ESCUELA DE SISTEMAS Y TECNOLOGÍAS

Transparencias de **ANALISTA DE SISTEMAS**
Edición 2023 - Materia: **Aplicaciones Móviles con Flutter**

TEMA: Widgets

Agenda

- Widgets y Árbol de Widgets
- Stateless vs Stateful Widgets
- WidgetsApp
- Widgets Básicos
  - Scaffold, AppBar, Text, Icon, Image, ElevatedButton, Container, Column, Row
- Temas
- Mostrar Mensajes al Usuario
  - SnackBar
  - AlertDialog
- SafeArea
- MediaQuery
- GestureDetector
- FutureBuilder
- StreamBuilder

Widgets y Árbol de Widgets (1)

- En *Flutter* prácticamente todo es un **widget**. Podría decirse que son los bloques de construcción con los que se crea una aplicación.
- Un *widget* es una descripción inmutable de una parte de la interfaz de usuario de la aplicación. En sí mismos no tienen un estado mutable (es decir, que pueda cambiar).
- Podría tratarse de un botón, un cuadro de texto, una imagen, un contenedor para organizar visualmente otros elementos, una animación, etc.
- Los *widgets* se organizan creando una jerarquía en forma de árbol, de manera que un *widget* puede estar contenido en otro, y a su vez contener a otros más.

Widgets y Árbol de Widgets (2)

- El árbol de *widgets* es una estructura jerárquica que representa la interfaz de usuario en *Flutter*.
- Es un concepto fundamental que nos permite construir interfaces de usuario complejas a partir de *widgets* simples y reutilizables, así como también actualizarlas de manera eficiente.
- En el árbol de *widgets*, cada *widget* es un nodo que contiene información sobre cómo debe renderizarse y cómo reaccionar a eventos.
- Flutter utiliza el concepto de "**Reconciliación**" para actualizar la interfaz de usuario de manera eficiente sin afectar otras partes de la aplicación.

Widgets y Árbol de Widgets (3)

- La reconciliación (*reconciliation*) es el proceso por el cual se compara y actualiza el estado y la estructura del árbol de *widgets* para reflejar los cambios realizados en la interfaz de usuario.
- Cuando ocurre un cambio en el estado de la aplicación (como una interacción del usuario o una actualización de datos), *Flutter* utiliza un algoritmo eficiente de reconciliación para determinar qué *widgets* deben actualizarse y cómo deben hacerlo.
- Durante la reconciliación, *Flutter* compara la estructura y las propiedades de los widgets en el árbol de *widgets* actual con los *widgets* en el árbol de *widgets* anterior.

Widgets y Árbol de Widgets (4)

- De esta forma se detectan las diferencias y se aplican las actualizaciones necesarias. *Flutter* realiza cambios precisos y mínimos en la interfaz de usuario para garantizar un rendimiento óptimo.
- La reconciliación en *Flutter* es un proceso eficiente debido a la inmutabilidad de los *widgets*. En lugar de modificar los *widgets* existentes, *Flutter* crea nuevos *widgets* con las propiedades actualizadas.
- Esto permite que el *framework* compare los *widgets* de manera rápida y precisa, reduciendo el tiempo y los recursos necesarios para actualizar la interfaz de usuario.

Stateless vs Stateful Widgets (1)

- En *Flutter* existen diferentes tipos de *widgets*: están los *stateless widgets*, los *stateful widgets* e incluso también los *inherited widgets* (los veremos más adelante).
- Los **stateless Widgets** son aquellos que no cambian su estado. Se construyen en base a las configuraciones indicadas desde un principio y así permanecen durante todo su ciclo de vida.
- Por otra parte los **stateful widgets** crean un objeto **State** que les permite mantener un estado mutable, y redibujarse cuando el mismo cambie, mediante la función *setState(VoidCallback)*. El redibujado se realiza aplicando los cambios mínimos necesarios.

Stateless vs Stateful Widgets (2)

- Existe una gran cantidad de *widgets* ya definidos y listos para utilizar, así como también es posible definir nuestros propios *widgets* creando una jerarquía personalizada a partir de otros.
- Para definir un *stateless widget* se debe crear una clase que extienda de **StatelessWidget** y redefinir su operación *build(BuildContext)*.
- Dicha operación recibe por parámetro un objeto *BuildContext* que entre otras cosas contiene la información del árbol de *widgets* hasta el punto donde se construye el *widget*, y debe retornar el objeto *widget* raíz de la jerarquía que representa el *widget* que se está definiendo.

