Flutter Bootcamp

State Management

Emanuel López – emanuel.lopez@globant.com

Globant

Repaso

- Contenido de un proyecto Flutter
- `pubspec.yaml`
- Dependencias y caret notation ^
- `StatelessWidget`s y `StatefulWidget`s
- Ciclo de vida de un `StatefulWidget`
- Ejemplos de widgets
- `BuildContext`
- `Key`s

Wifi

- Nombre: `GLB-Guest`
- Pass: `glo0b4ntgu3st`

Repositorio del Bootcamp: bit.ly/FlutterBootcampGlobant

Globant

State Management

A medida que avanzamos en el desarrollo de nuestra app, llegará un momento en el que necesitaremos compartir el estado de la aplicación entre 2 o mas pantallas.

Flutter es declarativo. Esto significa que crea interfaces de usuario para reflejar el estado actual de nuestra aplicación.

Cuando el estado la aplicación cambia, se desencadena una actualización de la interfaz de usuario. Cambia el estado y la interfaz de usuario se reconstruye desde cero.

Ephemeral state and App state

El estado de una aplicación es **cualquier dato que necesitemos para reconstruir tu interfaz de usuario en cualquier momento**. Dentro de esto podemos separarlos en 2 tipos conceptuales: **estado efímero** y **estado de la aplicación**.

Estado Efímero

Se refiere a los datos que pueden cambiar y actualizarse frecuentemente dentro de un widget en respuesta a diferentes eventos, pero no son persistentes en la aplicación.

Para gestionar este estado, se utilizan los `StatefulWidget`s, un widget que mantiene un estado mutable que puede cambiar durante su ciclo de vida. Cuando el estado cambia, el widget se reconstruye para reflejar los cambios en la interfaz de usuario.

Algunos ejemplos comunes de estado efímero son el texto de un cuadro de texto, el valor seleccionado en un menú desplegable o el contenido de una lista.

Ephemeral state and App state

Estado de la Aplicación

Se refiere a datos relevantes y necesarios para la aplicación en su conjunto y que deben conservarse a lo largo del tiempo, incluso cuando el usuario navega entre diferentes pantallas o cierra y vuelve a abrir la aplicación.

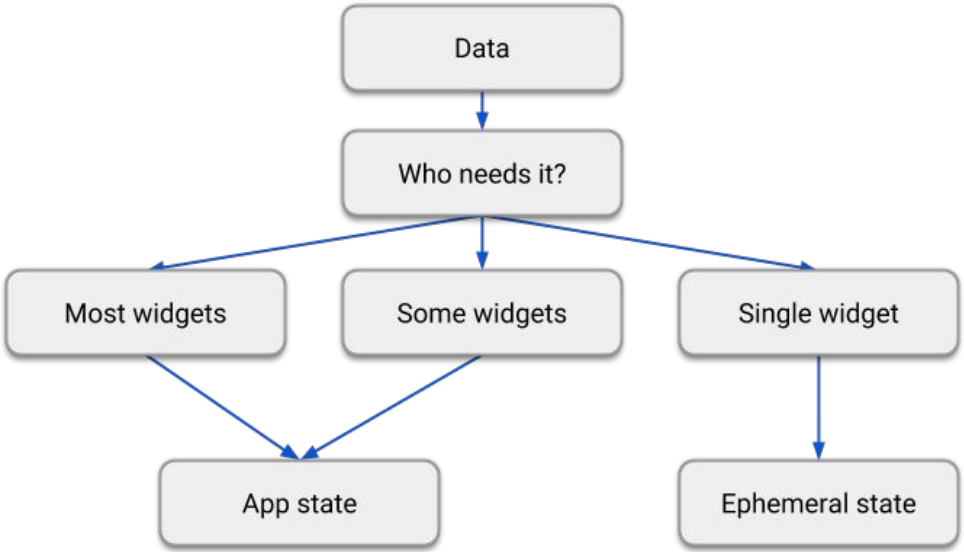
Estos datos son persistentes y pueden incluir cosas como la configuración de la aplicación, el estado de autenticación del usuario, datos almacenados en una base de datos local, etc.

Para gestionar el estado de la app en Flutter, se utilizan diferentes técnicas, como el manejo de estados a través de `Provider`s, el uso de `Bloc` para la gestión de estados basados en eventos, o incluso el almacenamiento de datos en una base de datos local o en la nube, según las necesidades de la aplicación.

Ephemeral state and App state

No existe una regla clara y universal para distinguir si una variable en particular es efímera o un estado de la aplicación. A veces, tendremos que refactorizar una en otra.

Por ese motivo, debemos tomar el siguiente diagrama "con pinzas":



Globant

InheritedWidget

Es un widget especial que permite que los datos se compartan eficientemente entre los widgets dentro de un árbol, sin necesidad de pasar manualmente esos datos a través de cada widget.

