuados** para que se puedan usar en el espacio disponible.

Por ejemplo, la interfaz de usuario en una tablet, ¿debería utilizar una navegación inferior o una navegación en el panel lateral?

¿Cómo se puede abordar una aplicación diseñada para dispositivos móviles convencionales y hacerla compatible con una amplia gama de dispositivos? ¿Qué pasos son necesarios?

Primer paso: Abstraernos

En primer lugar, analizar los widgets que pensamos hacer dinámicos y **extraer los datos que podemos compartir**. Los widgets comunes que requieren adaptabilidad son:

- **Diálogos**, tanto de pantalla completa como modales
- **Interfaz de navegación**, tanto en el navigation rail como en la bottom navigation bar.
- **Diseño personalizado**, como "¿el área de la interfaz de usuario es más alta o más ancha?"

Ej: Para la navegación en nuestra aplicación tal vez podríamos usar:

- un `BottomNavigationBar` cuando la ventana de la aplicación es pequeña
- un `NavigationRail` cuando la ventana de la aplicación es grande.

Estos widgets seguramente compartirán una lista de destinos navegables. Podemos crear un widget `Destination` para almacenar esta información y un ícono y una etiqueta de texto.

Segundo paso: Medir: MediaQuery

A continuación, evaluaremos el tamaño de la pantalla para decidir cómo mostrar nuestra interfaz de usuario. Tenemos dos formas de medir el espacio de la pantalla:

- `MediaQuery.sizeOf`: si queremos el tamaño de toda la ventana de la aplicación.
- `LayoutBuilder`: si queremos un tamaño más local.

Si deseamos que nuestro widget ocupe toda la pantalla utilizaremos `MediaQuery.sizeOf` para poder elegir la interfaz de usuario en función del tamaño de la ventana de la aplicación.

Solicitar el tamaño de la ventana de la aplicación desde dentro del método `build`, como en `MediaQuery.sizeOf(context)`, hace que el método dado `BuildContext` se reconstruya cada vez que cambia la propiedad de tamaño.

Segundo paso: Medir: LayoutBuilder

`LayoutBuilder` logra un objetivo similar al de `MediaQuery.sizeOf`, con algunas distinciones.

En lugar de proporcionar el tamaño de la ventana de la aplicación, `LayoutBuilder` proporciona las restricciones de diseño del elemento padre Widget. Esto significa que obtendremos información de tamaño en función del lugar específico en el árbol de widgets donde agregó el elemento `LayoutBuilder`.

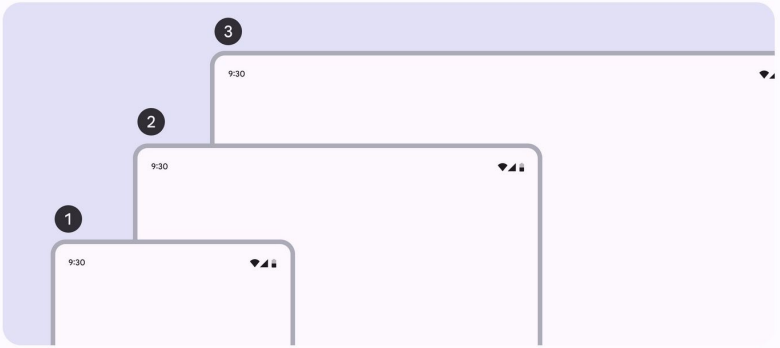
`LayoutBuilder` devuelve un objeto `BoxConstraints` que proporcionan los rangos válidos de ancho y alto (mínimo y máximo) para el contenido, en lugar de solo un tamaño fijo. Esto puede ser útil para widgets personalizados.

