



Chương 6

Nguyên lý phát triển ứng dụng với Flutter

Thuật ngữ

Bridge	Cho phép giao tiếp của hai thành phần, theo truyền thống sẽ không thể giao tiếp
Canvas	Một thành phần của một framework di động nhất định chịu trách nhiệm hiển thị pixel trên màn hình của thiết bị di động (Quartz 2D trong IOS)
Component	"Một component là một phần không tầm thường, gần như độc lập và có thể thay thế của một hệ thống đáp ứng một chức năng rõ ràng trong bối cảnh của một kiến trúc được xác định rõ". Thường được gọi là mô-đun.
Interface	Một "interface" là một ranh giới được chia sẻ trong đó hai hoặc nhiều thành phần riêng biệt của hệ thống máy tính trao đổi thông tin.
Localization	Một ứng dụng được “bản địa hóa” cho một quốc gia nhất định nếu tất cả các văn bản trong ứng dụng sẽ được viết bằng ngôn ngữ mẹ đẻ của quốc gia đó.

Thuật ngữ

Native	Một ứng dụng được coi là "native" khi nó được phát triển bằng ngôn ngữ dành riêng cho nền tảng và chỉ có thể chạy trên các thiết bị di động hoạt động theo nền tảng đó.
Package	Một tập hợp các lớp phần mềm được đóng gói cùng nhau. Chúng có thể được thêm vào một chương trình và cung cấp một số loại chức năng bổ sung cho chương trình đó.
Side Effect	Bất kỳ sửa đổi nào mà một hàm nhất định thực hiện đối với chương trình nằm ngoài phạm vi cục bộ của hàm.
State	Bất kỳ dữ liệu nào trong ứng dụng có thể thay đổi theo thời gian
User Interface (UI)	Bất kỳ thành phần nào của hệ thống tương tác (phần mềm hoặc phần cứng) cung cấp thông tin và điều khiển cần thiết để người dùng thực hiện một tác vụ công việc cụ thể với hệ thống tương tác
Widget	Thành phần trực quan (hoặc một thành phần tương tác với các khía cạnh trực quan) của một ứng dụng.

Mục lục

1. Giới thiệu
2. Mô tả sản phẩm
3. Flutter
4. Chiến lược quyết định
5. Tổng kết

1. Giới thiệu



1. Giới thiệu

Nếu nhóm phát triển chọn xây dựng ứng dụng của họ "native" đối với nhiều nền tảng, họ sẽ cần duy trì nhiều codebase => Chi phí tăng

Các công nghệ đa nền tảng hứa hẹn sẽ giảm thiểu chi phí đó bằng cách sử dụng một codebase để hỗ trợ nhiều nền tảng.

Đánh đổi: Ít bảo trì hơn và chi phí phát triển ít hơn cho các ứng dụng nhưng kém hiệu suất và kém ổn định hơn.

Năm 2018, Google đã phát hành "Flutter" với hứa hẹn sẽ đem lại những lợi thế của một giải pháp đa nền tảng trong khi vẫn duy trì hiệu năng.

Các công ty lớn như BMW đang chuyển sang dùng Flutter

Flutter hiện là kỹ năng phát triển nhanh nhất trong số các kỹ sư phần mềm trên "LinkedIn".

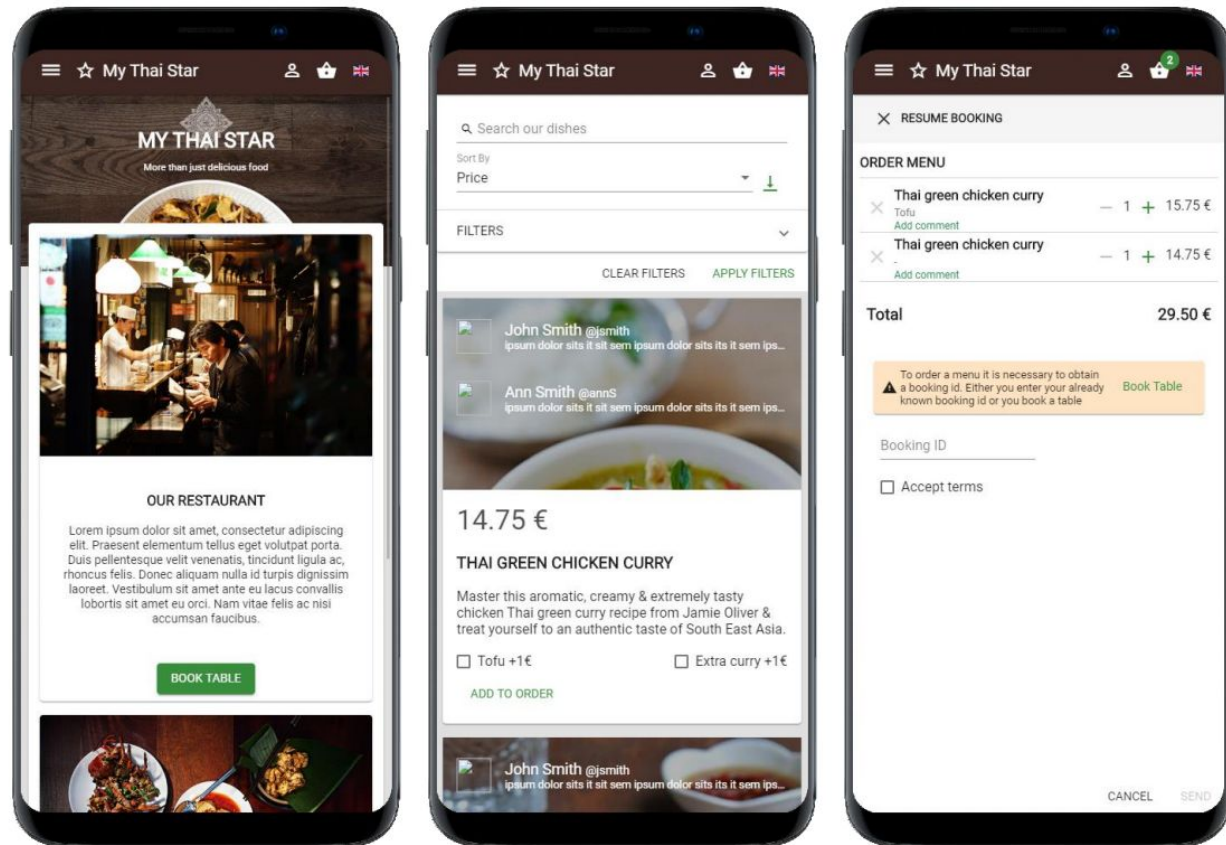
2. Mô tả sản phẩm



2. Mô tả sản phẩm

2.1 Triển khai

Hình dưới đây cho thấy ảnh chụp màn hình của ứng dụng phiên bản đầu tiên.



2. Mô tả sản phẩm

2.2. Domain

Bảng sau liệt kê tất cả các tính năng đầy đủ của ứng dụng My Thai Star :

Tính năng	Mô tả
Digital Menu	Hiển thị danh sách các món ăn mà nhà hàng cung cấp. Danh sách này có thể tìm kiếm và sắp xếp được.
Book a Table	Cung cấp tùy chọn đặt chỗ cho một bàn vào một ngày cụ thể
Invite a Friend	Cung cấp tùy chọn để mời ai đó tham gia vào phòng
Order Food	Cung cấp tùy chọn để đặt một các món ăn cho bàn đã được đặt trước
Waiter Cockpit	Cung cấp cho nhân viên tùy chọn để xem tất cả các bàn đã đặt và đặt hàng.
Localization	Chuyển đổi ngôn ngữ sử dụng

2. Mô tả sản phẩm

2.3 Thành phần

Thành phần Mô tả

Phía giao diện	Chịu trách nhiệm xác thực đầu vào của người dùng, bản địa hóa văn bản, xử lý trạng thái của đơn hàng hiện tại và giao tiếp với back-end thông qua các cuộc gọi HTTP.
Back-end	<p>Quản lý món ăn Cung cấp danh sách các món ăn có thể tìm kiếm. Các món ăn được tải từ Database.</p> <p>Quản lý đặt bàn Sẽ xác thực và lưu một đặt bàn đã nhận và tạo mã thông báo đặt chỗ. Nhân viên được xác thực có thể tìm kiếm thông qua các đặt bàn đã lưu. Đặt chỗ được lưu trong Database.</p> <p>Quản lý đơn hàng Sẽ thêm đơn hàng vào đặt bàn thông qua mã đặt bàn được cung cấp. Đơn đặt hàng được lưu trong Database.</p>

