Adaptive

### 1. ****Controlling the Device Orientation****

Flutter cho phép bạn kiểm soát hướng của thiết bị bằng cách sử dụng SystemChrome.setPreferredOrientations. Bạn có thể giới hạn hướng dọc (portrait) hoặc ngang (landscape) bằng cách định rõ hướng mong muốn:

dart

import 'package:flutter/services.dart';

void main() {

SystemChrome.setPreferredOrientations([

DeviceOrientation.portraitUp,

DeviceOrientation.portraitDown,

]).then((\_) => runApp(MyApp()));

}

Hàm này sẽ khóa thiết bị vào các hướng đã chọn khi chạy ứng dụng.

### 2. ****Rendering Alternative Landscape Content****

Để hiển thị nội dung khác nhau dựa trên hướng của thiết bị, bạn có thể sử dụng OrientationBuilder. Widget này lắng nghe sự thay đổi của hướng thiết bị và cho phép bạn cập nhật giao diện tương ứng:

dart

OrientationBuilder(

builder: (context, orientation) {

return orientation == Orientation.portrait

? PortraitContent()

: LandscapeContent();

},

);

Ở đây, nếu thiết bị ở chế độ dọc thì hiển thị PortraitContent, và ngược lại, LandscapeContent sẽ hiển thị cho chế độ ngang.

### 3. ****Finishing Landscape Mode****

Để kết thúc chế độ landscape (ngang) và quay về chế độ portrait (dọc), bạn có thể cập nhật lại hướng mặc định bằng cách sử dụng SystemChrome.setPreferredOrientations một lần nữa.

dart

SystemChrome.setPreferredOrientations([

DeviceOrientation.portraitUp,

]);

Điều này sẽ khóa ứng dụng của bạn vào chế độ dọc và ngăn thiết bị quay lại chế độ ngang.

### 4. ****Showing Different Content Based on Device Orientation****

Flutter cung cấp MediaQuery để kiểm tra trạng thái hiện tại của thiết bị. Dựa vào thuộc tính orientation của nó, bạn có thể hiển thị nội dung khác nhau:

dart

var orientation = MediaQuery.of(context).orientation;

if (orientation == Orientation.portrait) {

return PortraitLayout();

} else {

return LandscapeLayout();

}

Việc này sẽ giúp thay đổi giao diện dựa trên hướng của thiết bị.

### 5. ****Respecting the Softkeyboard Insets****

Khi bàn phím mềm xuất hiện, giao diện của bạn có thể cần điều chỉnh để tránh bị che khuất. Bạn có thể sử dụng MediaQuery.of(context).viewInsets.bottom để biết chiều cao của bàn phím và điều chỉnh layout của mình tương ứng.

dart

Padding(

padding: EdgeInsets.only(bottom: MediaQuery.of(context).viewInsets.bottom),

child: TextField(),

)

### 6. ****Using the Device Size in Conditions****

Để kiểm tra kích thước màn hình của thiết bị, bạn có thể sử dụng MediaQuery.of(context).size. Điều này hữu ích khi bạn muốn thay đổi layout dựa trên kích thước màn hình:

dart

var size = MediaQuery.of(context).size;

if (size.width > 600) {

return TabletLayout();

} else {

return MobileLayout();

}

### 7. ****Managing the MediaQuery Object****

MediaQuery là công cụ mạnh mẽ để lấy thông tin về kích thước màn hình, mật độ điểm ảnh, và nhiều thông tin khác về thiết bị. Bạn có thể quản lý và sử dụng nó trong ứng dụng của mình để tạo ra giao diện linh hoạt.

dart

var mediaQueryData = MediaQuery.of(context);

print(mediaQueryData.size.width); // In ra chiều rộng của màn hình

### 8. ****Checking the Device Platform****

Flutter cung cấp Platform để kiểm tra nền tảng thiết bị hiện tại (Android, iOS, web, v.v.) và điều chỉnh code cho phù hợp:

dart

import 'dart:io';

if (Platform.isAndroid) {

print("This is an Android device");

} else if (Platform.isIOS) {

print("This is an iOS device");

}

### 9. ****Using the SafeArea****

SafeArea là widget dùng để đảm bảo nội dung không bị tràn ra ngoài khu vực an toàn của màn hình (như notches trên iPhone hay thanh trạng thái trên Android):

dart

SafeArea(

child: Text("Hello World"),

)

Widget này tự động thêm padding vào nội dung để đảm bảo chúng không bị che khuất bởi các phần tử hệ thống.

### 10. ****Creating Custom Adaptive Widgets****

Để tạo các widget tùy biến phù hợp với các nền tảng khác nhau, bạn có thể tạo một widget với logic để hiển thị khác nhau tùy thuộc vào thiết bị.

dart

class AdaptiveButton extends StatelessWidget {

@override

Widget build(BuildContext context) {

if (Platform.isIOS) {

return CupertinoButton(

child: Text("Press Me"),

onPressed: () {},

);

} else {

return ElevatedButton(

child: Text("Press Me"),

onPressed: () {},

);

}

}

}

Widget này sẽ hiển thị CupertinoButton cho iOS và ElevatedButton cho Android, giúp tạo ra giao diện thân thiện với người dùng trên từng nền tảng.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Widget & Flutter Internal - Deep Dive