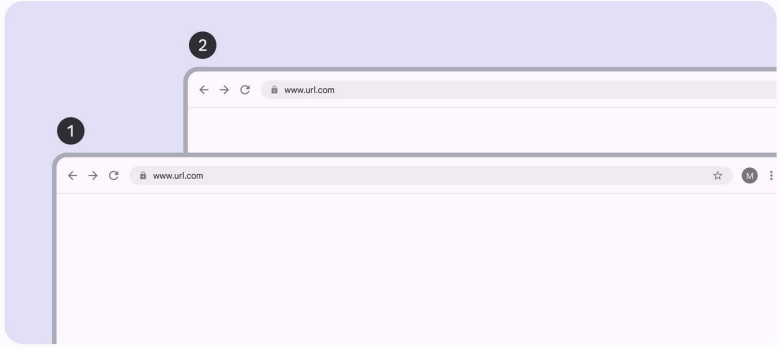
Por ejemplo, imaginemos un widget personalizado, en el que queremos que el tamaño se base en el espacio asignado específicamente a ese widget, y no en la ventana de la aplicación en general. En este escenario es mejor usar `LayoutBuilder`.

Tercer paso: Dividir

En este punto, debemos decidir que **breakpoints** vamos a utilizar para cada versión de la interfaz de usuario.

Las guías de diseño de **Material** sugieren utilizar una barra de navegación inferior para ventanas de menos de 600 píxeles lógicos de ancho y un riel de navegación lateral para aquellas que tengan 600 píxeles de ancho o más.





Globant

Material Design Guidelines: Layout

Window class (width)	Breakpoint (dp)	Common devices
Compact	Width < 600	Phone in portrait
Medium	600 ≤ width < 840	Tablet in portrait Foldable in portrait (unfolded)
Expanded	840 ≤ width < 1200*	Phone in landscape Tablet in landscape Foldable in landscape (unfolded) Desktop

Window class (width)	Breakpoint (dp)	Common devices
Large	1200 ≤ width < 1600	Desktop
Extra-large	1600 ≤ width	Desktop Ultra-wide

Material Design guidelines: m3.material.io/foundations/layout/applying-layout/window-size-classes

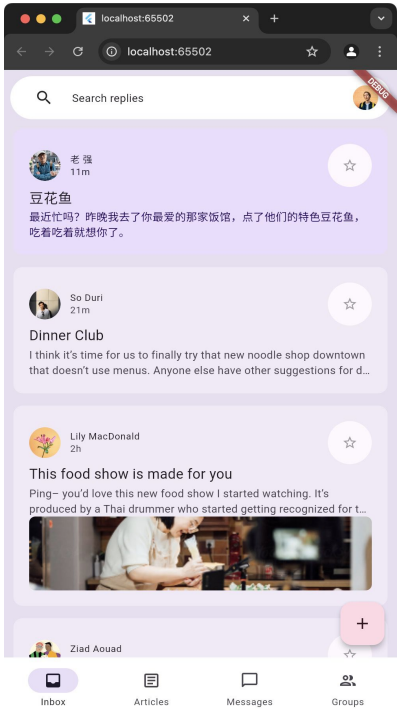
Globant

Material Design Guidelines: Layout

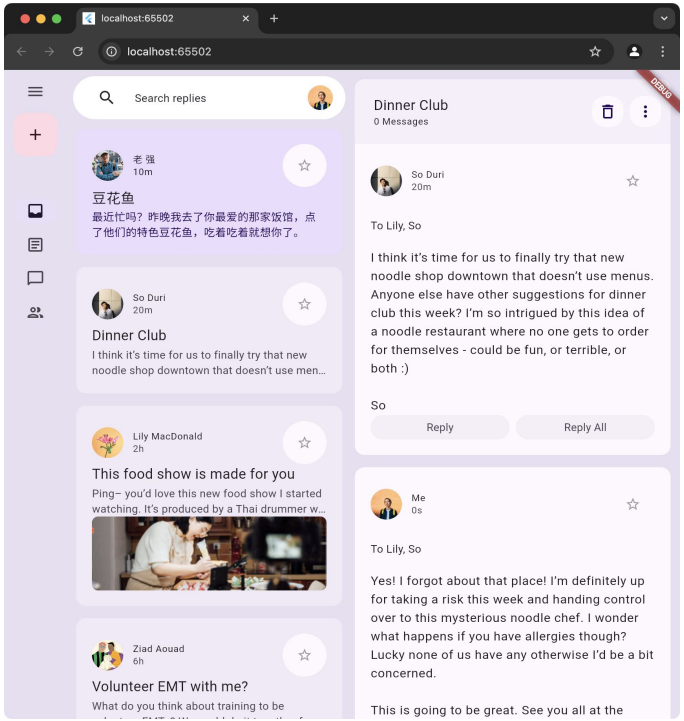
Window class	Panes	Navigation	Communication	Action
Compact	1	Navigation bar, modal navigation drawer	Simple dialog Fullscreen dialog	Bottom sheet
Medium	1 (recommended) or 2	Navigation rail, modal navigation drawer	Simple dialog	Menu
Expanded	1 or 2 (recommended)	Navigation rail, modal or standard navigation drawer	Simple dialog	Menu
Large	1 or 2 (recommended)	Navigation rail, modal or standard navigation drawer	Simple dialog	Menu
Extra-large	1 to 3 (recommended)	Navigation rail, modal or standard navigation drawer	Simple dialog	Menu