Database Lưu trữ dữ liệu

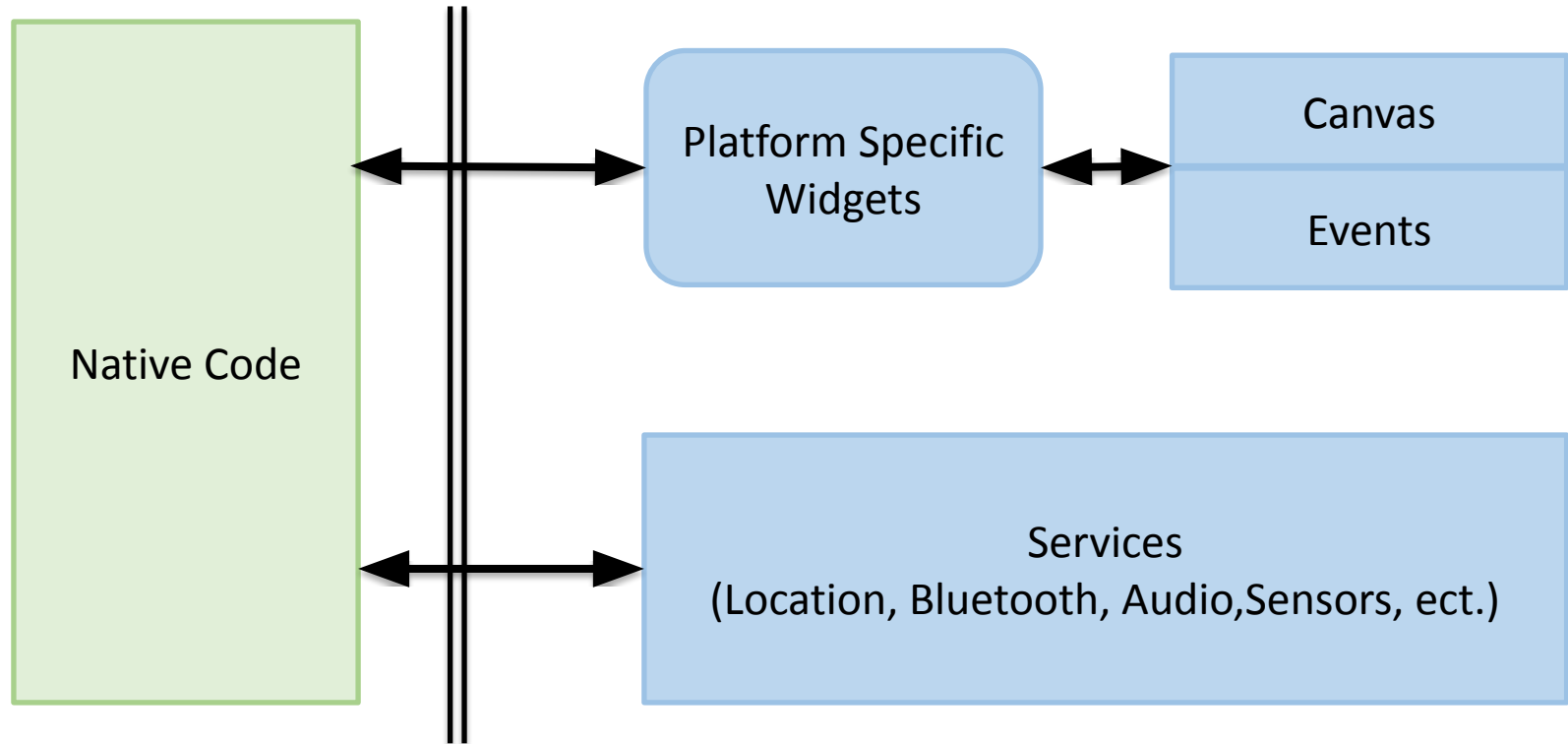
3. Framework Flutter



3 Framework Flutter

3.1 So sánh

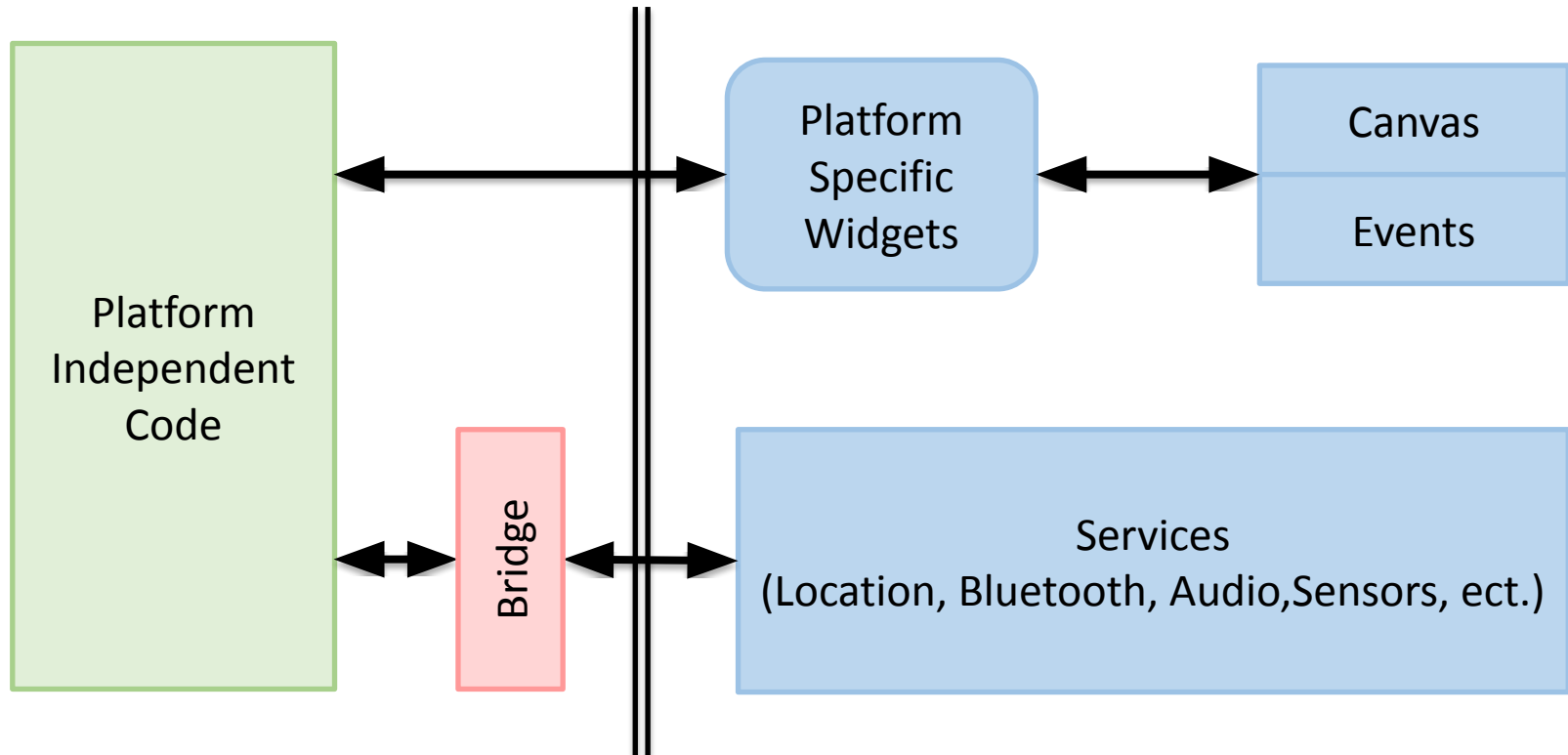
Native



3 Framework Flutter

3.1 So sánh

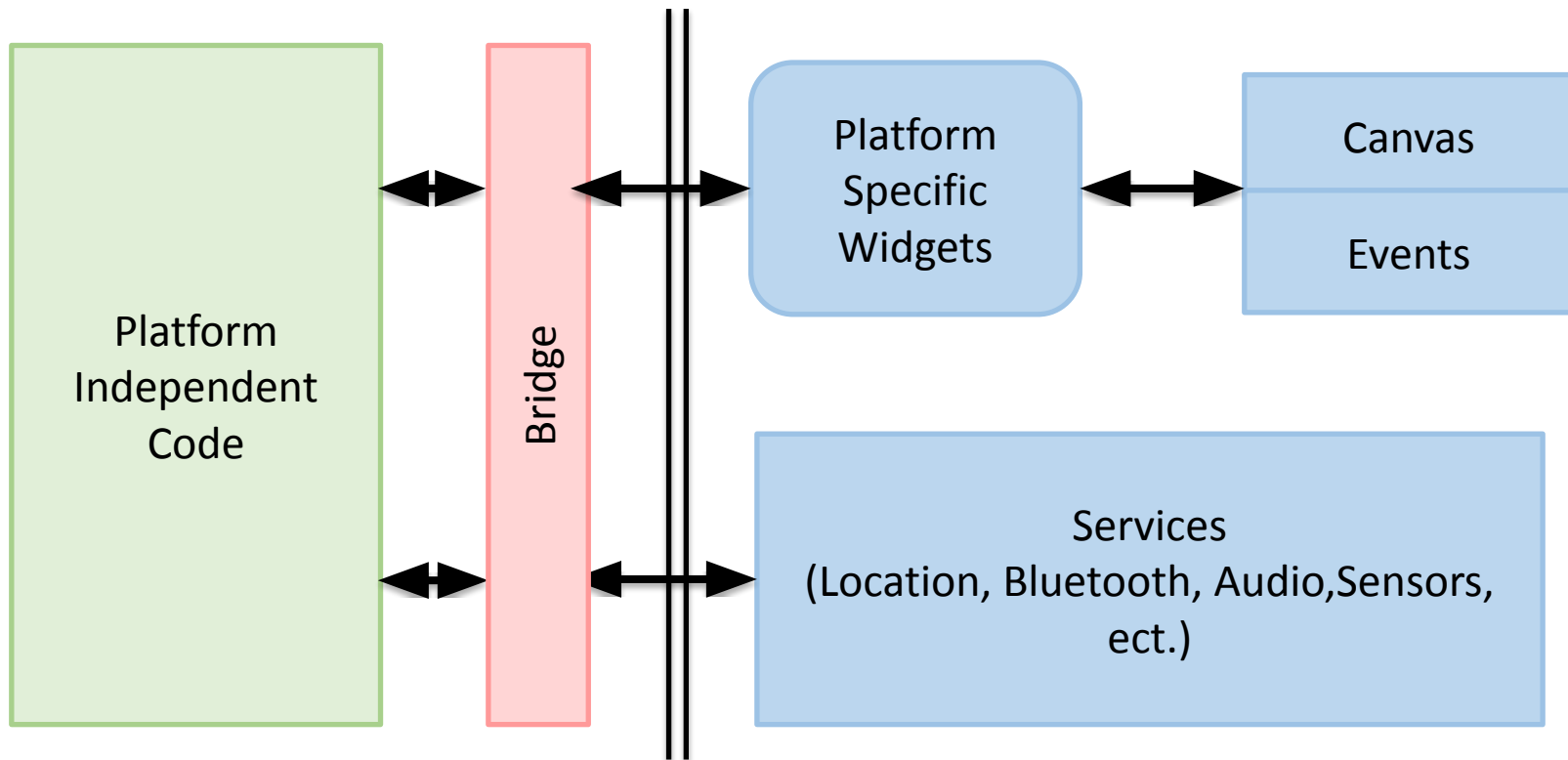
Web-based application



3 Framework Flutter

3.1 So sánh

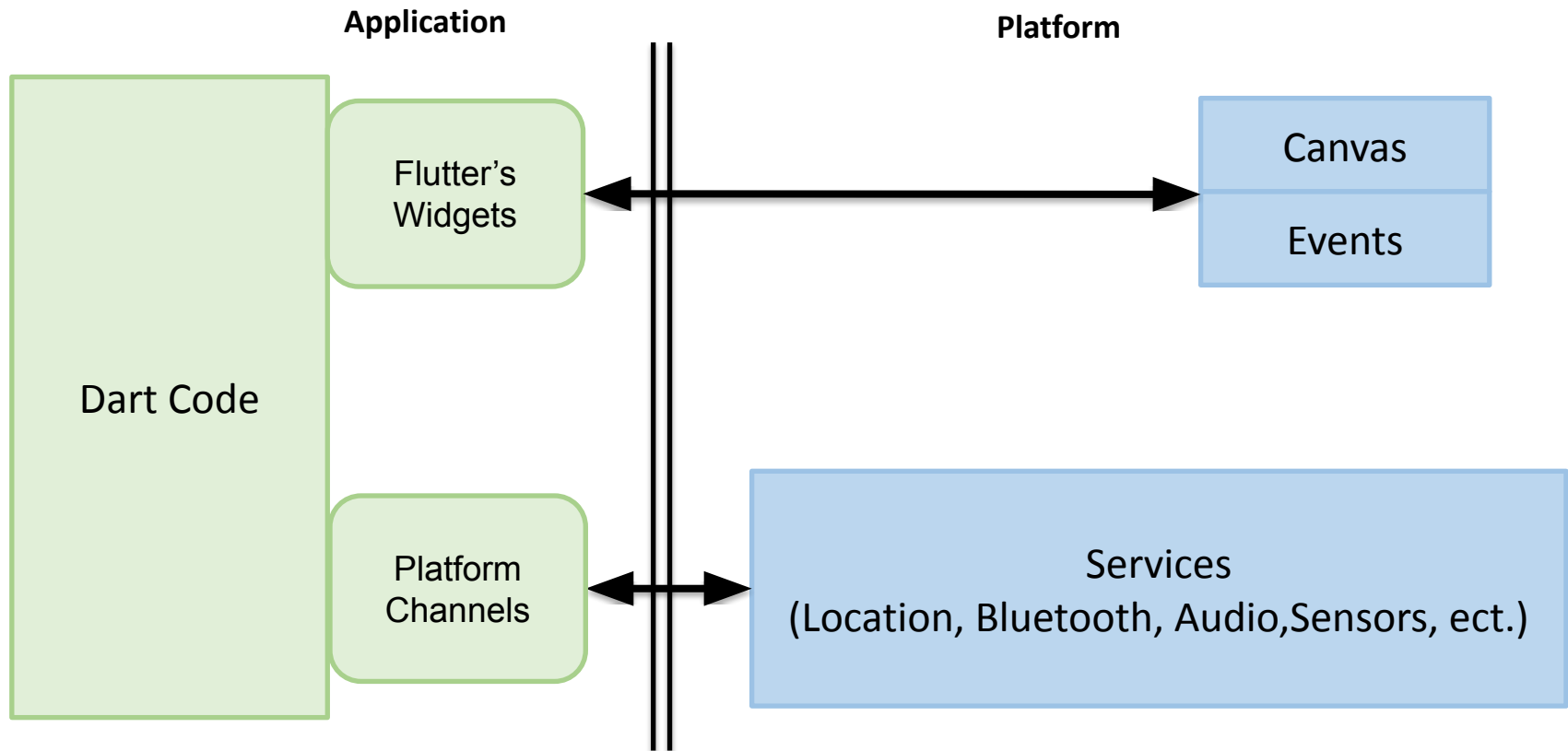
React Native



3 Framework Flutter

3.1 Kỹ thuật

Flutter



3 Framework Flutter

3.2 Declarative Framework

Ví dụ trong Flutter, mã Hexa của màu sắc được coi là một trạng thái nhất định.

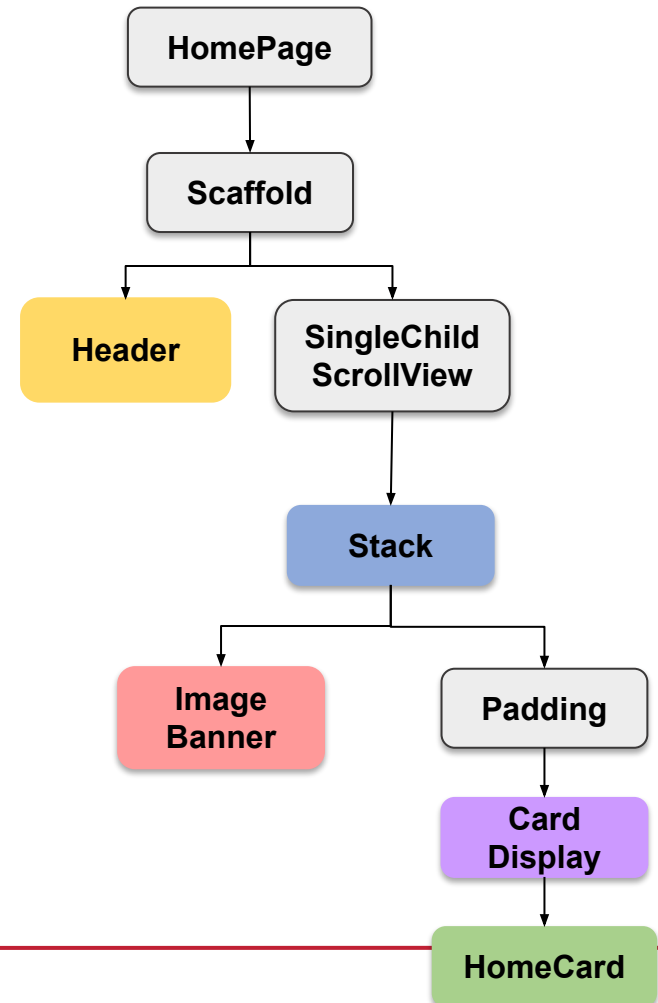
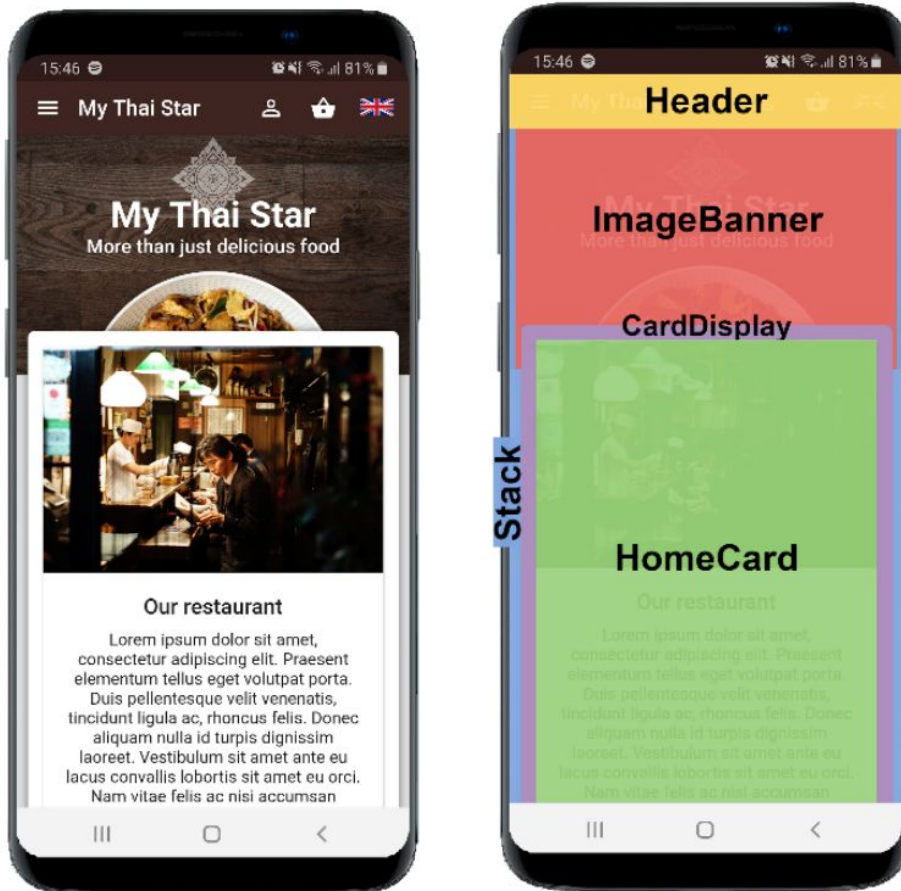
Button trong ví dụ Flutter là giá trị trả về của phương thức "build".

```
1.  bool pressed = false; //State
2.  @override
3.  Widget build(BuildContext context) {
4.      return FlatButton(
5.          color: pressed ? Colors.red : Colors.blue,
6.          onPressed: () {
7.              setState(){ //Kích hoạt update lại button
8.                  pressed = !pressed;
9.              }
10.         }
11.     );
12. }
```

3 Framework Flutter

3.3 Cây widget

Một ứng dụng Flutter về bản chất là một cây gồm các thành phần lồng nhau.



3 Framework Flutter

3.4 Các loại widget

Stateless Widgets

Các widget này hoàn toàn bất biến, điều đó có nghĩa là không có giá trị nào của chúng có thể thay đổi sau khi khởi tạo.

