FLUTTER LEANRING NOTES

## I.Introduction

- Dart language + Flutter framework

- with Flutter, we get real native apps compiled for the target platforms. --> good performance.

- Flutter don’t compile to iOS or Android UI components. It gives you app that controls the entire screen and every pixel on it. --> ability to customize.

-

## II.Flutter Basics

This sections content:

- How Flutter app starts and works

- Working with Widgets and building custom widgets.

- Reacting to user events

- Stateless & Statefull widgets

- Dart fundamentals

Flutter app structure:

- android: the Android project which later will be built into a real Android app.

- build

- ios

- lib\_dart: it is the folder where we will add all our Dart files, Dart is the programming language Flutter uses to write the code for our Flutter application.

- test

- metadata: be managed automatically by Flutter. Flutter save some information that need for a build correctly.

- packages: auto generated by Flutter, mention some dependencies, packages needed.

- pubspec.yaml: the file manage these dependencies of the project.

Widget:

* Home: is basically the core widget which Flutter will bring onto the screen

When the entire app is mounted to the screen and here we could use text widget which is yet another widget built into Flutter and you will work a lot with these built-in widge,

**StatelessWidget and StatefulWidget**

With StatefullWidget, when the app state is changed (by setState), then the Widget will be re-render again.

IV: Widget

**Flexible**:

- Fit

- FlexFit.tight: lấy hết phần trống còn lại trên màn hình (chiều ngang hoặc dọc tùy vào olum/Row).

- FlextFit.loose:

- Flex: tương tự như weight bên Android-LinearLayout, chỉ trọng số của đối tượng so với tổng thể của layout chính.

**ListTile**: đẹp :D. Mỗi item có dạng giống như default notification item

* Leading: …Avatar
* Title
* Subtitle

**V. Responsive and Adaptive User interface**

**Responsive & Adaptive**:

+ Responsive: your user interface display as expected in different screen size.

+ Adaptive: adapting your user interface to different operating systems your app runs on.

Using the one code base for multiple platform.

**Calculate sizes Dynamically**:

- Screen size = status bar (padding) + App bar + App contents views

- MediaQuery.of(context).size

**Responsive for internal Widget** (a parent contents more than 1 widgets; the parent already set width/height, then we want to make the children responsive)

- using the LayoutBuilder, then set the children width/height by using constraints.maxHeight/constraints.maxWidth with percent.

**Only allow the specific Orientation type for the app**

**-** Before the main() function, setup the allowed orientation list

SystemChrome.setPreferredOrientations(

  //     [DeviceOrientation.portraitUp, DeviceOrientation.portraitDown]);

**Showing the different content based on the Orientation**

**-** create a variable to know which screen orientation

final isLandscape =

        MediaQuery.of(context).orientation == Orientation.landscape;

- using this variable to check into a build Widget function.

Working with the "textScaleFactor"

In this course, I mostly focus on the device sizes (height and width) when it comes to working with the MediaQuery class.

As mentioned, it offers way more than that of course. On particularly interesting property is the textScaleFactor property:

1. final curScaleFactor = MediaQuery.of(context).textScaleFactor;

textScaleFactor tells you by how much text output in the app should be scaled. Users can change this in their mobile phone / device settings.

Depending on your app, you might want to consider using this piece of information when setting font sizes.

n addition, the **following resources** might be helpful:

More on MediaQuery & Responsive Layouts: <https://api.flutter.dev/flutter/widgets/MediaQuery-class.html> & <https://stackoverflow.com/questions/49704497/how-to-make-flutter-app-responsive-according-to-different-screen-size?rq=1>

More on LayoutBuilder: <https://api.flutter.dev/flutter/widgets/LayoutBuilder-class.html>

All Cupertino Widgets: <https://flutter.dev/docs/development/ui/widgets/cupertino>

Consider this example:

1. Text('Always the same size!', style: TextStyle(fontSize: 20));

This text ALWAYS has a size of 20 device pixels, no matter what the user changed in his / her device settings.