**1. Widget Tree & Element Tree**

* **Widget Tree**: Trong Flutter, mọi thứ bạn thấy trên màn hình đều là các **widget**. Các widget được tổ chức theo cấu trúc cây, được gọi là **Widget Tree**. Mỗi widget là một mô tả (declarative) về giao diện của ứng dụng tại một thời điểm nhất định.
  + Ví dụ, một màn hình có thể có một widget cha (parent widget) chứa nhiều widget con (child widgets). Widget này không giữ trạng thái và chỉ là một mô tả giao diện.
* **Element Tree**: Khi một widget được hiển thị, Flutter sẽ tạo ra một **Element** tương ứng với mỗi widget trong widget tree. **Element Tree** là nơi chứa trạng thái của các widget và quản lý sự tương tác giữa các widget và hệ thống render. Element là cầu nối giữa **Widget Tree** và **Render Tree**.
  + Widget chỉ là mô tả tĩnh, còn **Element** là sự hiện diện sống động của widget trên giao diện, giúp theo dõi và quản lý trạng thái và render.

**2. What, Why and How?**

* **What**: Flutter là một framework UI dùng để tạo ứng dụng đa nền tảng từ một codebase duy nhất. Điểm mạnh chính của Flutter là cách nó xây dựng và render giao diện với sự tối ưu cao, đồng thời cung cấp trải nghiệm mượt mà.
* **Why**: Flutter sử dụng cách tiếp cận declarative, nghĩa là bạn chỉ cần mô tả UI của bạn dưới dạng các widget. Flutter sẽ tự động quản lý việc cập nhật giao diện khi có sự thay đổi về dữ liệu. Việc này giúp việc phát triển ứng dụng trở nên dễ dàng, nhanh chóng và trực quan hơn.
* **How**: Flutter hoạt động bằng cách sử dụng **Dart** và bộ engine của mình để vẽ trực tiếp lên giao diện, không cần cầu nối với các UI toolkit gốc của hệ điều hành. Điều này mang lại hiệu suất cao và khả năng tương thích trên nhiều nền tảng.

**3. How Flutter Rebuilds & Repaints the Screen**

* Khi một phần của UI thay đổi (thường là khi state của widget thay đổi), Flutter sẽ **rebuild** lại widget tree để cập nhật giao diện. Trong quá trình này, Flutter chỉ **rebuild** những phần nào cần thiết, không phải toàn bộ widget tree. Các widget có thể được đánh dấu để tái xây dựng thông qua việc thay đổi state hoặc khi các hàm setState() được gọi.
* **Repaint** xảy ra khi có thay đổi trực tiếp đến render của một widget (ví dụ, thay đổi màu sắc hay kích thước). Flutter chỉ repaint những widget bị ảnh hưởng, giảm thiểu việc vẽ lại không cần thiết để giữ hiệu suất tốt.
* **Rebuild** là quá trình chạy lại hàm build() để tạo ra widget tree mới, trong khi **Repaint** là quá trình render lại những phần bị thay đổi của UI.

**4. How Flutter Executes build()**

Mỗi widget trong Flutter có phương thức build(). Đây là phương thức chịu trách nhiệm trả về giao diện của widget dưới dạng các widget con khác. Khi có thay đổi về dữ liệu hoặc trạng thái, Flutter sẽ chạy lại phương thức build() để cập nhật giao diện.

* Khi build() được gọi, Flutter sẽ duyệt qua **widget tree** và cập nhật những widget con. Nó không tái tạo toàn bộ giao diện mà chỉ xây dựng lại những widget cần thiết.

dart

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: Text('Flutter Demo'),

),

body: Center(

child: Text('Hello World!'),

),

);

}

* Trong ví dụ trên, phương thức build() tạo ra một **widget tree** với cấu trúc bao gồm Scaffold, AppBar, và Text. Nếu dữ liệu của ứng dụng thay đổi, phương thức build() sẽ được gọi lại để xây dựng lại giao diện mới.

**5. Using "const" Widgets & Constructors**

Sử dụng **const** giúp Flutter cải thiện hiệu suất bằng cách tránh việc phải xây dựng lại các widget không thay đổi. Khi bạn sử dụng từ khóa **const** trước một widget, Flutter hiểu rằng widget này không thay đổi trạng thái và có thể tái sử dụng mà không cần phải rebuild.

dart

const Text('Hello, World');

* Flutter có thể tối ưu hóa bộ nhớ bằng cách giữ lại và sử dụng lại những **const widget**. Điều này đặc biệt hữu ích khi có nhiều widget tĩnh trong giao diện.

**6. Using Builder Methods**

Để cải thiện hiệu suất và tăng tính rõ ràng cho code, bạn có thể chia nhỏ giao diện phức tạp thành nhiều phương thức build() hoặc phương thức khác trong cùng một class.