Một Stateless Widget chủ yếu bao gồm một phương thức build:

```
1. class MyWidget extends StatelessWidget{
2.   ///Được gọi nhiều lần trong một giây.
3.   ///
4.   ///Đây là nơi giao diện người dùng được xây dựng.
5.   @override
6.   Widget build(BuildContext context) {...}
7. }
```

Stateless Widgets nên chiếm phần lớn trong ứng dụng Flutter để cải thiện hiệu suất.

Tham khảo:

Putting build methods on a diet (I. Krankka) : <https://irokekrankka.com/2018/06/18/putting-build-methods-on-a-diet/>.

Performance best practices (Dart Team): <https://flutter.dev/docs/testing/best-practices>.

3 Framework Flutter

3.4 Các loại widget

Stateful Widgets

Đối tượng state có vòng đời phức tạp hơn widget;

Nó có một phương thức initState, một phương thức build và dispose

Các stateful widget nên được sử dụng càng ít càng tốt vì lý do hiệu suất.

Ba lý do sử dụng Stateful Widget

1. Widget có các dữ liệu mà chúng phải thay đổi trong suốt vòng đời của widget.
2. Widget cần phải thực hiện một số thao tác khởi tạo khi bắt đầu vòng đời của nó.
3. Widget cần phải vứt bỏ dữ liệu gì đó hoặc tự dọn dẹp vào cuối vòng đời.

3 Framework Flutter

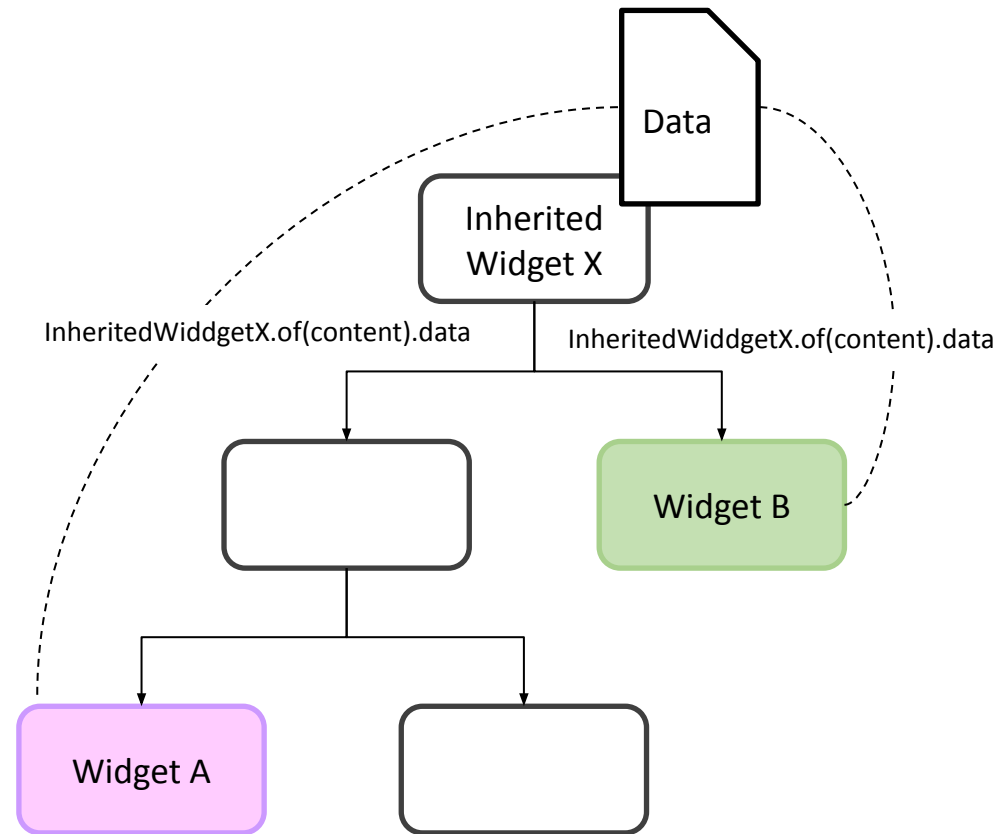
3.4 Các loại widgets

Inherited Widget

Loại widget cho phép truyền dữ liệu trong cây widget.