Material Design guidelines: m3.material.io/foundations/layout/applying-layout/window-size-classes

Animated responsive app layout with Material 3



BottomNavigationBar



NavigationRail

Codelab: [Building an animated responsive app layout with Material 3]

Globant

SafeArea

Al correr nuestra aplicación en dispositivos más recientes, es posible que nos encontremos con partes de la interfaz bloqueadas por recortes en la pantalla.

El widget `SafeArea` nos ayuda a evitar estas intrusiones (como notches y recortes para la cámara), así como la interfaz de usuario del sistema operativo (como la barra de estado en Android) o mediante esquinas redondeadas de la pantalla física.

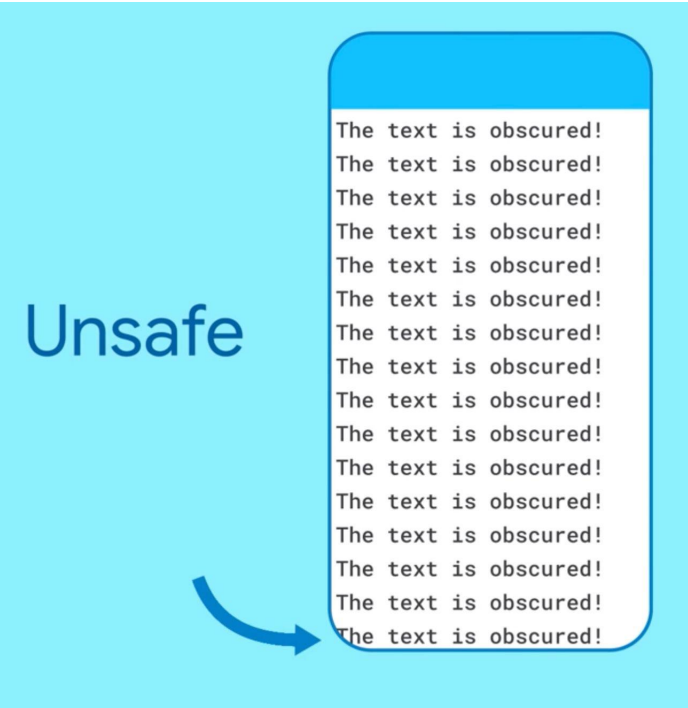
`SafeArea` nos permite desactivar el relleno en cualquiera de sus cuatro lados. De manera predeterminada, los cuatro lados están habilitados.

```
SafeArea(
  bottom: true,
  left: true,
  right: true,
  top: true,
  child: ListView(...)
);
```

Generalmente se recomienda envolver el `body` de un `Scaffold` como para comenzar, pero no siempre es necesario colocarlo tan alto en el árbol de widgets.