* **Builder Method**: Đây là phương pháp tạo ra các widget con riêng biệt trong các phương thức build() để giữ cho code dễ đọc và bảo trì. Điều này không làm ảnh hưởng đến hiệu suất và giúp bạn quản lý code dễ hơn.

dart

Widget buildButton() {

return ElevatedButton(

onPressed: () {},

child: Text('Click Me'),

);

}

@override

Widget build(BuildContext context) {

return Scaffold(

body: Center(

child: buildButton(), // Sử dụng phương thức builder

),

);

}

* **Why use Builder Methods**:
  + Giúp chia nhỏ widget tree phức tạp thành các phần nhỏ dễ quản lý.
  + Cải thiện khả năng tái sử dụng code.
  + Giảm thiểu sự lặp lại code và tăng tính rõ ràng trong cấu trúc.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**1. Stateless & Stateful**

* **StatelessWidget**:
  + Là một widget tĩnh, không có sự thay đổi trạng thái trong suốt vòng đời của nó. Khi một widget là **stateless**, nó chỉ được xây dựng một lần và không thay đổi. Nếu cần cập nhật, toàn bộ widget sẽ được tái dựng (rebuild).
  + Thích hợp cho các giao diện đơn giản và tĩnh, chẳng hạn như hiển thị văn bản hoặc hình ảnh cố định.

dart

class MyStatelessWidget extends StatelessWidget {

@override

Widget build(BuildContext context) {

return Text('This is a stateless widget');

}

}

* **StatefulWidget**:
  + Là một widget có thể thay đổi trong suốt vòng đời của nó. Nó có thể thay đổi giao diện khi trạng thái thay đổi thông qua phương thức setState().
  + Thích hợp cho các thành phần giao diện có sự tương tác, thay đổi, ví dụ như các nút bấm, hộp kiểm, hoặc bất kỳ thứ gì có logic phức tạp hơn.

dart

class MyStatefulWidget extends StatefulWidget {

@override

\_MyStatefulWidgetState createState() => \_MyStatefulWidgetState();

}

class \_MyStatefulWidgetState extends State<MyStatefulWidget> {

String text = 'This is a stateful widget';

void \_changeText() {

setState(() {

text = 'State changed!';

});

}

@override

Widget build(BuildContext context) {

return Column(

children: [

Text(text),

ElevatedButton(

onPressed: \_changeText,

child: Text('Change Text'),

),

],

);

}

}

**2. Understanding the Widget Lifecycle**

Đối với **StatefulWidget**, widget có vòng đời (lifecycle) bao gồm các giai đoạn khác nhau từ khi được tạo ra đến khi bị hủy. Hiểu rõ vòng đời của widget giúp bạn quản lý tài nguyên và logic một cách hợp lý.

Các giai đoạn chính của vòng đời:

1. **createState()**: Được gọi khi widget được tạo ra, đây là nơi State của widget được khởi tạo.
2. **initState()**: Được gọi một lần khi widget được thêm vào widget tree. Thường được dùng để khởi tạo những biến không phụ thuộc vào build() hoặc lấy dữ liệu ban đầu.
3. **build()**: Được gọi mỗi khi cần cập nhật giao diện của widget. Đây là nơi trả về widget tree của widget.
4. **didUpdateWidget()**: Được gọi khi có sự thay đổi về widget cha và widget con cần cập nhật.
5. **setState()**: Được gọi khi bạn muốn thay đổi trạng thái và yêu cầu Flutter rebuild lại giao diện.
6. **dispose()**: Được gọi khi widget bị loại bỏ khỏi widget tree và không còn sử dụng nữa. Thường dùng để giải phóng tài nguyên như controllers, listeners, hoặc subscriptions.

dart

@override

void initState() {

super.initState();

// Khởi tạo biến hoặc logic ban đầu

}

@override

void dispose() {

// Giải phóng tài nguyên

super.dispose();

}

**3. Understanding Context**

**BuildContext** là một đối tượng trong Flutter, đại diện cho vị trí của widget trong widget tree. **Context** là một phần rất quan trọng, vì nó chứa các thông tin về môi trường của widget, bao gồm các thuộc tính như theme, locale, hoặc navigator.

* **Context** cho phép bạn truy cập tới các ancestor widgets trong widget tree. Ví dụ, bạn có thể sử dụng Theme.of(context) để lấy thông tin về chủ đề (theme) hiện tại của ứng dụng.

dart

Widget build(BuildContext context) {

final theme = Theme.of(context);

return Text(

'Hello, Flutter!',

style: theme.textTheme.headline6,

);

}

* **Context trong Navigator**: Bạn có thể sử dụng **context** để điều hướng giữa các màn hình bằng Navigator.push() hoặc Navigator.pop().

dart

Navigator.push(

context,

MaterialPageRoute(builder: (context) => SecondScreen()),

);

* Context cũng có thể khác nhau tại các điểm khác nhau trong widget tree, vì vậy nó rất quan trọng để hiểu **scope** (phạm vi) của context khi sử dụng.

**4. Keys & GlobalKeys**

**Keys** là một cơ chế mạnh mẽ trong Flutter giúp quản lý trạng thái của widget trong quá trình tái dựng (rebuild). Trong một số trường hợp, Flutter có thể tái sử dụng các widget không chính xác nếu không có **Key**. Khi đó, **Key** sẽ giúp Flutter nhận dạng đúng widget và giữ nguyên trạng thái hoặc thay đổi theo cách hợp lý.