Một Inherited Widget nhất định có thể truyền bất kỳ loại dữ liệu nào cho tất cả con của nó trong cây widget.

Sau đó, tất cả con có thể truy cập dữ liệu đó thông qua Build Context như được hiển thị trong hình dưới đây.



Build Context là một đối tượng được truyền vào mọi phương thức xây dựng của một widget chứa tham chiếu đến tất cả cha của widget đó trong cây widget.

4. Chiến lược quyết định



4. Chiến lược quyết định

Mười chiến lược cần được quyết định khi sử dụng Flutter:

1. Lựa chọn giải pháp quản lý trạng thái
2. Lựa chọn một kiến trúc
3. Lựa chọn khai báo các lớp Model và cách cài đặt chúng
4. Lựa chọn các lớp sẽ kế thừa Equatable
5. Lựa chọn các đối tượng bất biến (immutable)
6. Lựa chọn Dependency injection
7. Lựa chọn đặt tên file và thư mục
8. Lựa chọn mô-đun hóa
9. Lựa chọn Validation
10. Lựa chọn bản địa hóa ứng dụng

4. Chiến lược quyết định

4.1 Quản lí State(trạng thái)

Sự khác biệt giữa quản lý state và kiến trúc

Quản lý state là quản lý trạng thái của một ứng dụng.

Kiến trúc là cấu trúc bao quát của một ứng dụng. Một tập hợp các quy tắc mà một ứng dụng tuân thủ.

Bất kỳ kiến trúc nào cho ứng dụng Flutter sẽ có một số loại quản lý trạng thái, nhưng quản lý trạng thái không phải là một kiến trúc của chính nó.

Chọn giải pháp quản lý state

Flutter không áp đặt bất kỳ loại kiến trúc hoặc giải pháp quản lý trạng thái nào đối với các nhà phát triển của nó.

Cách tiếp cận mở này đã dẫn đến nhiều giải pháp quản lý trạng thái và một số cách tiếp cận kiến trúc sinh ra từ cộng đồng. Một số cách tiếp cận này thậm chí đã được xác nhận bởi chính nhóm Flutter.

4. Chiến lược quyết định

4.1 Quản lí State(trạng thái)

Giải pháp	Nguồn gốc	Ưu điểm	Nhược điểm
Provider	Được phát triển bởi Remi Rousselet vào năm 2018. Bây giờ là sự hợp tác của đội Flutter và Rousselet	Tốt cho các ứng dụng nhỏ dễ làm: được xác nhận nhiều lần bởi nhóm Flutter	Không có liên quan kiến trúc; không phải là một cách tiếp cận có thể mở rộng
Redux	Ban đầu được tạo cho React vào năm 2015 bởi Dan Abramov. Sau đó được Brian Egan chuyển sang Flutter vào năm 2017.	Quy tắc được xác định rõ ràng; thay đổi trạng thái là hoàn toàn có thể dự đoán được; có thể được sử dụng để thực hiện kiến trúc ba lớp: UI- store - data	Store sẽ trở nên rất lớn với các ứng dụng lớn; mất nhiều công sức học tập

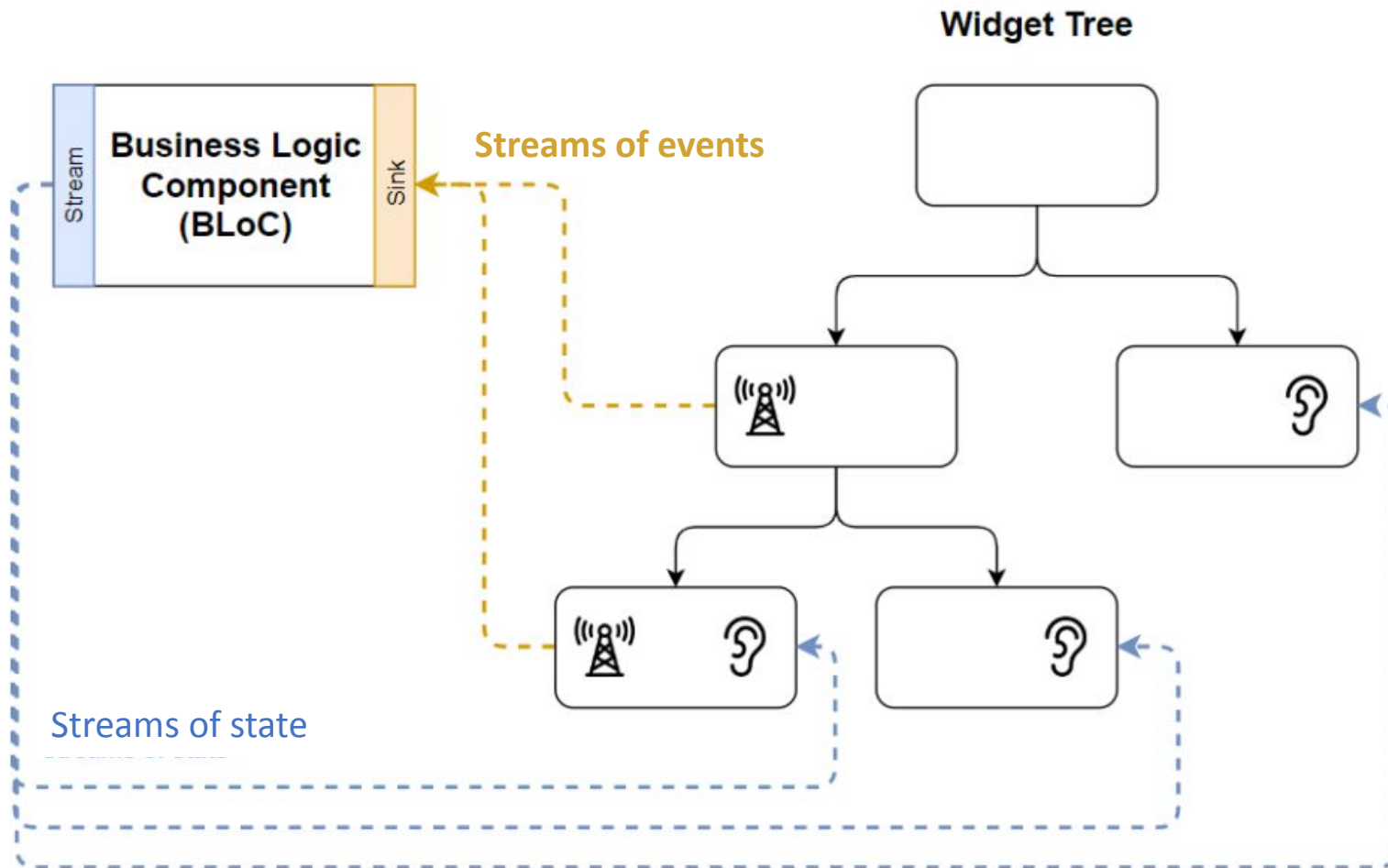
4. Chiến lược quyết định

4.1 Quản lí State(trạng thái)

Giải pháp	Nguồn gốc	Ưu điểm	Nhược điểm
BLoC Pattern	Được thiết kế bởi Paolo Soares, một trong những nhà phát triển riêng của Google, vào năm 2018	Có các quy tắc kiến trúc rõ ràng; đã được chứng thực nhiều lần bởi nhóm Flutter; thay đổi trạng thái là hoàn toàn có thể dự đoán được; cho phép thực hiện kiến trúc bốn lớp: UI – BLoC – repository - data	Cũng mất nhiều công sức học tập

4. Chiến lược quyết định

BLoC biến các sự kiện đầu vào thành một luồng trạng thái



4. Chiến lược quyết định

4.2 Kiến trúc

Lựa chọn và làm theo một kiến trúc được xác định rõ ràng là một phần thiết yếu của các dự án phần mềm quy mô lớn;

Một kiến trúc tốt sẽ đảm bảo rằng một ứng dụng nhất định luôn có thể quản lý được với codebase ngày càng phát triển.

Một ứng dụng triển khai các BLoC có thể dễ dàng được sửa đổi để cài đặt một kiến trúc Layer chỉ với một vài ràng buộc bổ sung.

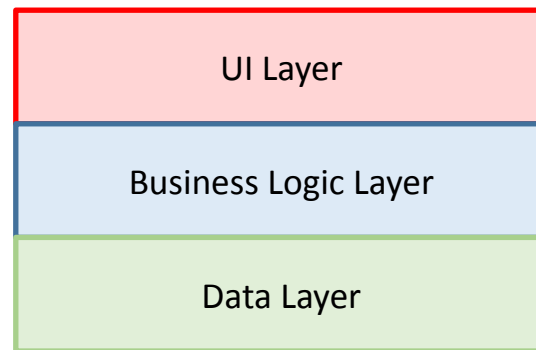
Phần này khám phá kiến trúc nhiều Layer là gì và cách nó có thể được triển khai với các BLoC.

4. Chiến lược quyết định

4.2 Kiến trúc

Mô hình BLoC và kiến trúc tầng

Ý tưởng chung của phong cách kiến trúc này là chia một ứng dụng "horizontally" thành các tầng chịu trách nhiệm cho một khía cạnh của ứng dụng.



Mục tiêu là giữ cho các tầng này độc lập nhất có thể với nhau. Thay đổi code trong một tầng không nên ảnh hưởng đến chức năng của các tầng khác.