SafeArea

```
ListView(
   children: List.generate(
     100,
     (i) => Text('This is some text')
   ),
)
```

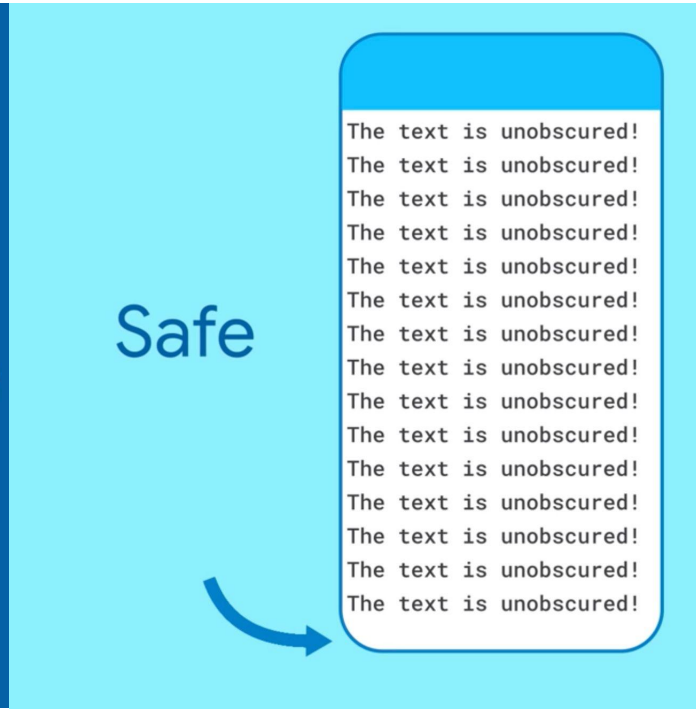


Fuente: SafeArea (Flutter Widget of the Week)

Globant

SafeArea

```
SafeArea(
  child: ListView(
    children: List.generate(
      100,
      (i) => Text('This is some text')
    ),
  )
)
```



Fuente: SafeArea (Flutter Widget of the Week)

Globant

MediaQuery

`MediaQuery` se utiliza para obtener información sobre las características del dispositivo y del entorno en el que se está ejecutando la aplicación. Esto nos permite ajustar nuestra interfaz de usuario para que sea responsiva y se adapte correctamente a diferentes pantallas y dispositivos.

Usos comunes de `MediaQuery`:

- **Obtener el tamaño de la pantalla:** Puedes utilizar `MediaQuery` para saber el ancho y alto de la pantalla del dispositivo, lo cual es útil para hacer que la UI sea responsiva o para adaptar elementos en función de las dimensiones disponibles.

```
var screenSize = MediaQuery.of(context).size;
var screenWidth = screenSize.width;
var screenHeight = screenSize.height;
```

MediaQuery

- **Determinar la orientación de la pantalla:** Puedes detectar si el dispositivo está en modo vertical (portrait) o horizontal (landscape).

```
var orientation = MediaQuery.of(context).orientation;
if (orientation == Orientation.portrait) {
  // Modo vertical
} else {
  // Modo horizontal
}
```

- **Acceder al padding del sistema:** Esto es útil para evitar que tu contenido sea obstruido por áreas del sistema como el notch, la barra de estado, o los bordes en dispositivos con pantallas curvas.

```
var padding = MediaQuery.of(context).padding;
var topPadding = padding.top;           // Ej. altura de la barra de estado
var bottomPadding = padding.bottom;     // Ej. área de gestos en iPhone
```

MediaQuery

- **Detectar la densidad de píxeles (pixel density):** La densidad de píxeles determina cuántos píxeles hay por unidad de espacio físico. Esto es útil para ajustar imágenes o elementos gráficos para que se vean nítidos en diferentes dispositivos.

```
var pixelRatio = MediaQuery.of(context).devicePixelRatio;
```

- **Ajustar el tamaño de texto o elementos gráficos:** Puedes usar MediaQuery para escalar el texto o los widgets en función del tamaño de la pantalla.

```
var textScaleFactor = MediaQuery.of(context).textScaleFactor;
```

Testing

Cuanto más funciones tenga nuestra aplicación, más difícil será probar que todo funcione bien manualmente.

Los tests automatizados ayudan a garantizar que nuestra aplicación funciona correctamente antes de publicarla, a la vez que conservan la velocidad de corrección de errores y funciones.

