ESCUELA DE SISTEMAS Y TECNOLOGÍAS

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

TEMA: Manejo del Estado

Agenda

- Introducción
- Patrón de Elevación
- InheritedWidget
- Provider
- BLoC

Introducción (1)

- En *Flutter*, el estado se refiere generalmente a cualquier dato que cambia a lo largo del tiempo y afecta la apariencia o el comportamiento de la interfaz de usuario.
- Se distinguen tres tipos de estado en *Flutter*: el estado efímero, el estado local, y el estado global.
- El estado efímero se refiere a los datos que son temporales y no afectan la apariencia o el comportamiento de la interfaz de usuario.
- El estado local se refiere a los datos que se mantienen dentro de un *widget* y se actualizan en respuesta a las interacciones del usuario o a las notificaciones del sistema.

Introducción (2)

- Los *StatefulWidget* permiten definir y modificar el estado en el ámbito local de un *widget*, lo que los hace adecuados para manejar el estado de componentes individuales.
- El estado global se refiere a los datos que se comparten entre múltiples *widgets* y se actualizan en respuesta a las interacciones del usuario o a las notificaciones del sistema.
- Para manejar el estado global en *Flutter*, se utilizan algunos patrones o técnicas populares: el patrón de elevación, *InheritedWidget*, el patrón *Provider* o el patrón *BLoC*.

Introducción (3)

- Con el manejo adecuado del estado utilizando alguno de estos patrones, se puede crear una interfaz de usuario dinámica y receptiva.

Patrón de Elevación (1)

- El **Patrón de Elevación** implica pasar el estado como argumento a través de la jerarquía de *widgets*.
- El estado es mantenido por un *StatefulWidget* padre y es compartido con sus *widgets* hijos.
- Esto significa que el estado se define en un *widget* padre común y se pasa hacia abajo como argumentos a los *widgets* hijos que lo necesiten.
- A este concepto se lo conoce habitualmente como "*levantar el estado*".
- De esta manera, varios *widgets* pueden acceder y modificar el mismo estado, lo que facilita la sincronización y actualización de la interfaz de usuario.

Patrón de Elevación (2)

- Para permitir que los *widgets* hijos modifiquen el estado que se encuentra en el *widget* padre, se pueden pasar funciones *callback*.
- Estas funciones se ejecutan en el *widget* padre y actualizan su estado, pero son invocadas desde los *widgets* hijos.
- De esta manera, los *widgets* hijos pueden comunicarse con el *widget* padre para reflejar cambios en la interfaz de usuario.
- El patrón de elevación es una estrategia efectiva para administrar el estado en aplicaciones *Flutter*, aunque puede resultar algo engorrosa y propensa a errores si la jerarquía de hijos es muy extensa.

InheritedWidget (1)

- La clase **InheritedWidget** proporcionada por *Flutter* permite pasar datos hacia abajo a través del árbol de *widgets* de manera eficiente.
- Es especialmente útil cuando se necesita mantener el estado global (como el usuario autenticado, la configuración de la aplicación o el tema), o compartir datos entre *widgets* de manera eficiente y sin tener que pasar manualmente los datos a través de todos los niveles de la jerarquía de *widgets*.
- Evita pasar datos innecesariamente a través de múltiples *widgets* en el árbol, ya que proporciona una forma de notificar a los *widgets* que dependen de esos datos cuando se producen cambios.

InheritedWidget (2)

- Una vez creado, un *InheritedWidget* es inmutable. Esto significa que no se puede modificar directamente luego de creado. En su lugar, si los datos cambian, se crea una nueva instancia, que se propaga automáticamente hacia abajo en el árbol de *widgets*.
- Los *widgets* descendientes pueden acceder a los datos proporcionados por un *InheritedWidget* utilizando el método estático *of(...)* (o *maybeOf(...)*), lo que les permite obtener acceso directo a los datos compartidos sin necesidad de pasarlos explícitamente.

InheritedWidget (3)

- Cuando los datos en el *InheritedWidget* cambian, *Flutter* automáticamente notifica a los *widgets* que dependen de esos datos para que se redibujen y reflejen los cambios.
- Para definir un *InheritedWidget* se debe crear una clase que extienda de *InheritedWidget* y redefinir la operación *bool updateShouldNotify(InheritedWidget)*.
- Dicha operación debe devolver un valor *true* en aquellos casos en que el *InheritedWidget* deba notificar a los *widgets* dependientes.
- Dentro de la clase se codifican el o los atributos que guardarán el estado que se desea compartir.

InheritedWidget (4)

- Además debe definirse un constructor que reciba el *widget* hijo (parámetro *child*) y se lo pase al constructor de la clase base.
- Por último se codifica un método estático llamado generalmente *of(...)* o *maybeOf(...)* (si pudiera devolver *null*) que retorne la instancia del *InheritedWidget*, recibiendo por parámetro el *BuildContext*.
- Para esto se utiliza el método *dependOnInheritedWidgetOfExactType<TIPO>()* del contexto.