Bloc Package: <https://felangel.github.io/bloc/#/>

Flutter Architecture Samples: <https://fluttersamples.com/>

A Practical Architecture for Flutter Apps:

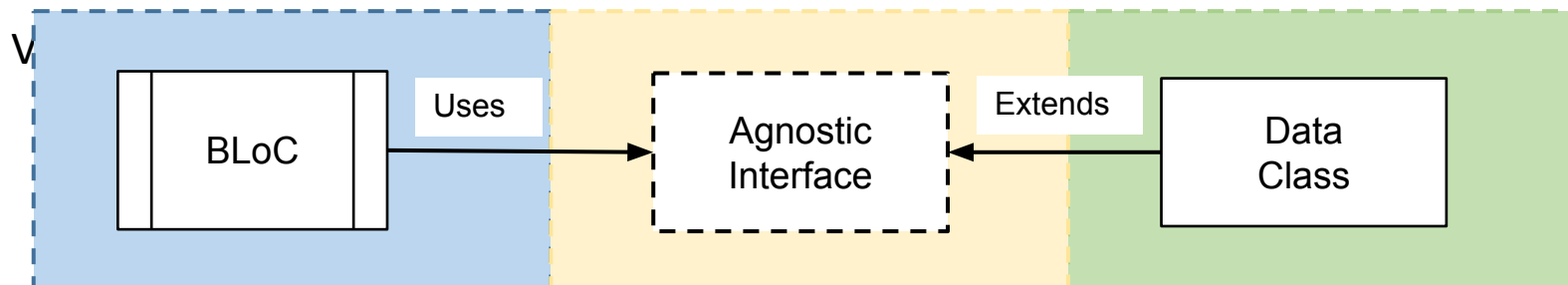
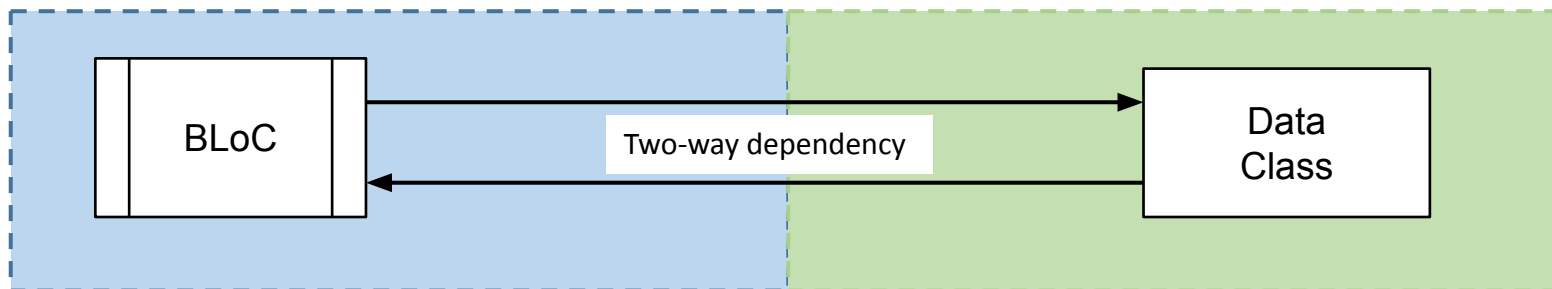
<https://medium.com/coding-with-flutter/widget-async-bloc-service-a-practical-architecture-for-flutte>

4 Phát triển ứng dụng quy mô lớn

Mô hình BLoC và kiến trúc tầng

Repository Pattern trong Flutter:

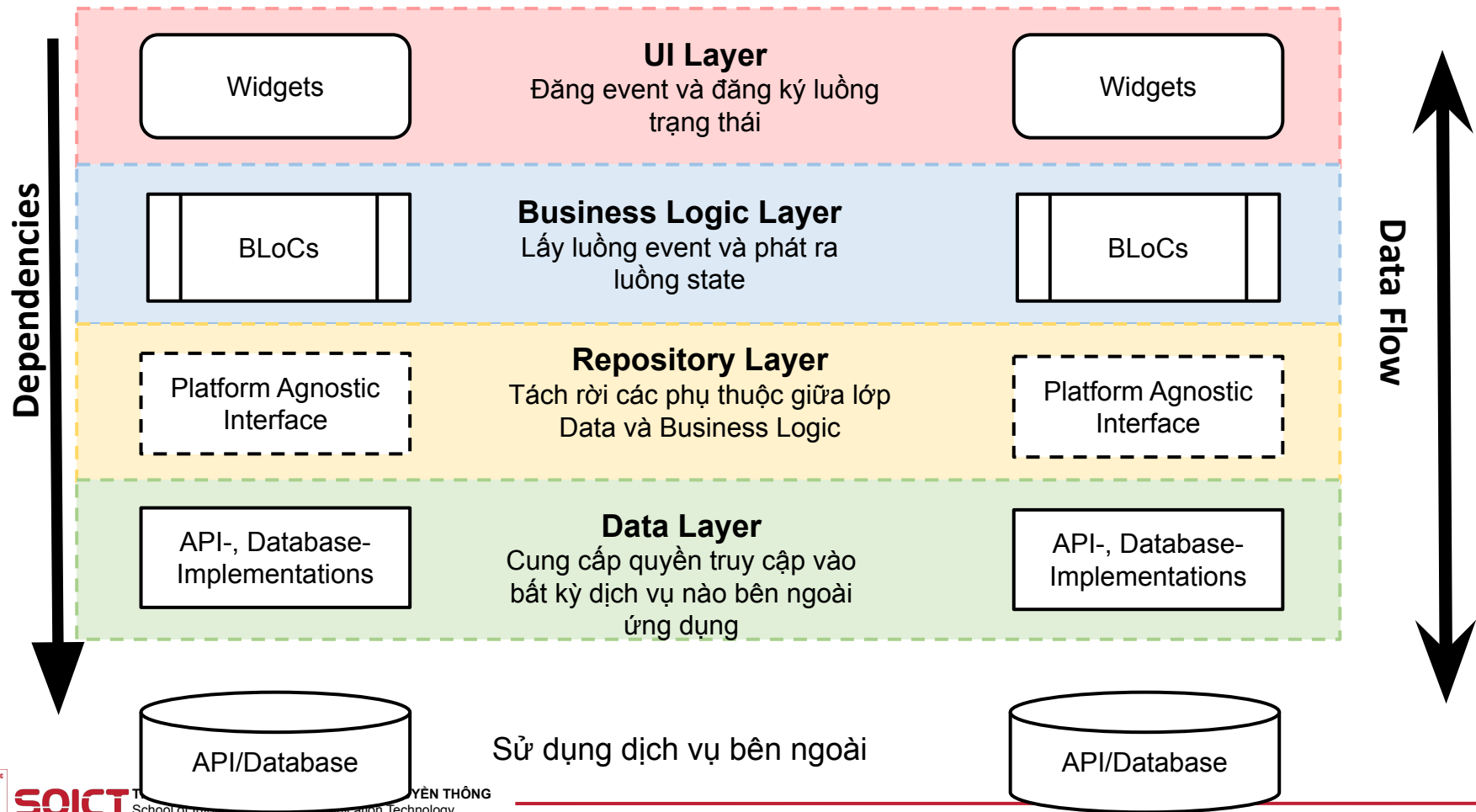
Không có Repository Pattern



4. Chiến lược quyết định

Mô hình BLoC và kiến trúc tầng

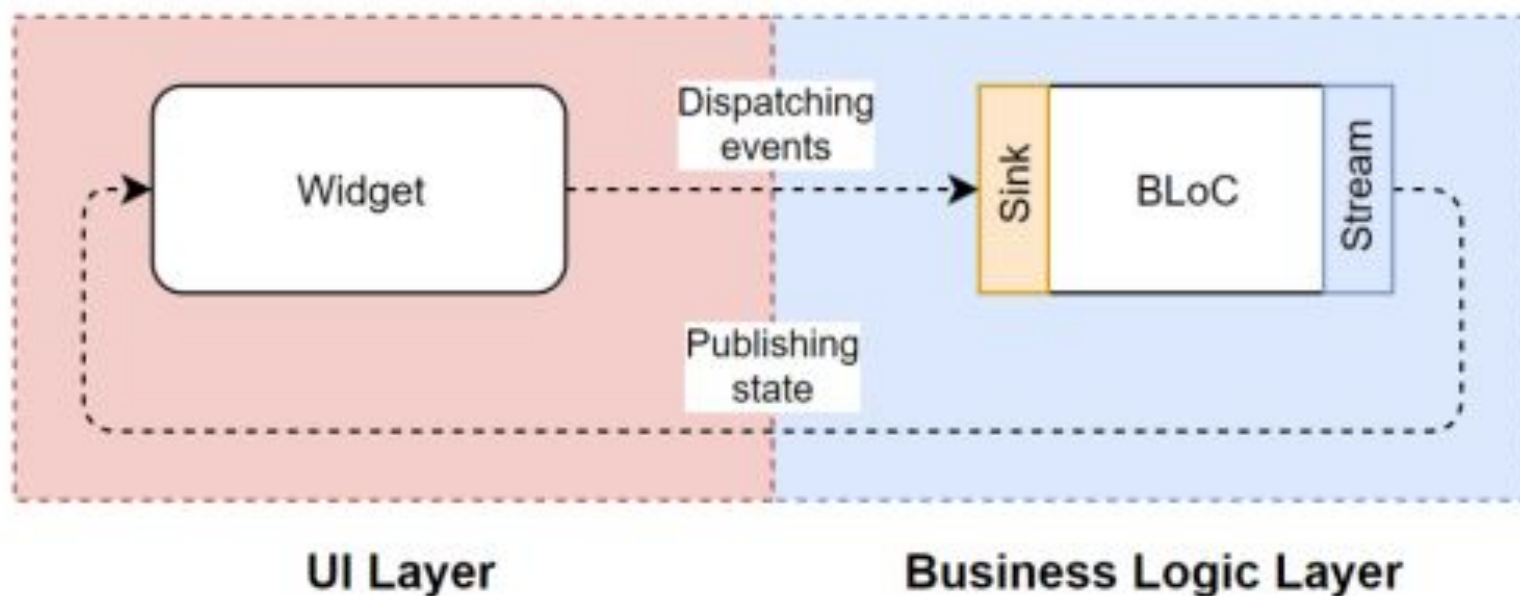
Kiến trúc bốn tầng sử dụng BLoC



4. Chiến lược quyết định

Mô hình BLoC và kiến trúc tầng

Để thực hiện quy tắc đầu tiên cho BLoCs, bất kỳ giao tiếp nào giữa tầng UI và business logic đều bị giới hạn ở các luồng như trong hình dưới đây:



4. Chiến lược quyết định

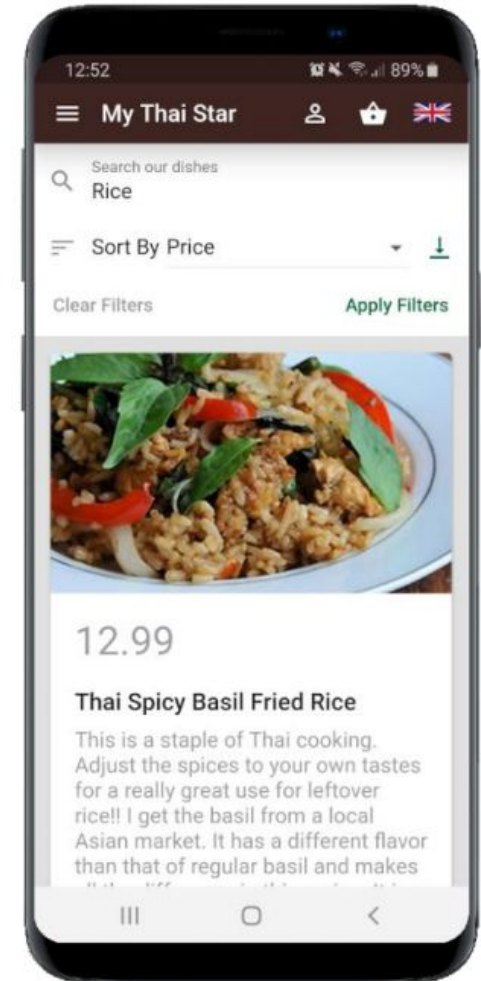
Thiết kế BLoCs

Tạo tính năng “Digital menu” trong ứng dụng My Thai Star Flutter

Tính năng này ban đầu được triển khai dưới dạng một "Dish BLoC".