Stateless vs Stateful Widgets (3)

- Por otra parte, los *stateful widgets* se definen creando una clase que extienda de **StatefulWidget** y redefiniendo su operación *createState()*.
- Dicha operación debe retornar un objeto cuyo tipo corresponde a otra clase que se debe definir extendiendo de **State<TipoWidget>** para manejar el estado del *widget*.
- En este caso, es en la clase del estado donde se debe redefinir la operación *build(BuildContext)* para retornar el *widget* raíz de la jerarquía.
- También se puede redefinir el método *initState()* para realizar tareas de inicialización. Este método se llama una sola vez, cuando el estado es creado.

Stateless vs Stateful Widgets (4)

- Debido al carácter inmutable de los *widgets*, los atributos que se definan en su clase (ya sea de tipo *StatelessWidget* o *StatefulWidget*) deben ser *final*.
- No ocurre lo mismo en la clase del estado de un *StatefulWidget*, donde los atributos se espera que cambien de valor.
- El objeto del estado se conserva incluso cuando el árbol de widgets se vuelve a construir.
- Desde la clase del estado de un *StatefulWidget* se puede obtener una referencia al *widget* correspondiente mediante el atributo *widget*.
- Los constructores de los *widgets* pueden recibir parámetros para inicializar sus atributos.

Stateless vs Stateful Widgets (5)

- También se recomienda (si bien en la práctica no es muy frecuente su uso) que los constructores de los *widgets* reciban un parámetro con nombre *key* que se puede pasar al constructor de la clase base para inicializar el atributo correspondiente.
- Este atributo en general se utiliza para conservar el estado del *widget* cuando es movido a otra posición dentro de la jerarquía (por ejemplo si se tiene una lista de *widgets* y dicha lista es alterada).

WidgetsApp (1)

- Si bien al método *runApp(Widget)* se le puede pasar cualquier tipo de *widget* para colocar en la raíz del árbol de *widgets*, lo más común es pasarle un *widget* de tipo *WidgetsApp*, o alguno de sus derivados *MaterialApp* (es el más utilizado) o *CupertinoApp*.
- La clase **WidgetsApp** agrupa un conjunto de *widgets* para colocar en la raíz del árbol, que son comunmente requeridos por una aplicación.
- Asimismo las clases **MaterialApp** y **CupertinoApp** extienden de la clase *WidgetsApp* y agregan *widgets* con funcionalidades que son requeridas por las aplicaciones que siguen los estilos de diseño típicos de *Android* e *iOS* respectivamente.

WidgetsApp (2)

- Entre los parámetros más interesantes para construir un *widget* de tipo *MaterialApp* encontramos:
  - *String* **title**: Define el título de la aplicación.
  - *ThemeData?* **theme**: Define el tema (propiedades visuales como colores, fuentes y formas) predeterminado que utiliza la aplicación.
  - *Widget?* **home**: Widget que se mostrará cuando se acceda a la ruta por defecto de la aplicación (/), a menos que se defina el parámetro initialRoute.
  - *bool* **debugShowCheckedModeBanner**: Indica si debe mostrarse un indicador en la pantalla si la aplicación se ejecuta en modo debug.

Widgets Básicos

- En las siguientes diapositivas veremos algunos de los widgets clasificados como básicos en el catálogo de widgets de Flutter (https://docs.flutter.dev/development/ui/widgets/basics):

Scaffold	*AppBar*	*Text*
Icon	*Image*	*ElevatedButton*
Container	*Column*	*Row*

Scaffold

- El *widget* **Scaffold** provee la estructura visual de layout básica del estilo *Material Design*.
- Permite mostrar una *AppBar*, un contenido, uno o varios *FloatingActionButton*, etc.
- Los parámetros más comunes de *Scaffold* son:
  - *Color?* **backgroundColor**: Define el color de fondo del *layout*.
  - *PreferredSizeWidget?* **appbar**: Configura una barra de herramientas (típicamente un *AppBar*) donde se puede mostrar un icono, un título, acciones, etc.
  - *Widget?* **body**: Establece el contenido principal.
  - *Widget?* **floatingActionButton**: Permite configurar uno o más botones flotantes.