Los tests automatizados se dividen en las siguientes categorías:

- **Unit test:** testean una sola función, método o clase.
- **Widget test:** testean un solo widget.
- **Integration test:** testean una aplicación completa o una gran parte de una aplicación.

Codelab: How to test a Flutter app: codelabs.developers.google.com/codelabs/flutter-app-testing

Unit tests o pruebas unitarias

Las pruebas unitarias (o unit tests) son un tipo de prueba de software que **se centra en probar componentes pequeños e individuales de un programa**, como funciones, métodos, o clases, de manera aislada.

El objetivo de las pruebas unitarias es verificar que estos componentes funcionan correctamente de forma independiente.

Estas pruebas **no** deben depender de otras partes del sistema, como la interfaz gráfica (UI), redes, bases de datos, etc. **Sólo prueban la lógica del componente específico.**

Si es necesario, **se utilizan mocks para simular comportamientos de dependencias externas.**

Unit tests o pruebas unitarias

1. Refactorizamos la aplicación de ejemplo para extraer y aislar la lógica:

```
class Counter {
  int value = 0;
  void increment() => value++;
  void decrement() => value--;
}
```

2. Agregamos la dependencia necesaria: `flutter pub add dev:test`
3. Agregamos los unit tests: `counter_test.dart`
4. Corremos los tests: `flutter test test/counter_test.dart`
5. Si usamos mockito debemos ejecutar `dart run build_runner build -d` para generar los mocks

Widget test o prueba de widget

Un widget test testea un único widget. El objetivo es verificar que la interfaz de usuario del widget se ve e interactúa como se espera.

El widget que se está testeando debe poder recibir y responder a las acciones y eventos del usuario, realizar el diseño y crear instancias de widgets secundarios. Por lo tanto, **un widget test es más completo que una prueba unitaria.**

Al igual que en un Unit test, el entorno de un widget test se reemplaza por una implementación mucho más simple que un sistema de interfaz de usuario completo.

Para testear widgets debemos asegurarnos que tenemos el paquete `flutter_test` entre las dependencias de desarrollo.

Integration test o pruebas de integración

Las pruebas de integración testean una aplicación completa o gran parte de ella. El objetivo es garantizar que las diferentes piezas de la aplicación trabajen en conjunto correctamente.

Estas pruebas se ejecutan en un dispositivo físico o en un emulador/simulador, y validan el comportamiento de la aplicación desde el punto de vista del usuario final.

Para las pruebas de integración necesitamos agregar la dependencia `integration_test`

```
flutter pub add 'dev:integration_test:{"sdk":"flutter"}'
```

Las pruebas de integración se agregan dentro del directorio `integration_test` en la raíz del proyecto.

Para ejecutar los test de integración:

```
flutter test integration_test/app_test.dart
```

Testing: resumen

Característica	Unit Tests	Widget Tests	Integration Tests
Velocidad	Muy rápida	Rápida	Lenta
Alcance	Funciones o métodos específicos.	Un widget y su comportamiento (incluye interacciones y cambios de estado).	Aplicación completa o secciones significativas (UI, lógica, API, etc.).
Entorno de Ejecución	Simulado (sin necesidad de UI o dispositivos).	Simulado (sin dispositivo, en el entorno de prueba).	Emulador/dispositivo real o simulador.
Ejemplo de Uso	Verificar que una función devuelva el resultado correcto.	Probar que un botón actualiza correctamente el estado de la UI.	Probar que el usuario puede navegar a través de pantallas y realizar una tarea completa.
Uso de Herramientas	`flutter_test` y frameworks de pruebas unitarias (como `mockito`).	`flutter_test`, `WidgetTester` y matchers de `find`.	`integration_test`, ejecución en dispositivos/emuladores.

Semantics

Semantics es un sistema que **proporciona información adicional sobre los widgets a servicios de accesibilidad**, como lectores de pantalla.

Esto es esencial para hacer que las aplicaciones sean accesibles para personas con discapacidades visuales o auditivas.

¿Qué hace el sistema de Semantics?