1. Text('This changes!', style: TextStyle(fontSize: 20 \* curScaleFactor));

This text on the other hand also has a size of 20 if the user didn't change anything in the settings (because textScaleFactor by default is 1). But if changes were made, the font size of this text respects the user settings.

**Section 6: Widgets & Flutter Internal**

Widget Tree – Element Tree – Render Tree

(Container – Container Element – Render Box)

using const object to avoid rebuild some object -> improve performance

**Flutter App Lifecycle**

* Stateless Widgets: constructor 🡪 build
* Statefull Widgets:

Constructor 🡪 initState 🡪 build 🡪 setState 🡪 didUpdateWidget 🡪 dispose

**App Lifecycle**

|  |  |
| --- | --- |
| **Lifecycle State Name** | **When is it hit?** |
| Inactive | App is inactive, no user input received |
| Paused | App not visible to user, running in background |
| Resumed | App is visible and responding to user input |
| Suspending | App is about to be suspended (exited) |

**App Context**

A context is nothing else but a reference to the location of a Widget within the tree structure of all the Widgets which are built

**Key of Object**

Need to matching the Element (state) to the Widget

Section 7 – Navigation

How to setup a navigation:

* Define a route in main app:
* routes: {
* '/categories': (ctx) => CategoryMealsScreen(),
* },
* Call the defined route to make an navigation (can pass arguments)

Navigator.of(context)

        .pushNamed('/categories', arguments: {'id': id, 'title': title});

* Get the passed arguments
* final routeArgs =
* ModalRoute.of(context)!.settings.arguments as Map<String, String>;
* final categoryId = routeArgs['id'];
* final categoryTitle = routeArgs['title'];

**STATE MANAGEMENT**

**Using Provider**

We have a single list of products using entire the app. We just store the product data in Product provider.

Only the Product can change the data. Whenever the Product list change, it listener will be changed and the related widgets will be rebuilt.

* **Define a Provider:**
* class Product with ChangeNotifier
* **Init the data for the provider**
* ChangeNotifierProvider(
* create: (\_) => Products(),
* **get the data from the provider**
* final productsData = Provider.of<Products>(context);

**Provider.of và Consumer**

https://viblo.asia/p/flutter-tim-hieu-ve-state-management-voi-provider-LzD5d1wOKjY

- Provider.of: có thể set listener = false để thay đổi của dữ liệu ko làm rebuild lại widget. Mặc định nếu listener = true, mọi thay đổi của dữ liệu sẽ làm rebuild lại toàn bộ widgets liên quan.

- Consumer:

+ Cho phép lấy giá trị từ provider khi ko lấy đc BuildContext do đó ko sử dụng đc Provider.of

+ Giúp tăng hiêu năng bằng cách chỉ rebuild lại đối tượng phụ thuộc đến dữ liệu.

<https://pub.dev/documentation/provider/latest/provider/Consumer-class.html>

USER INPUT & FORM

ListView or Column

When working with Forms, you typically have multiple input fields above each other - that's why you might want to ensure that the list of inputs is scrollable. Especially, since the soft keyboard will also take up some space on the screen.

For **very long forms** (i.e. many input fields) OR in **landscape mode** (i.e. less vertical space on the screen), you might encounter a **strange behavior**: **User input might get lost if an input fields scrolls out of view.**

That happens because the ListView widget dynamically removes and re-adds widgets as they scroll out of and back into view.

For short lists/ portrait-only apps, where only minimal scrolling might be needed, a ListView should be fine, since items won't scroll that far out of view (ListView has a certain threshold until which it will keep items in memory).

But for longer lists or apps that should work in landscape mode as well - or maybe just to be safe - you might want to use a Column (combined with SingleChildScrollView) instead. Since SingleChildScrollView doesn't clear widgets that scroll out of view, you are not in danger of losing user input in that case.

*For example:*

1. Form(
2. child: ListView(
3. children: [ ... ],
4. ),
5. ),

simply becomes

1. Form(
2. child: SingleChildScrollView(
3. child: Column(
4. children: [ ... ],
5. ),
6. ),
7. ),