AppBar

- El *widget* **AppBar** consiste en una barra de herramientas que puede incluir otros *widgets*.
- Los parámetros más comunes de *AppBar* son:
  - *Widget?* **leading**: Muestra un *widget* al comienzo de la barra (típicamente un *Icon*).
  - *Widget?* **title**: Define el título (típicamente mediante un *Text*).
  - *List<Widget>?* **actions**: Lista de *widgets* para invocar acciones (típicamente una lista de *IconButton*).

Text

- El *widget* **Text** permite mostrar un texto y aplicarle estilos.
- Su constructor recibe un parámetro posicional de tipo *String* para especificar el texto a mostrar.
- Otros parámetros comunes de *Text* son:
  - *TextAlign?* **textAlign**: Establece la alineación del texto.
  - *TextStyle?* **style**: Configura los estilos aplicados al texto.
  - *int?* **maxLines**: Limita la cantidad máxima de líneas a mostrar.
  - *TextOverflow?* **overflow**: Define cómo mostrar (o no) el texto que excede el tamaño del *widget*.

Icon

- El *widget* **Icon** muestra un icono.
- Su constructor recibe un parámetro posicional de tipo *IconData* para especificar el icono a mostrar. Generalmente el *IconData* se puede obtener de alguna colección de iconos como *Icons*.
- Otros parámetros comunes de *Icon* son:
  - *double?* **size**: Establece el tamaño del icono.
  - *Color?* **color**: Define el color del icono.
- Si se desea obtener un icono interactivo (pulsable), utilizar un *widget* de tipo **IconButton** con el parámetro *void Function()?* **onPressed**.

Image (1)

- El *widget* **Image** se utiliza para mostrar imágenes.
- Su constructor requiere un parámetro con nombre **image** de tipo *ImageProvider* para especificar la imagen a mostrar.
- Existen diferentes tipos de *ImageProvider* como *AssetImage* al que se le pasa por parámetro un *String* con el nombre del recurso de imagen (debe estar declarado en el archivo *pubspec.yaml*), o *NetworkImage* al que se le pasa por parámetro un *String* con la *URL* de la imagen, etc.
- Si se desea mostrar una imagen temporal mientras carga la imagen principal se puede utilizar el *widget* **FadeInImage**.

Image (2)

- Cuenta con parámetros para establecer su ancho (*double?* **width**), su alto (*double? height*), su alineación (*AlignmentGeometry* **alignment**), cómo se inscribe dentro de su contenedor (*BoxFit?* **fit**), etc.
- En el caso de *FadeInImage* también existe el parámetro **placeHolder** de tipo *ImageProvider*, para especificar la imagen a mostrar temporalmente mientras carga la imagen principal.
- La clase *Image* también cuenta con constructores como *Image.asset(...)* o *Image.network(...)* para especificar posicionalmente un parámetro *String* con el nombre del recurso o la *URL* de la imagen respectivamente.

ElevatedButton

- El widget **ElevatedButton** muestra un botón.
- Su constructor requiere un parámetro con nombre **child** de tipo *Widget?* para establecer su contenido (típicamente un *Text*).
- También requiere un parámetro con nombre **onPressed** de tipo *void Function()?* para definir su comportamiento. Si se pasa un valor *null*, el botón se deshabilita.
- También existen otros tipos de botones como el **TextButton** y **OutlinedButton**, ambos con un aspecto más minimalista.

Container (1)

- El *widget* **Container** es un contenedor que permite aplicar un conjunto de propiedades y contener un elemento hijo (parámetro *Widget?* **child**), aunque éste último no es requerido.
- Entre las propiedades que se pueden aplicar se destacan entre otras: márgenes externos (*EdgeInsetGeometry?* **margin**), márgenes internos (*EdgeInsetGeometry?* **padding**), ancho (*double?* **width**), alto (*double?* **height**), alineación de su contenido (*AlignmentGeometry?* **alignment**), color de fondo (*Color?* **color**), traslación + giro + sesgo (*Matrix4?* **transform**), etc.

Container (2)

- También tiene un parámetro con nombre **decoration** de tipo *Decoration?* (si se utiliza, no se debe utilizar el parámetro *color*) al que típicamente se le pasa un *widget* **BoxDecoration** que permite definir opciones más avanzadas de decoración como bordes (*BoxBorder?* **border**), redondeado de las esquinas (*BorderRadiusGeometry?* **borderRadius**), color de fondo (*Color?* **color**), degradado de fondo (*Gradient?* **gradient**), imagen de fondo (*DecorationImage?* **image**), sombreado (*List<BoxShadow>?* **boxShadow**), forma (*BoxShape* **shape**), etc.