InheritedWidget (5)

- Luego, para acceder al *InheritedWidget* y recibir notificaciones de cambios, un *widget* debe estar dentro del subárbol que contiene el *InheritedWidget* y obtener su instancia invocando el método estático *of(...)* (o *maybeOf(...)*).
- También existe el *widget* **InheritedModel**, que ofrece una propagación de cambios más precisa y granular, lo que puede resultar en una mayor eficiencia en ciertas situaciones.
- Permite actualizar únicamente los *widgets* que dependan de cierto aspecto, que se debe indicar al obtener su instancia.

InheritedWidget (6)

- Para definirlo se debe extender de la clase *InheritedModel<TIPO>* y redefinir las operaciones *updateShouldNotify(InheritedModel)* y *updateShouldNotifyDependent(InheritedModel, Set<TIPO>)*.
- En esta última operación se pueden utilizar los aspectos contenidos en el *Set* para armar la condición de actualización.
- El aspecto se puede recibir por parámetro en el método estático *of(...)* o *maybeOf(...)* y se puede pasar por parámetro al método *dependOnInheritedWidgetOfExactType<TIPO>()* del contexto.

Provider (1)

- El patrón **Provider** implica utilizar una instancia de la clase *Provider* para compartir el estado entre múltiples *widgets* de forma segura y eficiente.
- El paquete *Flutter Provider* es una solución popular para implementar el patrón *Provider* en *Flutter*.
- Proporciona una forma eficiente y sencilla de administrar el estado de una aplicación.
- Es una alternativa a otras soluciones de gestión de estado como *Redux* o *Bloc*, pero con una sintaxis más simple y menos código repetitivo.
- *Provider* permite la inyección de dependencias en toda la jerarquía del árbol de *widgets*.

Provider (2)

- Esto significa que los datos pueden ser compartidos entre diferentes *widgets* sin necesidad de pasarlos manualmente a través de los constructores.
- *Provider* se basa en el patrón *Observer* y utiliza las clases *ChangeNotifier* y *Listenable* para notificar a los *widgets* sobre los cambios en el estado.
- Cuando un objeto *ChangeNotifier* cambia su estado, notifica a los *widgets* que están escuchando y los actualiza automáticamente.
- De esta manera se promueve un flujo de datos reactivos, lo que facilita la actualización de la interfaz de usuario en respuesta a cambios en el estado.

Provider (3)

- El *widget* **ChangeNotifierProvider** se utiliza para exponer un valor a otros *widgets*. Se coloca en la parte superior del árbol de *widgets* y se especifica qué dato o modelo se proporcionará.
- En su parámetro *create* se define una función que recibe el *BuildContext* y debe retornar la instancia del modelo que provee el estado.
- Dicho modelo se define mediante una clase que extienda de **ChangeNotifier**, y dentro de ésta se invoca al método *NotifyListeners()* cada vez que su estado cambie.

Provider (4)

- Para consumir los datos proporcionados por el *widget* *ChangeNotifierProvider*, se utiliza el *widget* **Consumer**.
- Este *widget* se coloca en el lugar donde se necesita acceder al estado, con lo que automáticamente escuchará cambios y reconstruirá los *widgets* afectados cuando estos cambios ocurran.
- Otra opción es utilizar el método estático *of<TIPO>(Context)* de la clase *Provider*, que obtiene una instancia del modelo para acceder al estado.
- Este método recibe un parámetro opcional booleano llamado *listen* para indicar si se desea o no reconstruir el *widget* al cambiar el estado del modelo.

Provider (5)

- *Provider* también ofrece el *widget* **Selector**, que permite seleccionar partes específicas del modelo o datos proporcionados para escuchar cambios.
- Esto ayuda a optimizar el rendimiento al evitar reconstrucciones innecesarias de *widgets* que no dependen de ciertos cambios en el estado.
- Cuando se necesita proporcionar varios modelos a diferentes partes de la aplicación, *Provider* ofrece el *widget* **MultiProvider** para facilitar esta tarea.
- En su parámetro *providers* se define una lista de *ChangeNotifierProvider* para proporcionar los diferentes modelos.

BLoC (1)

- El patrón **BLoC**, que significa *Business Logic Component*, implica separar la lógica de negocio de la interfaz de usuario y utilizar un *stream* para comunicar los cambios de estado.
- Es un patrón de administración de estado ampliamente utilizado en el desarrollo de aplicaciones *Flutter*.
- Proporciona una forma estructurada y escalable de separar la lógica de negocio y la gestión del estado de la interfaz de usuario en una aplicación.
- El paquete *Flutter BLoC* es una solución popular para implementar el patrón *BLoC* en *Flutter*.