- Describe el propósito de un widget (como un botón, imagen o texto).
- Informa a los usuarios sobre el estado actual de un widget
- Proporciona acciones que un usuario puede realizar
- Proporciona etiquetas personalizadas para los widgets que pueden ser leídas por los lectores de pantalla.

Semantics

Muchos de los widgets de Material Design ya vienen con propiedades de semantics integradas, por lo que no necesitamos envolverlos con el widget `Semantics`.

Estos widgets proporcionan automáticamente la información semántica a los servicios de accesibilidad.

Algunos tienen la posibilidad de definir la propiedad `semanticLabel` para agregar la descripción del contenido:

```
Image.network(
    'https://picsum.photos/id/237/300/150',
    semanticLabel: 'Perro de raza Labrador color negro',
),
```

Semantics: widgets específicos

Algunos de los widgets que tenemos disponibles para proporcionar información de accesibilidad son:

- `Semantics`: Es el widget principal para este propósito. Se puede envolver cualquier widget dentro de un `Semantics` y configurar propiedades como `label`, `enabled`, `button`, etc.

```
Semantics(
  label: 'Botón para enviar el formulario',
  button: true,
  child: ElevatedButton(
    onPressed: () {...},
    child: Text('Enviar'),
  ),
)
```

- `ExcludeSemantics`: Se utiliza para excluir un widget o grupo de widgets del sistema de semantics. Esto es útil si un widget no debe ser visible para los servicios de accesibilidad.

Semantics: widgets específicos

Algunos de los widgets que tenemos disponibles para proporcionar información de accesibilidad son:

- `MergeSemantics`: Este widget combina varios widgets dentro de una misma región semántica. En lugar de que cada widget sea tratado por separado, `MergeSemantics` los combina para que sean vistos como una única entidad.

```
MergeSemantics(
  child: Row(
    children: [
      Icon(Icons.volume_up),
      Text('Volumen'),
    ],
  ),
);
```

Listado de todos los widgets de accesibilidad: docs.flutter.dev/ui/widgets/accessibility

Internacionalización (i18n) y localización (l10n)

Internacionalización es el **proceso de diseñar una aplicación de manera que pueda adaptarse a diferentes idiomas y regiones sin requerir cambios en el código fuente.** Esto implica preparar la aplicación para que sea capaz de manejar múltiples idiomas, formatos de fecha, monedas y otras configuraciones específicas de una región.

Localización es el proceso de **adaptar una aplicación para que se ajuste a un idioma y una cultura específicos**. Esto incluye la traducción de cadenas de texto, la adaptación de formatos de fecha y hora, la conversión de monedas, y la consideración de convenciones culturales, como colores y símbolos.

Librerías/paquetes utilizados:

```
flutter pub add intl flutter_localizations:{"sdk":"flutter"}'
```

Y ahora? Cómo seguimos?

Algunas recomendaciones sobre los próximos pasos que podemos seguir para continuar nuestro aprendizaje y mejorar nuestras habilidades en el desarrollo de aplicaciones con flutter:

- Creación de temas personalizados: docs.flutter.dev/cookbook/design/themes
- Explorar otras opciones para manejo de estado e inyección de dependencias: bloc, riverpod, etc.
- Trabajar con mapas de Google, geolocalización, notificaciones push
- Configurar CI/CD para pruebas y builds automáticos: GitHub Actions, Bitrise, etc.
- Explorar las plataformas nativas, principalmente Android e iOS
- Publicación en tiendas: docs.flutter.dev/deployment/obfuscate
- Contribuir en algún proyecto Open Source
- Explorar librerías de análisis de uso de nuestra aplicación: Analytics, Segment, MixPanel, etc.
- Para mas, revisar roadmap.sh/flutter

Links adicionales

- Adaptive and responsive design in Flutter: docs.flutter.dev/ui/adaptive-responsive
- Flutter Bottom Navigation Bar with Stateful Nested Routes using GoRouter: codewithandrea.com/articles/flutter-bottom-navigation-bar-nested-routes-gorouter
- Testing Flutter apps: docs.flutter.dev/testing/overview

Preguntas?

Globant

Gracias!

Globant