* **Key** được sử dụng để đảm bảo rằng Flutter có thể theo dõi các widget trong widget tree, đặc biệt khi bạn đang làm việc với các widget động hoặc danh sách có thể thay đổi.

dart

ListView.builder(

key: ValueKey('my\_list'), // Gán key để Flutter nhận diện đúng danh sách

itemBuilder: (context, index) {

return ListTile(

key: ValueKey(index), // Gán key để theo dõi từng item

title: Text('Item $index'),

);

},

);

* **GlobalKey**: Là một loại key đặc biệt cho phép bạn truy cập đến **State** của một widget từ bất cứ đâu trong ứng dụng. Nó hữu ích khi bạn cần tương tác với một widget cụ thể hoặc cần truy cập state của widget đó.

dart

final GlobalKey<FormState> \_formKey = GlobalKey<FormState>();

Form(

key: \_formKey,

child: Column(

children: <Widget>[

TextFormField(),

ElevatedButton(

onPressed: () {

if (\_formKey.currentState?.validate() ?? false) {

// Xử lý form

}

},

child: Text('Submit'),

),

],

),

);

* **GlobalKey** cũng thường được sử dụng để quản lý các widgets phức tạp như Form, giúp truy cập và thao tác trực tiếp lên các widget mà không cần phải thông qua quá trình truyền dữ liệu giữa các widget con.

**Tóm tắt:**

* **StatelessWidget**: Không có sự thay đổi trạng thái, thích hợp cho giao diện tĩnh.
* **StatefulWidget**: Có thể thay đổi trạng thái, thích hợp cho giao diện động và có tương tác.
* **Widget Lifecycle**: Quản lý vòng đời widget trong Flutter giúp tối ưu hóa tài nguyên và cập nhật giao diện.
* **Context**: Đại diện cho vị trí của widget trong widget tree, giúp truy cập thông tin của ancestor widgets và điều hướng.
* **Keys & GlobalKeys**: Đảm bảo Flutter nhận diện đúng các widget trong widget tree, quản lý state khi làm việc với các danh sách động và widgets phức tạp.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Working with User Input & Forms

### 1. ****Using ScaffoldMessenger for Snackbars****

Trước đây, để hiển thị **Snackbar** trong Flutter, chúng ta thường dùng trực tiếp Scaffold bằng cách truy cập từ Scaffold.of(context). Tuy nhiên, từ phiên bản Flutter 2.0 trở đi, Flutter giới thiệu **ScaffoldMessenger**, giúp quản lý các snackbar một cách độc lập và dễ dàng hơn, đặc biệt là trong các tình huống ứng dụng có nhiều Scaffold hoặc snackbar cần xuất hiện xuyên suốt các màn hình.

* **ScaffoldMessenger** giúp quản lý trạng thái của snackbar và đảm bảo snackbar không bị mất khi thay đổi Scaffold. Đây là cách sử dụng:

dart

ScaffoldMessenger.of(context).showSnackBar(

SnackBar(

content: Text('This is a snackbar!'),

duration: Duration(seconds: 3),

),

);

Với **ScaffoldMessenger**, snackbar có thể hiển thị bất kể thay đổi nào trong widget tree mà không bị mất đi giữa các thay đổi giao diện.

### 2. ****ListView or Column, GridView****

Flutter cung cấp nhiều cách để tạo các danh sách và bố cục linh hoạt như **ListView**, **Column**, và **GridView**.

#### **ListView**:

* **ListView** là một widget linh hoạt để tạo danh sách cuộn, đặc biệt hữu ích khi bạn có nhiều phần tử và cần cuộn danh sách.
* Có hai cách sử dụng phổ biến:
  + **ListView.builder()**: Dùng để tạo danh sách động và tiết kiệm bộ nhớ bằng cách chỉ xây dựng các mục hiển thị trên màn hình.
  + **ListView.separated()**: Dùng để tạo danh sách với các phần tử ngăn cách nhau bởi một widget (ví dụ, Divider).

dart

ListView.builder(

itemCount: 50,

itemBuilder: (context, index) {

return ListTile(

title: Text('Item $index'),

);

},

);

#### **Column**:

* **Column** là widget xếp chồng các widget con theo chiều dọc, nhưng không tự động cuộn. Nếu bạn muốn cuộn, cần đặt nó trong **SingleChildScrollView**.

dart

SingleChildScrollView(

child: Column(

children: <Widget>[

Text('Item 1'),

Text('Item 2'),

// Các item khác

],

),

);

* **Column** thích hợp khi bạn có số lượng phần tử cố định và không cần tối ưu hóa việc tải các phần tử ngoài màn hình.

#### **GridView**:

* **GridView** dùng để tạo danh sách lưới, đặc biệt hữu ích cho việc hiển thị dữ liệu theo dạng ô vuông hoặc bảng.
* **GridView.builder()** là cách hiệu quả để tạo lưới cuộn với số lượng phần tử lớn.

dart

GridView.builder(

gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(

crossAxisCount: 2, // Số cột trong lưới

),

itemCount: 20,

itemBuilder: (context, index) {

return Card(

child: Center(child: Text('Item $index')),

);

},

);

### 3. ****TextInput, Forms****

#### **TextInput**:

* Flutter cung cấp widget **TextField** để nhận dữ liệu từ người dùng. Nó có thể được sử dụng đơn giản cho các input nhỏ, nhưng cũng có thể được tùy chỉnh cho các trường hợp phức tạp hơn.

dart

TextField(

decoration: InputDecoration(

labelText: 'Enter your name',

),

onChanged: (text) {

print('User entered: $text');

},

);

* **TextField** hỗ trợ nhiều tùy chọn như kiểu bàn phím (số, văn bản), số dòng, và các bộ điều khiển như **TextEditingController** để quản lý nội dung đầu vào.

dart

final myController = TextEditingController();

