Flutter Advanced
Lec : 6

Eng. Mahmoud Elgharabawi

Agenda

- Assignment Check (Discussion).
- Flutter Advanced Concepts (ValueListenableBuilder)
- Flutter Advanced Concepts (Stateful LifeCycle)
- Leveraging Flutter Packages (get_it)
- Study Case Of Building Screens (final project).

Flutter Advanced Concepts (ValueListenableBuilder)

ValueListenableBuilder widget. It is an amazing widget. It builds the widget every time the valueListenable value changes. Its values remain synced with there listeners i.e. whenever the values change the ValueListenable listen to it. It updates the UI without using setState() or any other state management technique

ValueListenableBuilder

- Properties:

valueListenable:

builder:

child:

- These are the three properties of ValueListenableBuilder. bulder build widget depending upon the valueListenable value. valueListenable is an instance of ValueNotifier . child property is optional, it can be null if valueListenable value entirely depends upon the builder widget.

ValueListenableBuilder

- Example:

- Creating a AppValueNotifier class.

```
class AppValueNotifiier{}
```

- ValueNotifier

```
class AppValueNotifiier{
  ValueNotifier valueNotifier = ValueNotifier(0);
}
```

ValueListenableBuilder

- Creating an increment function

```
class AppValueNotifier{
  ValueNotifier valueNotifier = ValueNotifier(0);  void incrementNotifier() {
    valueNotifier.value++;
  }
}
```

Creating an object of AppValueNotifier

-
  ```
  class MyApp extends StatefulWidget {
    @override
    _MyAppState createState() => _MyAppState();
  }
  ```

ValueListenableBuilder

-
  ```
  class MyApp extends StatefulWidget {
    @override
    _MyAppState createState() => _MyAppState();
  }

  class _MyAppState extends State<MyApp> {  AppValueNotifier appValueNotifier =
  AppValueNotifier();

    @override
    Widget build(BuildContext context) {
      return Container();
    }
  }
  ```

ValueListenableBuilder

- Initializing ValueListenableBuilder

```
class MyApp extends StatefulWidget {
  @override
  _MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  AppValueNotifier appValueNotifier = AppValueNotifier();

  @override
  Widget build(BuildContext context) {
    return ValueListenableBuilder(
      valueListenable: appValueNotifier.valueNotifier,
      builder: (context, value, child) {
        return Text(value.toString());
      },
    );
  }
}
```

ValueListenableBuilder

- Using increment function

```
Scaffold(
  body: ValueListenableBuilder(
    valueListenable: appValueNotifier.valueNotifier,
    builder: (context, value, child) {
      return Text(value.toString());
    },
  ),
  floatingActionButton: FloatingActionButton(
    onPressed: () {
      appValueNotifier.incrementNotifier();
    },
  ),
);
```

- Using multiple ValueListenableBuilder, listening more than one value:

ValueListenableBuilder

- To use multiple ValueListenableBuilder , we will create the following ValueNotifier:

```
ValueNotifier incrementValueNotifier = ValueNotifier(10);
ValueNotifier decrementValueNotifier = ValueNotifier(0);
ValueNotifier colorValueNotifier = ValueNotifier(false);
ValueNotifier subtractValueNotifier = ValueNotifier(0);
```

- Creating multiple function to perform various action:

```
void decrementNotifier() {
  decrementValueNotifier.value = decrementValueNotifier.value - 3;
}

void colorNotifier() {
  colorValueNotifier.value = !colorValueNotifier.value;
}
```

ValueListenableBuilder

```
void operation() {
  subtractValueNotifier.value =
      incrementValueNotifier.value + decrementValueNotifier.value;
}

void incrementNotifier() {
  incrementValueNotifier.value++;
}
```

We have created two valueNotifier for increment and decrement, one for color, and one for subtraction operation. We will change the color of the container, the increment and decrement counters, and the substation of both the counters.