BLoC (2)

- Un *BLoC* es una clase que encapsula la lógica de negocio y la gestión del estado de una parte específica de la aplicación. Puede recibir eventos desde la interfaz de usuario, procesarlos y emitir cambios de estado.
- Los eventos son las acciones que ocurren en la interfaz de usuario y que afectan al estado. Los eventos se envían al *BLoC* para que éste pueda procesarlos y tomar las decisiones adecuadas.
- Los estados representan la información actual de la parte de la aplicación que está siendo gestionada por el *BLoC*. Cada estado es inmutable y refleja una instantánea en el tiempo.

BLoC (3)

- Los *BLoCs* generalmente utilizan *Streams* para emitir cambios de estado a medida que ocurren. Los *Widgets* de la interfaz de usuario pueden suscribirse a estos *Streams* para recibir actualizaciones y reflejar el estado actual en la pantalla.
- Un **Sink** es una interfaz para agregar eventos al *BLoC*. Los eventos ingresan a través de un *Sink* y luego se procesan internamente para actualizar el estado y emitir cambios.
- Un **StreamController** es una clase que permite crear *Streams* y *Sinks* personalizados. Los *BLoCs* suelen utilizar *StreamControllers* para gestionar la comunicación entre eventos y cambios de estado.

BLoC (4)

- Dentro del *BLoC*, se puede aplicar lógica de negocio y transformaciones a los eventos antes de emitir un nuevo estado. Esto permite realizar cálculos, filtrar datos y tomar decisiones basadas en los eventos.
- Los **Cubits** son una variante simplificada de los *BLoCs* que se introdujeron en la biblioteca *bloc* de *Flutter*.
- Los *Cubits* siguen el mismo concepto de manejo de estado, pero son más ligeros y están diseñados para casos más simples.

BLoC (5)

- Flujo típico de trabajo con *BLoC* en *Flutter*:
  - **Definir eventos y estados**: Definir las clases que representarán los eventos y los estados asociados con el *BLoC*.
  - **Definir el BLoC**: Crear una clase *BLoC* que maneje la lógica de negocio y el estado de una característica específica de la aplicación.
  - **Configurar el StreamController**: Crear un *StreamController* para emitir los cambios de estado.
  - **Procesar eventos**: Dentro del *BLoC*, definir cómo manejar los eventos recibidos y cómo generar nuevos estados en función de estos eventos.

BLoC (6)

- Flujo típico de trabajo con *BLoC* en *Flutter* (cont.):
  - **Exponer el Stream de estados**: Exponer el *Stream* de estados para que los *widgets* de la interfaz de usuario puedan suscribirse y recibir actualizaciones.
  - **Consumir el estado en widgets**: En los *widgets* de la interfaz de usuario, suscribirse al *Stream* de estados (ver *widget StreamBuilder*) y actualizar la interfaz en función de los cambios de estado.
  - **Cerrar el StreamController**: Asegurarse de cerrar el *StreamController* cuando ya no sea necesario para evitar posibles fugas de memoria.

BLoC (7)

- Para definir un bloc utilizando el paquete **flutter_bloc** debe crearse una clase que extienda de *Bloc<TIPO_EVENTO, TIPO_ESTADO>*.
- En dicha clase se define un constructor que invoque al constructor de la clase base que recibe el estado inicial, y en su cuerpo se definen los manejadores de eventos con el método *on<TIPO_EVENTO>(...)*.
- Al éste método se le suministra una función manejadora del evento, que recibe por parámetro el evento manejado y un *Emitter<TIPO_ESTADO>* para emitir un nuevo estado luego de procesar el evento.
- Para enviar eventos al *bloc* se utiliza su método *add(...)*.

BLoC (8)

- Para que el *bloc* esté disponible en el árbol de *widgets* se utiliza un *widget* **BlocProvider** o **MultiBlocProvider** de forma similar a cómo se hace con el paquete *Provider*.
- Luego se puede obtener el *bloc* con el método estático *BlocProvider.of<TIPO_BLOC>(context)*.
- El *widget* **BlocBuilder<TIPO_BLOC, TIPO_ESTADO>** permite construir una interfaz de usuario con los datos del estado actual de un *bloc*, a través de su parámetro *builder*.
- Al parámetro builder se le suministra una función que recibe el *context* y el estado actual del *bloc*, y debe retornar el *widget* a mostrar.

BLoC (9)

- Para trabajar con **Cubits** la clase del *bloc* debe extender de *Cubit<TIPO_ESTADO>*.
- En dicha clase se define un constructor que invoque al constructor de la clase base que recibe el estado inicial.
- En un *Cubit* se suministran métodos para ser invocados por los consumidores, en lugar de enviar eventos con el método *add(...)*.
- Dentro de estos métodos se emite el nuevo estado con el método *emit(...)* luego de ejecutar la lógica correspondiente.