TextField(

controller: myController,

decoration: InputDecoration(

labelText: 'Enter text',

),

);

RaisedButton(

onPressed: () {

print('Text entered: ${myController.text}');

},

child: Text('Submit'),

);

#### **Forms**:

* **Form** là một container giúp quản lý nhiều input liên quan đến nhau (như một mẫu đăng ký hoặc đăng nhập). Nó cung cấp cơ chế để xác thực các trường input và theo dõi trạng thái của form.

dart

final \_formKey = GlobalKey<FormState>();

Form(

key: \_formKey,

child: Column(

children: <Widget>[

TextFormField(

decoration: InputDecoration(labelText: 'Enter your name'),

validator: (value) {

if (value == null || value.isEmpty) {

return 'Please enter some text';

}

return null;

},

),

ElevatedButton(

onPressed: () {

if (\_formKey.currentState?.validate() ?? false) {

// Nếu form hợp lệ, xử lý dữ liệu ở đây

print('Form submitted successfully');

}

},

child: Text('Submit'),

),

],

),

);

* **FormState** giúp bạn truy cập trạng thái của form và các trường input. Bạn có thể dùng **validator** để xác thực các trường trước khi form được gửi đi.

### Tóm tắt:

* **ScaffoldMessenger**: Quản lý snackbars linh hoạt và dễ dàng hơn so với Scaffold.of.
* **ListView/Column/GridView**: Các widget cung cấp cách xếp chồng và hiển thị danh sách dữ liệu, với **ListView** và **GridView** thích hợp cho danh sách dài, còn **Column** thích hợp cho số lượng nhỏ hơn và không cần cuộn.
* **TextInput, Forms**: **TextField** là widget cơ bản để nhập văn bản, trong khi **Form** cung cấp công cụ để quản lý và xác thực các input phức tạp.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Navigation & Route

### 1. ****Navigating to a New Page****

Điều hướng giữa các trang (màn hình) là một trong những tác vụ phổ biến trong ứng dụng. Flutter sử dụng **Navigator** để quản lý điều hướng. Bạn có thể sử dụng phương thức Navigator.push() để chuyển sang trang mới:

dart

Navigator.push(

context,

MaterialPageRoute(builder: (context) => SecondPage()),

);

* **Navigator** quản lý một stack (ngăn xếp) các màn hình. Khi gọi push(), một màn hình mới sẽ được thêm vào stack.

### 2. ****Passing Data via the Constructor****

Khi điều hướng tới một trang mới, bạn có thể truyền dữ liệu thông qua **constructor** của widget. Ví dụ, nếu bạn có một màn hình chi tiết và muốn truyền thông tin từ trang trước, bạn có thể làm như sau:

dart

// Màn hình chi tiết với constructor nhận dữ liệu

class DetailScreen extends StatelessWidget {

final String data;

DetailScreen({required this.data});

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: Text('Detail Screen')),

body: Center(

child: Text('Data passed: $data'),

),

);

}

}

// Điều hướng và truyền dữ liệu

Navigator.push(

context,

MaterialPageRoute(

builder: (context) => DetailScreen(data: 'Hello from first page'),

),

);

### 3. ****Using Named Routes & Passing Data With Named Routes****

**Named Routes** cho phép bạn đặt tên cho các trang và điều hướng tới chúng bằng cách sử dụng tên đó thay vì tạo một MaterialPageRoute mới. Điều này rất hữu ích trong các ứng dụng lớn khi có nhiều màn hình.

1. Định nghĩa các named routes trong MaterialApp:

dart

MaterialApp(

initialRoute: '/',

routes: {

'/': (context) => FirstPage(),

'/second': (context) => SecondPage(),

},

);

1. Điều hướng đến một trang bằng **named route**:

dart

Navigator.pushNamed(context, '/second');

1. Truyền dữ liệu khi điều hướng qua named routes:

dart

Navigator.pushNamed(

context,

'/second',

arguments: 'Data passed via Named Route',

);

* Để truy cập dữ liệu ở trang đích, sử dụng ModalRoute.of(context) để lấy **arguments**:

dart

class SecondPage extends StatelessWidget {

@override

Widget build(BuildContext context) {

final String data = ModalRoute.of(context)!.settings.arguments as String;

return Scaffold(

body: Center(

child: Text('Received data: $data'),

),

);

}

}

### 4. ****Diving Deeper into Named Routes****

**Named Routes** cung cấp sự linh hoạt và tiện lợi khi bạn có nhiều màn hình cần điều hướng tới. Dưới đây là một số mẹo nâng cao:

* **initialRoute**: Đặt route đầu tiên khi ứng dụng được khởi chạy.
* **arguments**: Truyền nhiều kiểu dữ liệu hơn (ví dụ: object, map) thay vì chỉ string.
* **popUntil()**: Quay lại nhiều màn hình một cách linh hoạt, dựa trên điều kiện của route.

dart

Navigator.popUntil(context, ModalRoute.withName('/home'));

### 5. ****onGenerateRoute & onUnknownRoute****

Trong các ứng dụng phức tạp hơn, bạn có thể sử dụng onGenerateRoute để kiểm soát cách ứng dụng xử lý các route một cách linh hoạt hơn. Điều này cho phép bạn tạo route một cách động dựa trên thông tin được cung cấp.