ValueListenableBuilder

- Nested ValueListenableBuilder

```
ValueListenableBuilder(
    valueListenable: appValueNotifier.incrementValueNotifier,
    builder: (context, increment, _) => ValueListenableBuilder(
        valueListenable: appValueNotifier.decrementValueNotifier,
        builder: (context, decrement, _) => ValueListenableBuilder(
            valueListenable: appValueNotifier.colorValueNotifier,
            builder: (context, color, _) => ValueListenableBuilder(
                valueListenable:
                    appValueNotifier.subtractValueNotifier,
                builder: (context, subtract, _) => Container(
                    width: 100,
                    height: 30,
                    color: color ? Colors.red : Colors.orangeAccent,
                    child: Center(
                      child:
                          Text("$increment $decrement = $subtract"),
                    )))))))
```

ValueListenableBuilder

- Changing floatingActionButton

```
floatingActionButton: FloatingActionButton(
    child: Icon(Icons.add),
    onPressed: () {
      appValueNotifier.incrementNotifier();
      appValueNotifier.decrementNotifier();
      appValueNotifier.colorNotifier();
      appValueNotifier.operation();
    }),
```

ValueListenableBuilder

- disposing all value notifier

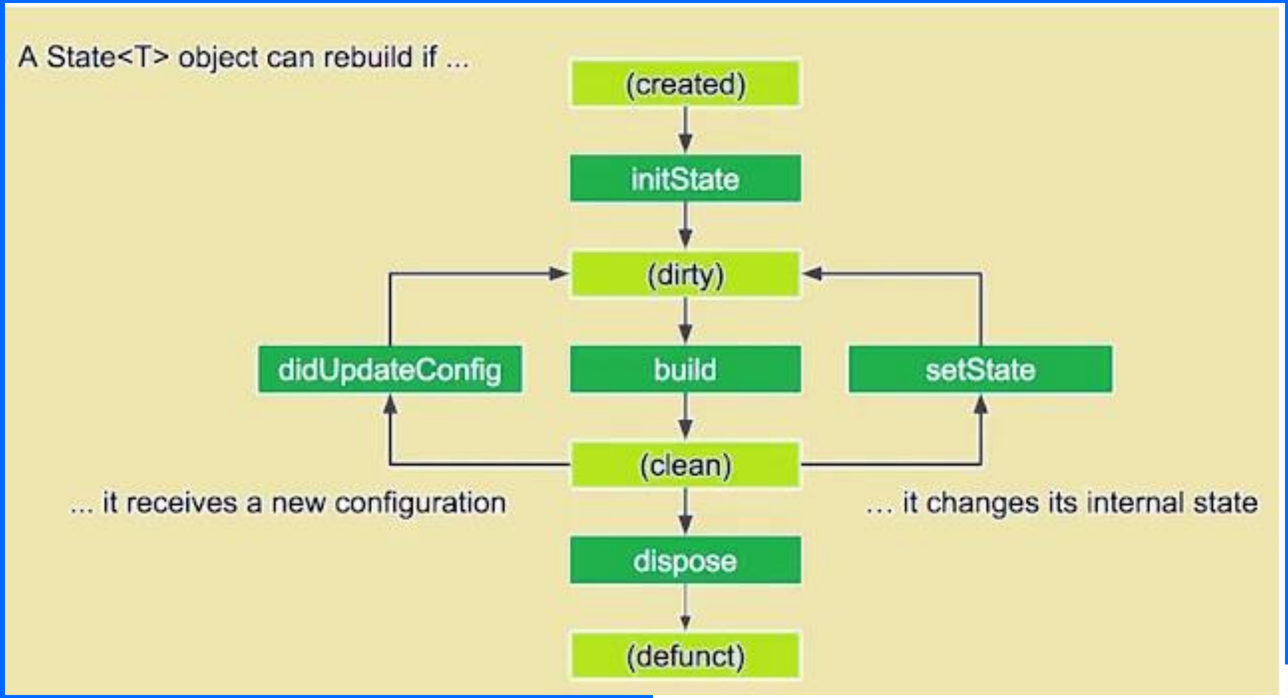
We must dispose valueNotifier as ValueNotifier is a disposable value. Also, prevent app memory loss.

-
```
@override
void dispose() {
  appValueNotifier.subtractValueNotifier.dispose();
  appValueNotifier.incrementValueNotifier.dispose();
  appValueNotifier.decrementValueNotifier.dispose();
  appValueNotifier.colorValueNotifier.dispose();
  super.dispose();
}
```

Let's CODE

Flutter Advanced Concepts (Stateful LifeCycle)

A stateful widget has the following lifecycle stages:


A State<T> object can rebuild if ...
(created)
initState
(dirty)
didUpdateConfig
build
setState
(clean)
... it receives a new configuration
... it changes its internal state
dispose
(defunct)

Stateful LifeCycle

- Widget Lifecycle Methods:

- The life cycle is based on the state and how it changes. A stateful widget has a state so we can explain the life cycle of flutter based on it. Stage of the life cycle:

- createState
- initState()
- didChangeDependencies()
- build()
- didUpdateWidget()
- setState()
- deactivate()
- dispose()

Stateful LifeCycle

- initState(): Flutter’s initState() method is the first method that is used while creating a stateful class, here we can initialize variables, data, properties, etc. for any widget.