Giao diện người dùng phát ra các sự kiện có chứa truy vấn đầy đủ. Điều này khiến BLoC lấy các món ăn mới từ backend.

Dish BLoC sau đó sẽ phát ra một danh sách các món ăn phù hợp với truy vấn.



4. Chiến lược quyết định

Thiết kế BLoCs

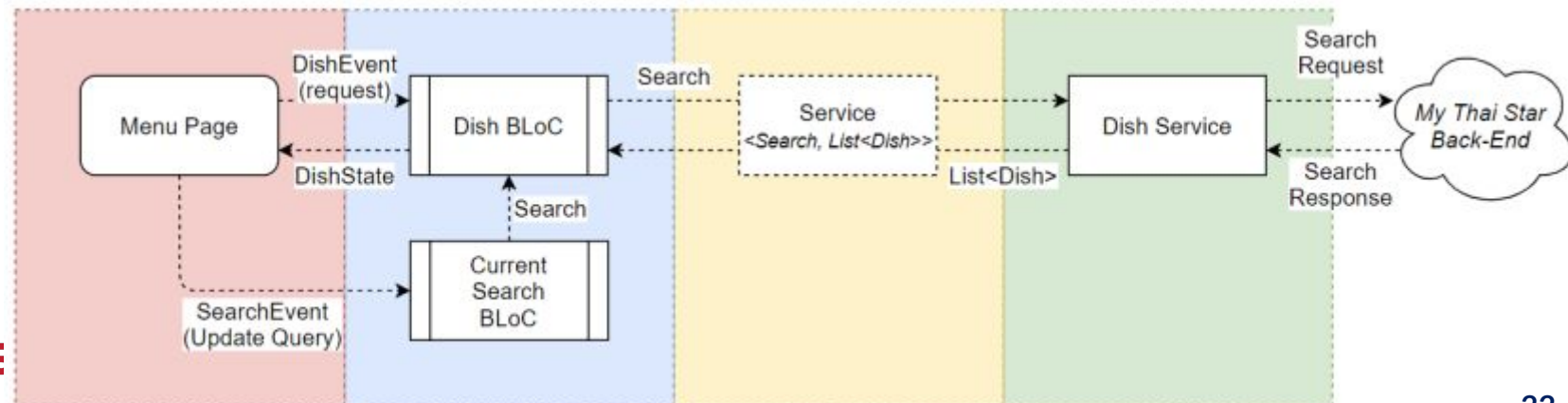
Split the old Dish BLoC into

- Current Search BLoC: xử lý trạng thái tìm kiếm
- new Dish BLoC: lấy các món ăn mới.

Digital Menu Feature as 1 BLoC



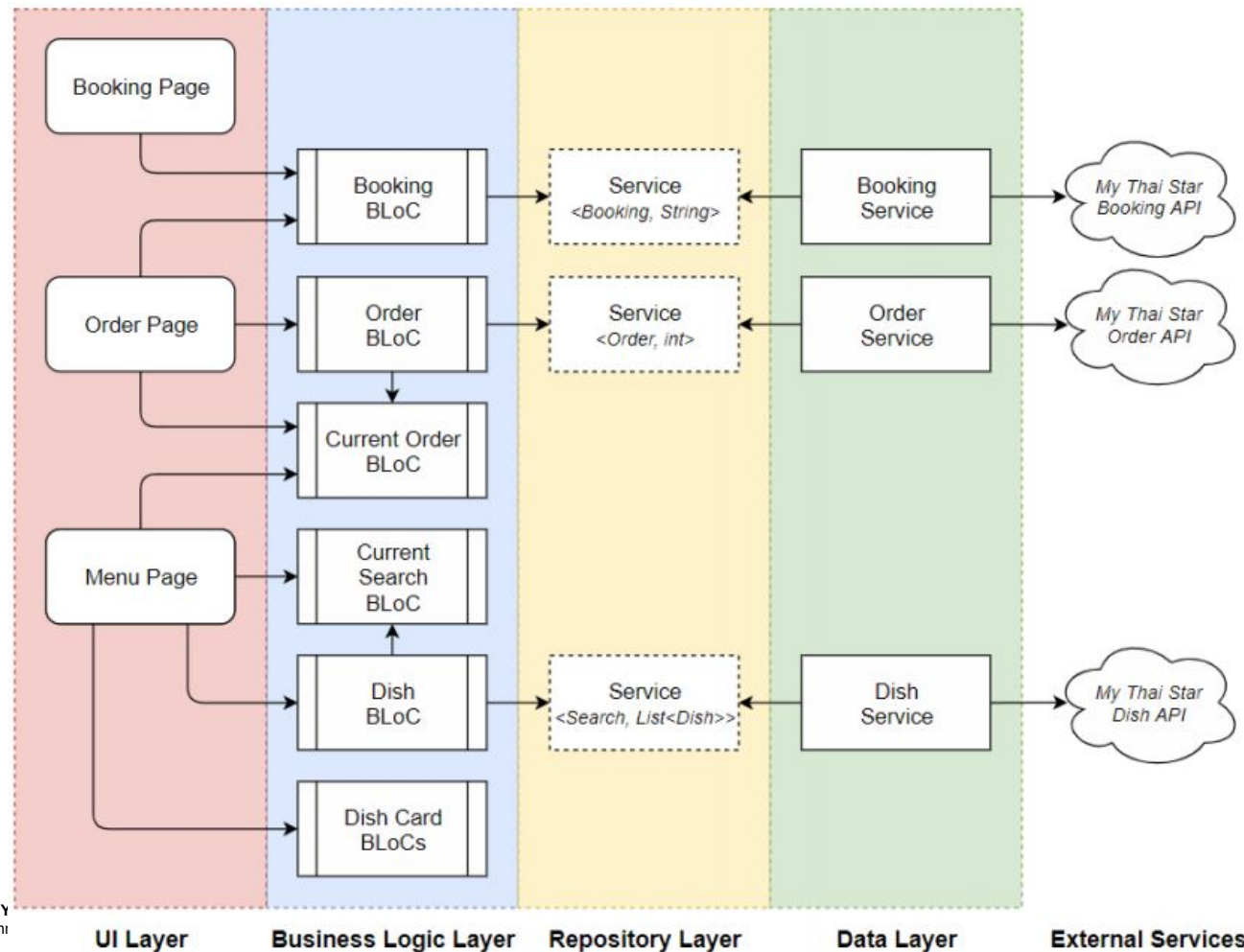
Digital Menu Feature as 2 BLoCs



4. Chiến lược quyết định

Kiến trúc của dự án

kiến trúc tầng của ứng dụng My Thai Star Flutter



4. Chiến lược quyết định

Kiến trúc của dự án

Tầng Business Logic

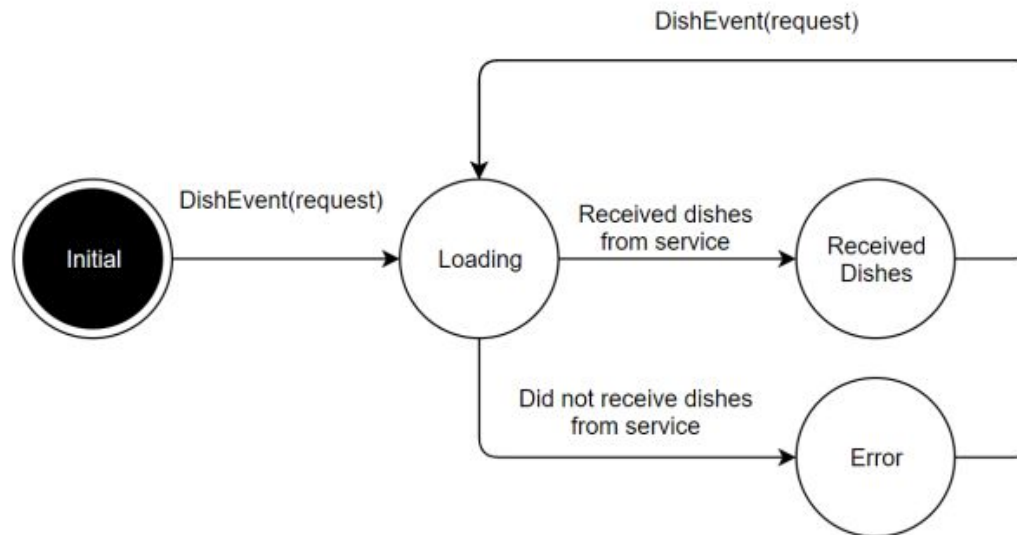
BLoC	Mô tả
Booking	Sử dụng thông tin cần thiết để đặt bàn. Thực hiện đặt chỗ đó và phát ra trạng thái thông báo nếu đặt phòng thành công
Current Order	Giữ danh sách các món ăn tạo nên trật tự hiện tại. Sử dụng các sự kiện thêm/loại bỏ món ăn khỏi đơn hàng hiện tại. Phát ra state mô tả thứ tự hiện tại.
Order	Dùng mã thông báo đặt chỗ và cố gắng đặt hàng danh sách các món ăn được cung cấp bởi The Current Order BLoC. Các món ăn được đặt cho đặt phòng liên quan đến mã đặt phòng. Phát ra trạng thái mô tả nếu đặt hàng thành công.
Current Search	Giữ truy vấn tìm kiếm hiện tại. Sử dụng các sự kiện thay đổi các khía cạnh của tìm kiếm đó. Phát ra trạng thái mô tả tìm kiếm hiện tại.
Dish	Sử dụng các sự kiện kích hoạt yêu cầu cho nhiều món ăn hơn. Khi một yêu cầu như vậy được sử dụng, BLoC này cố gắng lấy các món ăn mới dựa trên tìm kiếm được xác định bởi BLoC tìm kiếm hiện tại. Sau đó, nó phát ra trạng thái mô tả nếu việc tìm nạp thành công. Nếu có, state chứa một danh sách các món ăn phù hợp với tìm kiếm hiện tại.
Dish Card	Xử lý trạng thái của một widget Dish Card. Trạng thái của Dish Card có thể được sửa đổi bằng cách chọn hoặc bỏ chọn các đặt món bổ sung cho món ăn nhất định. Dish Card BLoC dùng các sự kiện thêm/loại bỏ các đặt món bổ sung. Sau đó, nó phát ra món ăn mới qua việc thay đổi state

4. Chiến lược quyết định

Kiến trúc của dự án

Tầng Business Logic

Sơ đồ trạng thái sau đây cho thấy tất cả các chuyển tiếp trạng thái có thể có của Dish BLoC.