* **onGenerateRoute**: Xử lý route dựa trên logic tùy chỉnh.

dart

MaterialApp(

onGenerateRoute: (settings) {

if (settings.name == '/second') {

return MaterialPageRoute(

builder: (context) => SecondPage(data: settings.arguments),

);

}

return null; // Trả về null nếu không tìm thấy route

},

);

* **onUnknownRoute**: Được gọi khi không tìm thấy route nào phù hợp (giống như trang 404 trên web).

dart

MaterialApp(

onUnknownRoute: (settings) {

return MaterialPageRoute(

builder: (context) => UnknownPage(),

);

},

);

### 6. ****Tabs & Drawers!****

#### **Tabs**:

* Flutter cung cấp **TabBar** và **TabBarView** để tạo các tab trong ứng dụng. Các tab giúp tổ chức nội dung theo dạng phân nhóm, dễ dàng chuyển đổi giữa các phần khác nhau của ứng dụng.

dart

DefaultTabController(

length: 3,

child: Scaffold(

appBar: AppBar(

title: Text('Tabs Demo'),

bottom: TabBar(

tabs: [

Tab(text: 'Tab 1'),

Tab(text: 'Tab 2'),

Tab(text: 'Tab 3'),

],

),

),

body: TabBarView(

children: [

Center(child: Text('Content for Tab 1')),

Center(child: Text('Content for Tab 2')),

Center(child: Text('Content for Tab 3')),

],

),

),

);

#### **Drawers**:

* **Drawer** là một thành phần phổ biến giúp tạo thanh điều hướng bên hông trong ứng dụng, chứa các liên kết hoặc điều hướng đến các phần khác nhau.

dart

Scaffold(

appBar: AppBar(title: Text('Drawer Demo')),

drawer: Drawer(

child: ListView(

children: <Widget>[

DrawerHeader(

child: Text('Header'),

decoration: BoxDecoration(color: Colors.blue),

),

ListTile(

title: Text('Item 1'),

onTap: () {

Navigator.pop(context); // Đóng Drawer

},

),

ListTile(

title: Text('Item 2'),

onTap: () {

Navigator.pop(context);

},

),

],

),

),

body: Center(child: Text('Home Page')),

);

### 7. ****Replacing, Popping & Data****

#### **Replacing (Navigator.pushReplacement)**:

Khi bạn muốn chuyển sang một trang mới và **thay thế** trang hiện tại, bạn có thể sử dụng Navigator.pushReplacement(). Điều này sẽ thay thế màn hình hiện tại bằng một màn hình mới và loại bỏ màn hình cũ khỏi ngăn xếp.

dart

Navigator.pushReplacement(

context,

MaterialPageRoute(builder: (context) => NewPage()),

);

#### **Popping (Navigator.pop)**:

Để quay lại trang trước, bạn có thể sử dụng Navigator.pop(). Phương thức này sẽ xóa màn hình hiện tại khỏi ngăn xếp và quay về màn hình trước đó.

dart

Navigator.pop(context);

#### **Passing Data Back When Popping**:

Bạn có thể truyền dữ liệu trở lại màn hình trước khi sử dụng Navigator.pop(). Điều này rất hữu ích khi bạn muốn truyền kết quả từ một trang "chi tiết" trở lại trang "danh sách".

1. Chuyển đến màn hình và chờ nhận dữ liệu:

dart

final result = await Navigator.push(

context,

MaterialPageRoute(builder: (context) => SecondPage()),

);

// Sử dụng kết quả

print('Received result: $result');

1. Trả về dữ liệu từ trang chi tiết:

dart

Navigator.pop(context, 'This is the result');

### Tóm tắt:

* **Navigator.push()** điều hướng tới trang mới và bạn có thể truyền dữ liệu thông qua constructor hoặc arguments.
* **Named Routes** giúp quản lý và điều hướng trang tiện lợi, đặc biệt là trong các ứng dụng lớn.
* **onGenerateRoute** và **onUnknownRoute** cho phép tùy chỉnh logic điều hướng và xử lý khi không tìm thấy route.
* **Tabs** và **Drawers** giúp cải thiện trải nghiệm người dùng qua cách tổ chức nội dung ứng dụng.
* **pushReplacement** và **pop** giúp điều khiển luồng điều hướng của ứng dụng, bao gồm việc truyền và nhận dữ liệu giữa các trang.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

State Management

### 1. ****State, State Management & The Provider Package/Approach****

**State** trong Flutter biểu thị dữ liệu thay đổi theo thời gian của ứng dụng và **State Management** (quản lý trạng thái) giúp điều phối các phần của ứng dụng phản hồi lại những thay đổi này.

Flutter cung cấp nhiều phương thức quản lý trạng thái, từ **setState** (phù hợp cho các ứng dụng nhỏ) đến các giải pháp mạnh mẽ hơn như **Provider**. **Provider** là một gói phổ biến để quản lý trạng thái toàn cục trong ứng dụng, cung cấp cách tiếp cận đơn giản và hiệu quả để quản lý dữ liệu được chia sẻ giữa nhiều widget.