Su propósito es facilitar el acceso a datos compartidos (como configuraciones o estado) en cualquier parte del árbol de widgets sin tener que propagar parámetros a través de múltiples widgets intermedios.

Los widgets descendientes pueden acceder a los datos del `InheritedWidget` usando el método estático `of(BuildContext context)`.

Algunos ejemplos de InheritedWidget son:

- `Theme.of(context)`: Información del tema de la aplicación
- `MediaQuery.of(context)`: Información sobre las dimensiones y orientación de la pantalla
- `AppLocalizations.of(context)`: Referencia a los strings de la aplicación

Provider

Provider es **una** soluciones recomendada por el equipo de Flutter para gestionar el estado y la inyección de dependencias en nuestra aplicación.

Permite desacoplar la lógica de negocio de la interfaz de usuario, facilitando la reutilización y testing.

- **Simplicidad:** Proporciona una forma limpia y fácil de gestionar el estado sin la complejidad de otros enfoques.
- **Escalabilidad:** A medida que la app crece, el uso de Provider permite dividir el estado en partes manejables y reutilizables.
- **Inyección de dependencias:** Facilita la inyección de objetos como servicios o controladores que pueden ser reutilizados en cualquier parte del árbol de widgets.
- **Optimización de reconstrucción:** Provider solo reconstruye los widgets que dependen de los cambios, lo que mejora el rendimiento.

Patrón de diseño Observador (Observer)

El Patrón **Observador** es un patrón de diseño de software en el que un objeto (**Observable**) mantiene una lista de dependientes (**Observers**) que se suscriben para recibir notificaciones cuando el estado del sujeto cambia.

Cuando ocurre un cambio en el estado del observable, todos los observadores son notificados, y cada uno puede actualizar su comportamiento en consecuencia.

Este patrón es ideal para situaciones donde múltiples componentes necesitan reaccionar a los cambios de estado de un solo objeto sin estar fuertemente acoplados a ese objeto.

Estructura básica:

- Sujeto (Observable): El objeto cuyo estado se está monitoreando.
- Observadores (Observers): Los objetos que desean ser notificados de los cambios en el sujeto.
- Notificación: Cuando el sujeto cambia, notifica a todos los observadores registrados.

ChangeNotifierProvider y ChangeNotifier

En Flutter, `ChangeNotifier` implementa el patrón observador. Actúa como el sujeto (observable) que notifica a todos los widgets suscriptos (observers) cuando su estado cambia.

Los widgets que se suscriben al `ChangeNotifier` se actualizan automáticamente cuando este invoca **notifyListeners()**, el método encargado de notificar a los observadores.

`ChangeNotifierProvider` es uno de los componentes más usados del paquete **Provider**. Actúa como el intermediario para exponer un `ChangeNotifier` a todos los widgets descendientes que lo necesiten, permitiéndoles reaccionar automáticamente cuando el estado cambia.

StreamController y Stream

Un **Stream** es una secuencia asíncrona de eventos que puede ser escuchada. Es una herramienta fundamental en Flutter para trabajar con datos que cambian en el tiempo.

Los streams permiten transmitir datos a medida que están disponibles, y múltiples objetos pueden escucharlos, lo que sigue el principio del patrón observador.

Un **StreamController** es una clase en Flutter que actúa como el intermediario para crear y gestionar un Stream.

Sirve para:

- Proveer un Stream al que otros objetos pueden suscribirse (actuando como el sujeto en el patrón observador).
- Añadir eventos al Stream, permitiendo la emisión de nuevos datos.

StreamController y Stream

Un Stream puede tener uno o varios observadores (dependiendo si es broadcast o no), y estos observadores pueden actuar según los eventos que reciban.

- **Single-subscription Streams**: Solo permiten un observador a la vez. Son útiles cuando los datos no necesitan ser compartidos entre múltiples partes de la app.
- **Broadcast Streams**: Permiten múltiples observadores simultáneamente, lo que los hace adecuados para escenarios donde múltiples widgets o servicios necesitan estar escuchando el mismo stream.

Para evitar memory leaks de los streams, es importante cerrar los `StreamController` cuando terminemos de usarlos.

Links adicionales

- Ephemeral state and app state: docs.flutter.dev/data-and-backend/state-mgmt/ephemeral-vs-app
- Observer Design Pattern: refactoring.guru/design-patterns/observer
- List of State Management approaches: docs.flutter.dev/data-and-backend/state-mgmt/options
- Flutter Architecture Samples: fluttersamples.com
- Asynchronous programming: Streams: dart.dev/libraries/async/using-streams

Preguntas?

Globant

Gracias!

Globant