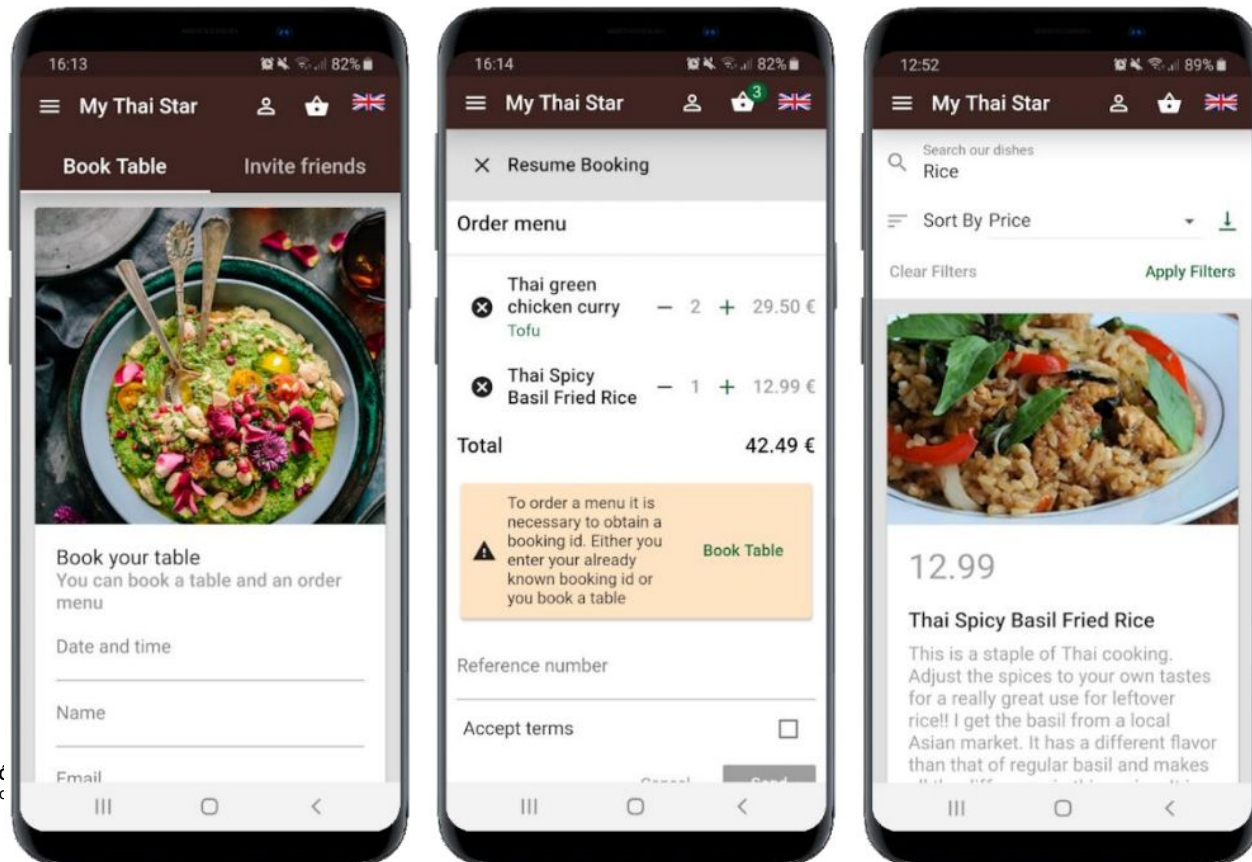


Đây là một thực tế phổ biến để sử dụng các lớp kế thừa để thực hiện các state và sự kiện. Bằng cách này, tên của lớp có thể truyền đạt state/event đã cho là gì và các thành viên của lớp có thể mang theo bất kỳ dữ liệu liên

4. Chiến lược quyết định

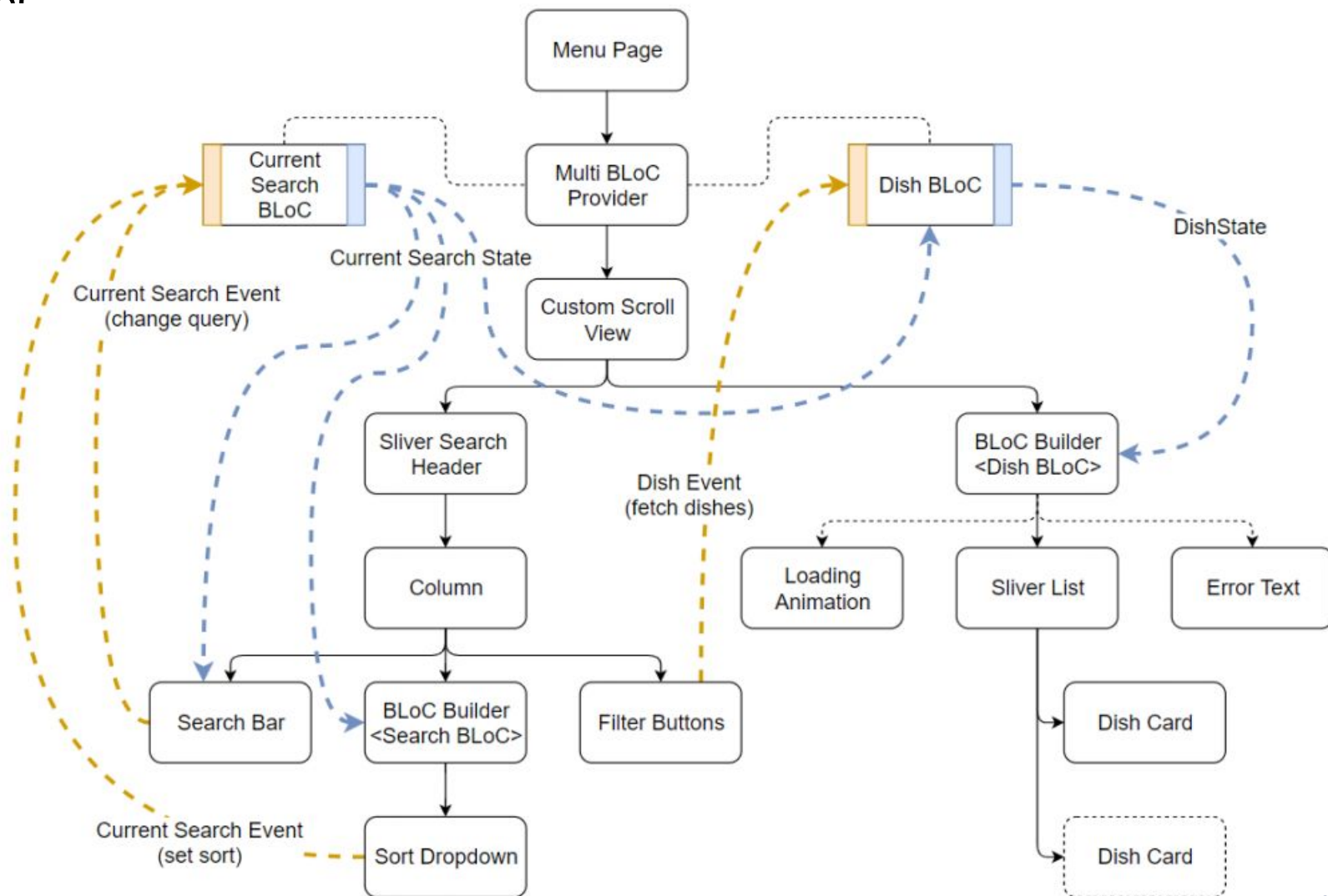
Tầng UI

Giao diện người dùng của các trang này được hiển thị trong Hình dưới đây. Mỗi trang bao gồm một cây gồm nhiều tiện ích lồng nhau. Mỗi BLoC của ứng dụng này được khởi tạo trong lớp UI.



4. Chiến lược quyết định

Cây widget của trang Menu với Current Search & Dish BLoC



4. Chiến lược quyết định

Tầng Repository

Tầng Repository của ứng dụng này chỉ bao gồm một giao diện vì tất cả các lớp trong tầng dữ liệu đều đủ giống nhau để mở rộng một giao diện chung.

1. `/// Sử dụng interface độc lập`
2. `[Input] cho [Output] không đồng bộ`
3. `abstract class Service<Input, Output> {`
4. `///Trao đổi một [Input] cho một [Future//<Output>]`
5. `Future<Output> post(Input input);`
6. `}`

4. Chiến lược quyết định

Lớp Data

Triển khai dịch vụ món ăn

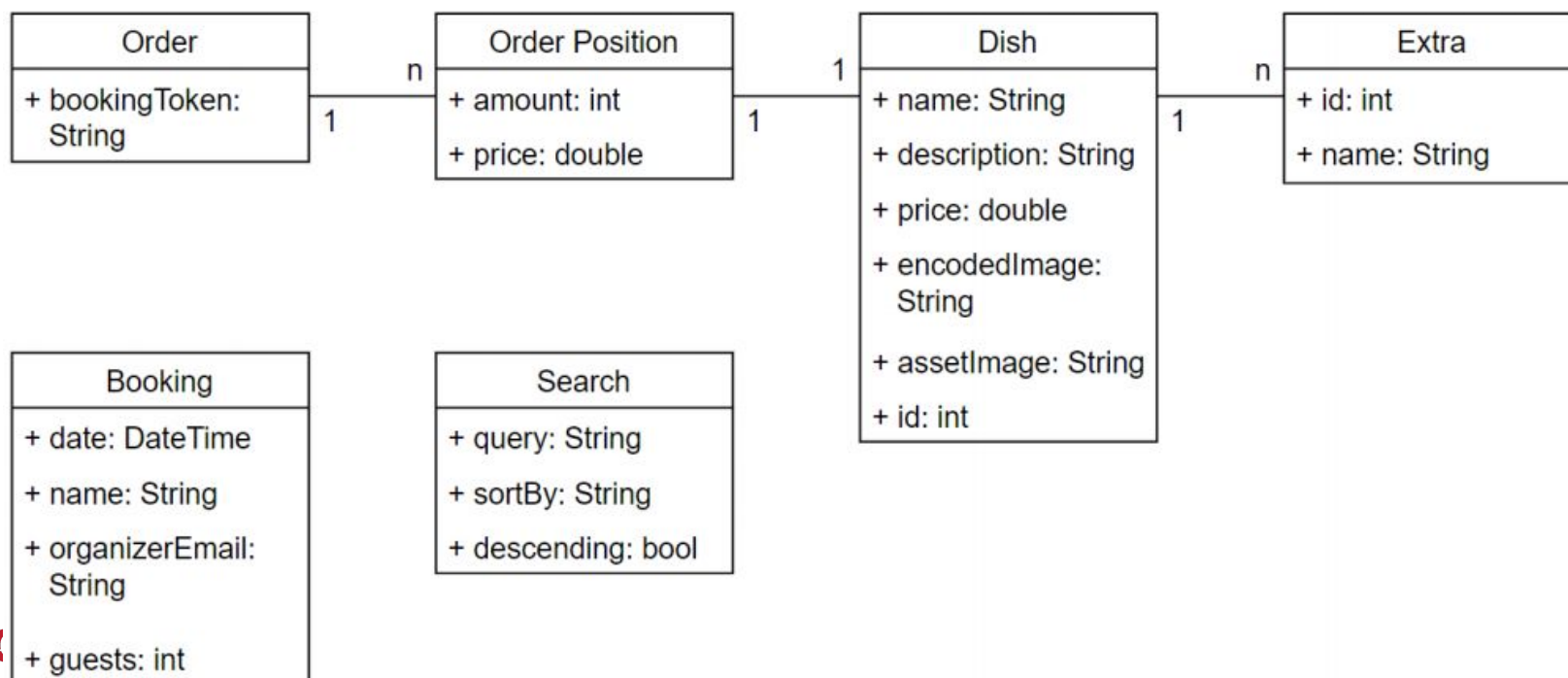
```
1. @immutable
2. class DishService extends Service<Search, List<Dish>> {
3.     static const String _route =
4.         'mythaistar/services/rest/dishmanagement/v1/dish/search'
5.     ;
6.     final String _baseUrl;
7.
8.     static const Map<String, String> _requestHeaders = {
9.         'Content-type': 'application/json',
10.        'Accept': 'application/json',
11.    };
12. DishService({@required String baseUrl}) : _baseUrl = baseUrl;
13.
14.    ///Đăng [Search] đến API và trả về [Dish] khớp với [Search].
15.    /// trả về [Ngoại lệ] nếu giao tiếp với API không thành công
16.    /// hoặc không tìm thấy [Dish] phù hợp.
```

4. Chiến lược quyết định

4.3 Lớp Model

Mọi loại lớp chính liên quan đến dự án này được mô tả trong phần kiến trúc ngoại trừ các lớp thường được gọi là “Model”.