* **Provider** quản lý việc cung cấp các đối tượng và dữ liệu cho các widget con mà không cần truyền dữ liệu qua nhiều lớp.
* Dưới đây là một ví dụ về cách sử dụng Provider:

dart

class Counter with ChangeNotifier {

int \_count = 0;

int get count => \_count;

void increment() {

\_count++;

notifyListeners(); // Thông báo cho các widget khác về sự thay đổi

}

}

1. Khởi tạo **Provider** ở cấp cao trong widget tree:

dart

void main() {

runApp(

ChangeNotifierProvider(

create: (context) => Counter(),

child: MyApp(),

),

);

}

1. Truy cập trạng thái và lắng nghe thay đổi trong các widget con:

dart

class CounterDisplay extends StatelessWidget {

@override

Widget build(BuildContext context) {

final counter = Provider.of<Counter>(context); // Lấy dữ liệu từ Provider

return Text('Counter: ${counter.count}');

}

}

* Khi trạng thái thay đổi, mọi widget sử dụng **Provider** sẽ tự động được xây dựng lại.

### 2. ****Using Nested Models & Providers****

Trong các ứng dụng lớn, bạn thường cần sử dụng **Nested Providers** (nhiều lớp Provider) để quản lý các mô hình phức tạp hơn.

* Khi một mô hình phức tạp chứa nhiều mô hình con, bạn có thể lồng các **Provider** để quản lý các phần khác nhau của trạng thái.

Ví dụ: bạn có một ứng dụng với hai mô hình User và Settings. Bạn có thể lồng hai Provider để quản lý cả hai mô hình này.

dart

void main() {

runApp(

MultiProvider(

providers: [

ChangeNotifierProvider(create: (context) => UserModel()),

ChangeNotifierProvider(create: (context) => SettingsModel()),

],

child: MyApp(),

),

);

}

* Bên trong các widget con, bạn có thể truy cập một hoặc cả hai mô hình:

dart

class UserInfo extends StatelessWidget {

@override

Widget build(BuildContext context) {

final user = Provider.of<UserModel>(context);

final settings = Provider.of<SettingsModel>(context);

return Column(

children: [

Text('User: ${user.name}'),

Text('Theme: ${settings.theme}'),

],

);

}

}

### 3. ****Exploring Alternative Provider Syntaxes****

**Provider** hỗ trợ nhiều cách truy cập dữ liệu khác nhau trong các widget con. Bạn có thể sử dụng Provider.of, Consumer, hoặc Selector tùy thuộc vào nhu cầu cụ thể.

* **Provider.of**: Truy cập dữ liệu một lần và nhận thông báo về mọi thay đổi.

dart

final counter = Provider.of<Counter>(context);

* **Consumer**: Một widget giúp xây dựng lại giao diện chỉ khi một phần cụ thể của trạng thái thay đổi. **Consumer** tránh việc xây dựng lại toàn bộ widget tree.

dart

Consumer<Counter>(

builder: (context, counter, child) {

return Text('Counter: ${counter.count}');

},

);

* **Selector**: Tối ưu hóa hiệu suất bằng cách chỉ lắng nghe các thay đổi cụ thể của mô hình, bỏ qua các thay đổi không cần thiết.

dart

Selector<Counter, int>(

selector: (context, counter) => counter.count,

builder: (context, count, child) {

return Text('Selected Count: $count');

},

);

### 4. ****Using "Consumer" instead of "Provider.of"****

**Consumer** là cách tiếp cận tốt hơn so với Provider.of khi bạn muốn tránh xây dựng lại toàn bộ widget con mỗi khi có thay đổi. **Consumer** chỉ xây dựng lại widget bên trong nó khi trạng thái liên quan thay đổi.

#### So sánh giữa Provider.of và Consumer:

* **Provider.of**: Dữ liệu sẽ được lấy ngay khi widget được xây dựng. Widget sẽ tự động được xây dựng lại khi có bất kỳ thay đổi nào của trạng thái mà nó phụ thuộc.
* **Consumer**: Cho phép tách riêng phần logic lấy dữ liệu và phần giao diện. Nó cũng giúp tránh việc xây dựng lại các phần không liên quan trong widget tree.

Ví dụ:

dart

Consumer<Counter>(

builder: (context, counter, child) {

return Text('Counter: ${counter.count}');

},

);

Trong ví dụ trên, chỉ **Consumer** sẽ được xây dựng lại khi trạng thái của **Counter** thay đổi, giúp tối ưu hóa hiệu suất so với việc sử dụng Provider.of để lắng nghe và xây dựng lại cả widget con.

### Tóm tắt:

* **Provider** là một giải pháp mạnh mẽ để quản lý trạng thái trong ứng dụng Flutter, giúp chia sẻ dữ liệu một cách hiệu quả giữa các widget.
* **Nested Providers** hỗ trợ quản lý nhiều mô hình phức tạp cùng lúc.
* **Consumer** và **Selector** cung cấp các cách tiếp cận linh hoạt để tối ưu hóa việc xây dựng lại giao diện, giúp cải thiện hiệu suất.
* **Consumer** là một phương pháp linh hoạt và tối ưu hơn **Provider.of** khi muốn kiểm soát chính xác các phần của widget tree được xây dựng lại.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

### 1. ****Using Provider, MultiProvider, ProxyProvider, ListenableProvider****

#### **Provider**:

**Provider** là phương pháp quản lý trạng thái cơ bản nhất. Nó cung cấp một đối tượng hoặc dữ liệu cho các widget con trong cây widget.