Container (3)

- Si se desea aplicar una animación al cambiar las propiedades del contenedor, se puede reemplazar por un widget **AnimatedContainer**.
- El constructor de *AnimatedContainer* tiene un parámetro requerido **duration** de tipo *Duration* que establece la duración de la animación.
- Además tiene un parámetro **curve** de tipo *Curve* para definir la curva de la animación (se puede obtener de la colección de curvas de la clase *Curves*).
- Existen otros *widgets* contenedores especializados en algún aspecto particular de su hijo como darle un tamaño (**SizedBox**), alinearlo o centrarlo (**Align** o **Center**), agregarle márgenes internos (**Padding**), etc.

Column

- El widget **Column** muestra sus elementos hijos en una disposición vertical o de columna.
- Sus hijos se establecen mediante el parámetro *List<Widget>* **children**.
- Otros parámetros comunes de *Column* son:
  - *MainAxisAlignment* **mainAxisAlignment**: Define la alineación vertical de los elementos hijos.
  - *CrossAxisAlignment* **crossAxisAlignment**: Define la alineación horizontal de los elementos hijos.
  - *MainAxisSize* **mainAxisSize**: Define el tamaño máximo vertical de la columna.
- Se puede utilizar el widget **Expanded** para forzar a un elemento hijo a llenar el espacio disponible.

Row

- El widget **Row** muestra sus elementos hijos en una disposición horizontal o de fila.
- Sus hijos se establecen mediante el parámetro *List<Widget>* **children**.
- Otros parámetros comunes de *Row* son:
  - *MainAxisAlignment* **mainAxisAlignment**: Define la alineación horizontal de los elementos hijos.
  - *CrossAxisAlignment* **crossAxisAlignment**: Define la alineación vertical de los elementos hijos.
  - *MainAxisSize* **mainAxisSize**: Define el tamaño máximo horizontal de la fila.
- Se puede utilizar el widget **Expanded** para forzar a un elemento hijo a llenar el espacio disponible.

Temas (1)

- Los estilos de colores y fuentes se pueden compartir a través del árbol de *widgets* mediante la definición de temas.
- Se puede definir un tema global a toda la aplicación, proveyendo un objeto **ThemeData** en el parámetro **theme** del *widget MaterialApp*.
- Si no se define un tema global, *Flutter* provee uno por defecto.
- *ThemeData* ofrece parámetros para definir el color principal (*primaryColor*), la fuente por defecto (*fontFamily*), los estilos de texto (*textTheme*), de las *AppBar* (*appBarTheme*), de los botones (*buttonTheme*), etc.

Temas (2)

- El *widget* **Theme** permite sobreescribir el tema global de la aplicación para una parte del árbol de widgets.
- Tiene dos parámetros requeridos: *ThemeData* **data** y *Widget* **child**.
- Un *ThemeData* se puede crear sin heredar de otro, o heredar de otro *ThemeData* mediante el método *copyWith(...)*:
  - *ThemeData(primarySwatch: Colors.blue, ...)*
  - *ThemeData.light().copyWith(colorScheme: ...)*
  - *Theme.of(context).copyWith(colorScheme: ...)*

Temas (3)

- La clase *ThemeData* tiene varios constructores que permiten crear el tema a partir de estilos por defecto (*.light()*, *.dark()*, etc.), o de ciertos parámetros (*from()*, *raw()*, etc.).
- Si se desea aplicar a un *widget* un estilo determinado del tema más próximo en el arbol de *widgets* se puede utilizar *Theme.of(context)*:

```
Container(
  color: Theme.of(context).hoverColor,
  child: Text('¡Hola!',
    style: Theme.of(context).textTheme.titleLarge
  ),
)
```

Mostrar Mensajes al Usuario

- Veremos a continuación un par de *widgets* que permiten mostrar mensajes al usuario, e incluso obtener algún tipo de retroalimentación como una acción que el mismo podrá tomar.
- Para mensajes rápidos se puede utilizar el *widget Snackbar*.
- Para mensajes que requieran una mayor atención del usuario veremos el *widget AlertDialog*.