- int a;

```
@override
void initState() {
  a= 0;
  super.initState();
}
```

Stateful LifeCycle

- createState(): When we create a stateful widget, our framework calls a createState() method and it must be overridden.

-
```
class MyPage extends StatefulWidget {
  @override
  _MyPageState createState() => _MyScreenState();
}
```

Stateful LifeCycle

- build(): The build method is used each time the widget is rebuilt. This can happen either after calling initState, didChangeDependencies, didUpdateWidget, or when the state is changed via a call to setState

-
```
@override
Widget build(BuildContext context, MyButtonState state) {
  return Container(color:Colors.red);
}
```

Stateful LifeCycle

- didChangeDependencies(): This method is called immediately after initState and when dependency of the State object changes via InheritedWidget.

-
  ```
  @override
  void didChangeDependencies() {
    super.didChangeDependencies()
  }
  ```

Stateful LifeCycle

- didUpdateWidget(): This method is called whenever the widget configuration changes. A typical case is when a parent passes some variable to the children() widget via the constructor.

-
```
@override
void didUpdateWidget(MyHomePage oldWidget) {
  super.didUpdateWidget(oldWidget)
}
```

Stateful LifeCycle

- deactivate(): It is used when the state is removed from the tree but before the current frame change can be re-inserted into another part of the tree

-
  ```
  @override
  void deactivate() {
    super.deactivate();
  }
  ```

- dispose(): We use this method when we remove permanently like should release resource created by an object like stop animation

-
  ```
  @override
  dispose() {
    animationController.dispose(); // you need this
    super.dispose();
  }
  ```

Let's CODE

Leveraging Flutter Packages (get_it)

This is a simple Service Locator for Dart and Flutter projects with some additional goodies highly inspired by Splat. It can be used instead of InheritedWidget or Provider to access objects e.g. from your UI.

Typical usage:

Accessing service objects like REST API clients or databases so that they easily can be mocked.

Accessing View/AppModels/Managers/BLoCs from Flutter Views

get_it

- Why GetIt
- As your App grows, at some point you will need to put your app's logic in classes that are separated from your Widgets. Keeping your widgets from having direct dependencies makes your code better organized and easier to test and maintain. But now you need a way to access these objects from your UI code. When I came to Flutter from the .Net world, the only way to do this was the use of InheritedWidgets. I found the way to use them by wrapping them in a StatefulWidget; quite cumbersome and have problems working consistently.
- Also:
- I missed the ability to easily switch the implementation for a mocked version without changing the UI.
- The fact that you need a BuildContext to access your objects made it inaccessible from the Business layer.

get_it

- Accessing an object from anywhere in an App can be done in other ways, but:
- If you use a Singleton you can't easily switch the implementation out for a mock version in tests
- IoC containers for Dependency Injections offer similar functionality, but with the cost of slow start-up time and less readability because you don't know where the magically injected object comes from. Most IoC libs rely on reflection they cannot be ported to Flutter.
- As I was used to using the Service Locator Splat from .Net, I decided to port it to Dart. Since then, more features have been added.

get_it

- GetIt is:
- Extremely fast (O(1))
- Easy to learn/use
- Doesn't clutter your UI tree with special Widgets to access your data like, Provider or Redux does.
- The get_it_mixin
- GetIt **isn't a state management solution**! It's a locator for your objects so you need some other way to notify your UI about changes like Streams or ValueNotifiers. But together with the get_it_mixin, it gets a full-featured easy state management solution that integrates with the Objects registered in get_it.

get_it

- Getting Started
- At your start-up you register all the objects you want to access later like this:
- final getIt = GetIt.instance;
-
- void setup() {
-   getIt.registerSingleton<AppModel>(AppModel());
-
- // Alternatively you could write it if you don't like global variables
-   GetIt.I.registerSingleton<AppModel>(AppModel());
- }
- After that you can access your AppModel class from anywhere like this:
- MaterialButton(
-   child: Text("Update"),
-   onPressed: getIt<AppModel>().update
  update

Let's CODE

Study Case Of Building Screens (final project)

Assignment

Summary

- Learn How Listeners And Notifiers
- Understand Stateful Widget Lifecycle.
- Understand The Concept singeltongBy git_it

Thank you

Eng. Mahmoud Elgharabawi