* **Cách sử dụng đơn giản** của **Provider**:

dart

Copy code

Provider<String>(

create: (context) => 'Hello, Provider!',

child: MyWidget(),

);

* Truy cập giá trị từ **Provider**:

dart

Copy code

final String value = Provider.of<String>(context);

#### **MultiProvider**:

**MultiProvider** giúp bạn khai báo nhiều **Provider** cùng một lúc, giúp mã dễ quản lý hơn, đặc biệt trong các ứng dụng phức tạp.

dart

Copy code

MultiProvider(

providers: [

Provider<int>(create: (context) => 42),

ChangeNotifierProvider(create: (context) => MyModel()),

],

child: MyApp(),

);

#### **ProxyProvider**:

**ProxyProvider** cho phép bạn tạo một **Provider** dựa trên dữ liệu của các **Provider** khác. Điều này rất hữu ích khi một mô hình hoặc dịch vụ phụ thuộc vào dữ liệu từ một mô hình khác.

dart

Copy code

ProxyProvider<UserModel, SettingsModel>(

update: (context, user, previousSettings) => SettingsModel(user),

child: MyApp(),

);

* Ở đây, SettingsModel dựa trên UserModel, và nó sẽ được cập nhật khi UserModel thay đổi.

#### **ListenableProvider**:

**ListenableProvider** cho phép bạn cung cấp một đối tượng mà thực hiện giao diện Listenable, chẳng hạn như AnimationController hoặc ChangeNotifier.

dart

Copy code

ListenableProvider<AnimationController>(

create: (context) => AnimationController(vsync: tickerProvider),

child: MyApp(),

);

### 2. ****Using ChangeNotifierProvider, ValueListenableProvider, StreamProvider, FutureProvider****

#### **ChangeNotifierProvider**:

**ChangeNotifierProvider** cung cấp một đối tượng **ChangeNotifier**, giúp dễ dàng lắng nghe và phản hồi các thay đổi trạng thái.

dart

Copy code

class MyModel extends ChangeNotifier {

String value = "Initial value";

void updateValue(String newValue) {

value = newValue;

notifyListeners(); // Thông báo cho widget con về sự thay đổi

}

}

Sử dụng **ChangeNotifierProvider** để cung cấp MyModel:

dart

Copy code

ChangeNotifierProvider(

create: (context) => MyModel(),

child: MyApp(),

);

Truy cập trạng thái và lắng nghe các thay đổi:

dart

Copy code

Consumer<MyModel>(

builder: (context, model, child) {

return Text('Current value: ${model.value}');

},

);

#### **ValueListenableProvider**:

**ValueListenableProvider** cung cấp một **ValueListenable**, thường được sử dụng với các widget như TextEditingController, AnimationController, hoặc bất kỳ đối tượng nào triển khai ValueListenable.

dart

Copy code

ValueNotifier<int> counter = ValueNotifier<int>(0);

ValueListenableProvider<int>(

create: (context) => counter,

child: MyApp(),

);

* Truy cập dữ liệu từ ValueNotifier:

dart

Copy code

Consumer<int>(

builder: (context, value, child) {

return Text('Counter: $value');

},

);

#### **StreamProvider**:

**StreamProvider** cho phép bạn cung cấp dữ liệu từ một **Stream**, lý tưởng cho các trường hợp dữ liệu thời gian thực như từ Firebase hoặc WebSockets.

dart

Copy code

Stream<int> myNumberStream() async\* {

await Future.delayed(Duration(seconds: 2));

yield 42;

}

StreamProvider<int>(

create: (context) => myNumberStream(),

initialData: 0,

child: MyApp(),

);

* Truy cập dữ liệu từ **Stream**:

dart

Copy code

Consumer<int>(

builder: (context, number, child) {

return Text('Stream number: $number');

},

);

#### **FutureProvider**:

**FutureProvider** cung cấp dữ liệu từ một **Future**, rất hữu ích khi bạn cần tải dữ liệu không đồng bộ như từ API hoặc cơ sở dữ liệu.

dart

Copy code

Future<String> fetchData() async {

await Future.delayed(Duration(seconds: 2));

return 'Fetched Data';

}

FutureProvider<String>(

create: (context) => fetchData(),

initialData: 'Loading...',

child: MyApp(),

);

* Truy cập dữ liệu từ **Future**:

dart

Copy code

Consumer<String>(

builder: (context, data, child) {

return Text('Data: $data');

},

);

### 3. ****Tóm tắt****

* **Provider** cung cấp dữ liệu một cách đơn giản giữa các widget.
* **MultiProvider** giúp khai báo nhiều **Provider** dễ dàng hơn trong cùng một cây widget.
* **ProxyProvider** sử dụng dữ liệu từ một hoặc nhiều **Provider** khác để tạo ra dữ liệu mới.
* **ListenableProvider** cung cấp các đối tượng có thể lắng nghe như AnimationController hoặc ChangeNotifier.
* **ChangeNotifierProvider** là một phương pháp quản lý trạng thái phổ biến và dễ sử dụng với ChangeNotifier.
* **ValueListenableProvider** lắng nghe và cung cấp giá trị từ một đối tượng ValueListenable.
* **StreamProvider** lắng nghe dữ liệu thời gian thực từ một **Stream**.
* **FutureProvider** cung cấp dữ liệu từ một **Future**, lý tưởng cho các tác vụ không đồng bộ.

Những phương thức này cung cấp sự linh hoạt và hiệu quả trong quản lý trạng thái cho các ứng dụng Flutter, từ các trường hợp đơn giản đến phức tạp.