SnackBar (1)

- El *widget* **SnackBar** representa un mensaje rápido al usuario que se visualiza en la parte inferior de la pantalla.
- Opcionalmente puede contener una acción que el usuario puede tomar.
- Para mostrar un *SnackBar* se utiliza un objeto **ScaffoldMessenger** que se puede obtener a partir del context, invocando al método *showSnackBar(...)*:
  - *ScaffoldMessenger.of(context).showSnackBar(...)*
- A este método se le pasa un widget *SnackBar* con la configuración deseada.

SnackBar (2)

- Al constructor de *SnackBar* se le pueden pasar los siguientes parámetros:
  - *Widget* **content**: El *widget* que se desea mostrar como contenido del *SnackBar*. Es requerido.
  - *bool?* **showCloseIcon**: Indica si se desea mostrar un icono para que el usuario pueda cerrar el *SnackBar*.
  - *Color?* **closeIconColor**: Color del icono para cerrar el *SnackBar*.
  - *Duration* **duration**: Establece el tiempo que permanecerá visible el *SnackBar* antes de desaparecer automáticamente.

SnackBar (3)

- Al constructor de *SnackBar* se le pueden pasar los siguientes parámetros (cont.):
  - *SnackBarAction* **action**: Define la acción que el usuario puede tomar. El constructor de *SnackBarAction* recibe al menos dos parámetros que son requeridos: *label* de tipo *String* para definir la etiqueta de la acción, y *onPressed* de tipo *void Function()* con el código a ejecutar.

AlertDialog (1)

- Un **AlertDialog** es un *widget* que muestra al usuario un mensaje que requiere su atención.
- Puede tener un título, un contenido, y acciones que el usuario puede tomar.
- Para mostrar un *AlertDialog* se utiliza el método *showDialog(...)* de la *library* de *material*.
- Esta función requiere que se le pase por parámetro el *BuildContext* y también una función que debe retornar un *widget* con el diálogo a mostrar (generalmente un *AlertDialog*).
- Otro parámetro interesante de *showDialog(...)* es *bool barrierDismissible*, que establece si el usuario puede cerrar el diálogo con un clic fuera del mismo.

AlertDialog (2)

- El constructor de *AlertDialog* recibe los siguientes parámetros:
  - *Widget?* **title**: El título del diálogo.
  - *Widget?* **content**: El contenido del diálogo.
  - *List<Widget>?* **actions**: Lista de *widgets* con las acciones que el usuario puede tomar (generalmente mediante algún tipo de botón).
  - *ShapeBorder?* **shape**: Establece la forma del *AlertDialog* (por ejemplo *RoundedRectangleBorder*).
- Para cerrar el *AlertDialog* si no se estableció el parámetro *barrierDismissible* en *true*, se puede utilizar el método *pop()* del *widget Navigator*.

SafeArea

- El *widget* **SafeArea** asegura que el contenido de la aplicación se muestre correctamente en dispositivos móviles, evitando áreas de recorte no deseadas en pantallas con muescas (*notch*), barras de estado, barras de navegación y otros elementos del sistema.
- Detecta los límites del sistema y ajusta el espacio seguro para acomodar los elementos superpuestos.
- Funciona en dispositivos *Android* e *iOS*, garantizando una experiencia uniforme en diferentes plataformas.
- Es fácil de implementar en la estructura de diseño de una aplicación *Flutter* existente. Sólo hay que envolver con *SafeArea* el *widget* a proteger (parámetro *child*).

MediaQuery (1)

- El *widget* **MediaQuery** proporciona información sobre el contexto de la aplicación, como la orientación de la pantalla, el tamaño de la pantalla, la densidad de píxeles, etc.
- Permite que nuestra aplicación se adapte automáticamente a diferentes tamaños de pantalla y resoluciones.
- Con el *widget MediaQuery* se puede obtener el tamaño de pantalla actual para ajustar los diseños de nuestra aplicación según el tamaño y la resolución de pantalla de los diferentes dispositivos donde se utilice.

MediaQuery (2)

- Para acceder a todas estas métricas se utiliza la clase *MediaQueryData*, que contiene información como la altura y el ancho de la pantalla, la orientación de la pantalla, la densidad de píxeles, etc.
- Para obtener el *MediaQueryData* se utiliza el método estático *of(BuildContext)* de la clase *MediaQuery*.
- Por ejemplo, para obtener el ancho y/o alto de la pantalla actual:

MediaQuery.of(context).size.width

MediaQuery.of(context).size.height

GestureDetector

- El widget **GestureDetector** es una herramienta útil en *Flutter* para detectar interacciones de usuario en la pantalla.
- Se puede usar para detectar gestos como toques, arrastres, deslizamientos, pellizcos y más.
- Puede envolver cualquier otro *widget* en su árbol de *widgets* especificándolo mediante su parámetro child, y escuchar gestos específicos en ese *widget*.
- Gracias a esto es posible por ejemplo reaccionar a un clic (mediante el parámetro *onTap* de su constructor), en widgets que en principio no lo soportan de manera predeterminada.

FutureBuilder (1)

- **FutureBuilder** es un *widget* que permite construir una interfaz de usuario dinámica y reactiva para datos asincrónicos provenientes de un *Future*.
- Esto quiere decir que tiene la capacidad de responder y adaptarse a los cambios en los datos asincrónicos que utiliza.
- Cuando estos datos asincrónicos se cargan y están disponibles, *FutureBuilder* reconstruye la interfaz de usuario para reflejar esos cambios y mostrar los datos actualizados.
- *FutureBuilder* espera un objeto *Future*, que puede contener datos que se están cargando o que aún no están disponibles.

FutureBuilder (2)

- Cuando se completa el *Future*, *FutureBuilder* reconstruye la interfaz de usuario con los datos nuevos.
- Para usar *FutureBuilder*, primero debemos crear u obtener de algún modo un objeto *Future* que contenga los datos que queremos mostrar en la interfaz de usuario.
- Luego, creamos el *FutureBuilder* que espera ese objeto *Future*.
- Dentro del *FutureBuilder*, podemos construir a través del parámetro *builder*, la interfaz de usuario utilizando los datos de dicho *Future*.

FutureBuilder (3)

- Los parámetros más comunes de *FutureBuilder* son:
  - **future**: Establece el objeto *Future* que se utilizará para construir la interfaz de usuario.
  - **builder**: Es una función que toma un contexto y un *AsyncSnapshot* y devuelve el *widget* que se utilizará para construir la interfaz de usuario. El *snapshot* permite consultar si se está a la espera de los datos (*snapshot.connectionState* == *ConnectionState.waiting*), si ya llegaron (*snapshot.hasData*), los datos (*snapshot.data*), o si hubo un error (*snapshot.hasError*).
  - **initialData**: Es opcional y proporciona datos iniciales para la interfaz de usuario mientras se espera la finalización del *Future*.

StreamBuilder (1)

- **StreamBuilder** es un *widget* que permite construir una interfaz de usuario dinámica y reactiva para datos asincrónicos provenientes de un *Stream*.
- Esto quiere decir que tiene la capacidad de responder y adaptarse a los cambios en los datos asincrónicos que utiliza.
- Un *Stream* es una secuencia de eventos asincrónicos que pueden ser transmitidos por un origen, como por ejemplo, una base de datos en tiempo real, una *API*, o cualquier otro flujo de datos asincrónico.
- El *StreamBuilder* escucha los eventos del *Stream* y se reconstruye automáticamente cada vez que un nuevo evento llega al flujo.

StreamBuilder (2)

- Para usar *StreamBuilder*, primero debemos crear u obtener de algún modo un objeto *Stream* que emita los datos que queremos mostrar en la interfaz de usuario.
- Luego, creamos el *widget StreamBuilder* que espera ese objeto *Stream*.
- Dentro del *StreamBuilder*, podemos construir, a través del parámetro *builder*, la interfaz de usuario utilizando los datos de dicho *Stream*.

StreamBuilder (3)

- Los parámetros más comunes de *StreamBuilder* son:
  - **stream**: Establece el objeto *Stream* al que se suscribirá para construir la interfaz de usuario.
  - **builder**: Es una función que toma un contexto y un *AsyncSnapshot* y devuelve el *widget* que se utilizará para construir la interfaz de usuario. El *snapshot* permite consultar si se está a la espera de los datos (*snapshot.connectionState* == *ConnectionState.waiting*), si ya llegaron (*snapshot.hasData*), los datos (*snapshot.data*), o si hubo un error (*snapshot.hasError*).
  - **initialData**: Es opcional y proporciona datos iniciales para la interfaz de usuario mientras se esperan los datos del *Stream*.