Các lớp này được sử dụng để mô hình hóa các đối tượng có trong một miền cụ thể.



4. Chiến lược quyết định

4.4 So sánh bằng

Theo mặc định, bất kỳ đối tượng nào trong Dart đều được so sánh bằng tham chiếu của nó.

```
Extra extra1 = Extra(name: "Tofu", id: 1);  
Extra extra2 = Extra(name: "Tofu", id: 1);  
print(extra1 == extra2); //False
```

Điều này có thể dẫn đến hành vi không mong muốn khi so sánh các trạng thái hoặc mô hình với nhau.

Dart cung cấp tùy chọn để ghi đè hành vi so sánh của bất kỳ đối tượng nào.

Một lớp sẽ kế thừa "Equatable" sẽ so sánh các thuộc tính của đối tượng bởi mảng "props" của nó thay vì địa chỉ của nó trong bộ nhớ.

4. Chiến lược quyết định

4.5 Tính bất biến

Một đối tượng được coi là "bất biến" khi các giá trị của nó không thể thay đổi sau khi đối tượng được khởi tạo.

Trong ngữ cảnh lập trình hướng đối tượng, điều này chuyển thành tất cả các trường thành viên của một lớp nhất định được đánh dấu là "final".

Một khi một đối tượng bất biến được khởi tạo, nó không bao giờ có thể bị thay đổi.

```
@immutable
abstract class Widget extends DiagnosticableTree {
    const Widget({ this.key });

    ///...
    final Key key;
}
```

4. Chiến lược quyết định

4.6 Dependency Injection

Dependency injection có nghĩa là nếu một lớp A phụ thuộc vào một đối tượng của lớp B, lớp A không khởi tạo ra đối tượng của lớp B.

Thay vào đó, lớp A được truyền vào một đối tượng thuộc lớp B.

Ví dụ cơ bản về DI được hiển thị trong đoạn mã sau:

```
class A {  
    B b;  
    A(this.b);  
}
```



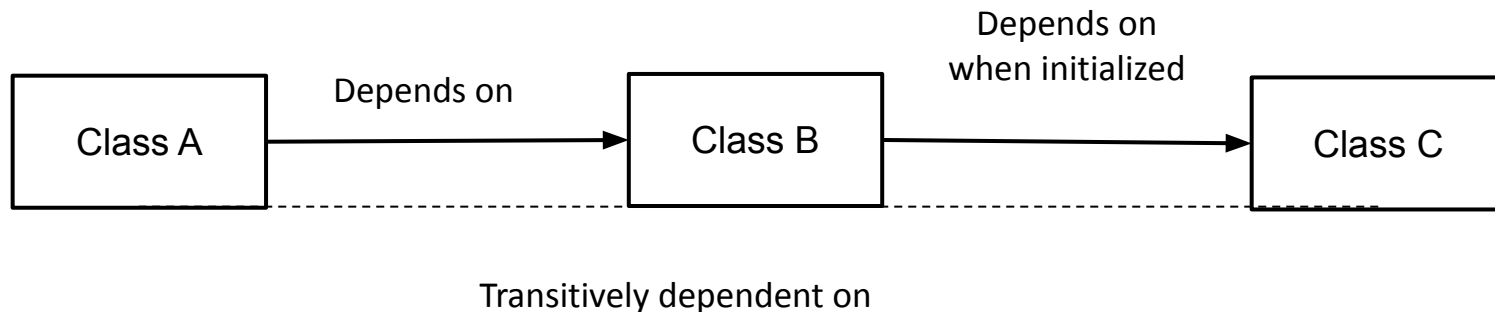
4. Chiến lược quyết định

4.6 Dependency Injection

Ưu điểm

Sử dụng DI dẫn đến mã có thể tái sử dụng nhiều hơn, vì việc tiêm các phụ thuộc phá vỡ "sự phụ thuộc tạm thời".

Ví dụ về phụ thuộc tạm thời



Khả năng tái sử dụng được khuếch đại hơn nữa khi sử dụng DI kết hợp với Repository Pattern

Tên	Cấu trúc	Mô tả	Ưu điểm	Nhược điểm
Component-Based	<pre> lib ├── componentA ├── componentB ├── componentC ├── componentD ├── blocs ├── data ├── models ├── repositories ├── ui └── main.dart </pre>	Đây là một cách tiếp cận thường được sử dụng trong React. Ý tưởng là căn chỉnh cấu trúc tệp với các thành phần khác nhau có trong ứng dụng. Mỗi thư mục thành phần chứa tất cả các BLoCs, model, widget, v.v. liên quan đến thành phần đó	Làm cho ứng dụng dễ mô-đun hóa hơn; thư mục có thể được biến thành các gói riêng biệt; tất cả các tệp liên quan đến một tính năng nhất định đều ở một nơi.	Không thích hợp cho các ứng dụng nhỏ
Tree Like	<pre> lib ├── root │ ├── blocs │ ├── data │ ├── models │ ├── repositories │ ├── pageA │ ├── pageB │ └── pageC │ ├── blocs │ ├── data │ ├── models │ ├── repositories │ ├── featureC1 │ └── featureC2 └── main.dart </pre>	Cách tiếp cận này được sử dụng bởi nhóm của Felix Angelov tại BMW. Theo cách tiếp cận này, cấu trúc tệp được căn chỉnh với cây widget của ứng dụng. Các tệp khác nhau được đặt ở một vị trí trong cấu trúc tệp liên quan đến vị trí của chúng trong cây widget.	Vị trí từ một BLoC nhất định có thể dễ dàng nhận dạng; các nhà phát triển biết cấu trúc của cây widget, cũng biết cấu trúc của dự án.	Nhiều thành phần; các widget được sử dụng ở nhiều vị trí của ứng dụng không thể tuân thủ nghiêm ngặt cấu trúc.
Layered	<pre> lib ├── blocs ├── data ├── models ├── repositories ├── ui └── main.dart </pre>	Cách tiếp cận này được trích dẫn phổ biến nhất trong cộng đồng Flutter . Nó căn chỉnh kiến trúc tầng của ứng dụng với cấu trúc tệp của nó.	Dễ dàng mở rộng	Nhiều tệp trong một thư mục nhất định; các tệp liên quan đến một tính năng được chia thành nhiều vị trí.

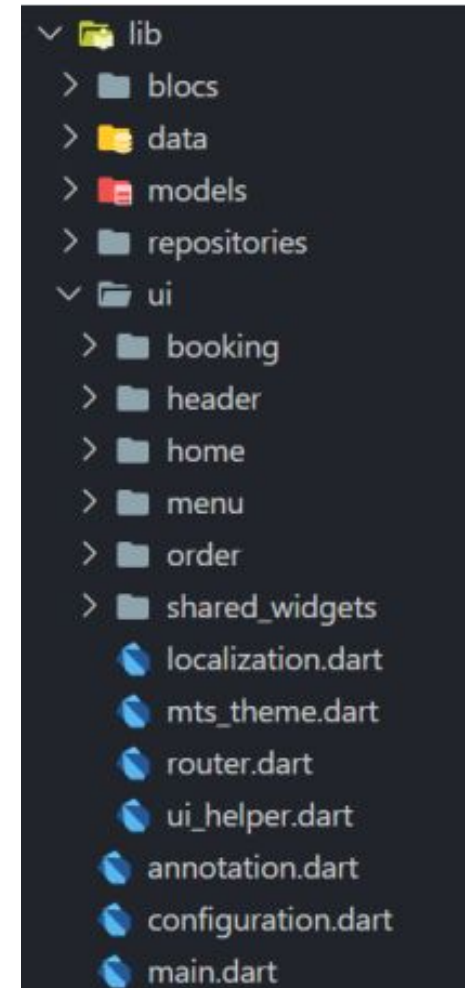
4. Chiến lược quyết định

4.7 Đặt tên file và thư mục

Thư mục UI được cấu trúc để hiển thị các trang khác nhau của ứng dụng như trong hình dưới đây.

Tiêu đề của ứng dụng cũng có thư mục riêng của nó, vì chứa nhiều tệp tiện ích.

Bằng cách này, nhược điểm chính của phương pháp tiếp cận cấu trúc tệp Layered đã được giảm thiểu.




4. Chiến lược quyết định

4.7 Đặt tên file và thư mục

Mục đích của một số tệp trong dự án

Tên tệp	Mục đích
Localization	Giữ lớp Localization Delegate and Translation chịu trách nhiệm “bản địa hóa” các văn bản trong dự án này.
MTS Theme	Giữ dữ liệu chủ đề tùy chỉnh. Xác định màu sắc chính, phong chữ, v.v.
Router	Chịu trách nhiệm điều hướng ở giữa các trang của ứng dụng
UI Helper	Giữ một tập hợp các giá trị hằng số tĩnh cho lẽ giữa các widget
Annotation	Xác định chú thích tùy chỉnh cho ứng dụng để dễ dàng tìm thấy các lớp phục vụ cùng một mục đích hoặc đang trong giai đoạn phát triển tương tự.
Configuration	<p>Giữ một tập hợp các giá trị hằng số tĩnh tạo nên một danh sách các cấu hình chung cho ứng dụng như được hiển thị trong đoạn mã sau:</p> <pre>@immutable class Configuration { static final String baseUrl = "http://138.197.218.225:8082/"; static final bool useMockData = false; static final bool logging = true; static final int defaultTimeout = 5; //In Seconds }</pre>

**SOICT**
TRƯỜNG CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
School of Information and Communication Technology

Main

Xác định gốc của cây widget và điểm bắt đầu của ứng dụng.

4. Chiến lược quyết định

4.8 Mô-đun hóa

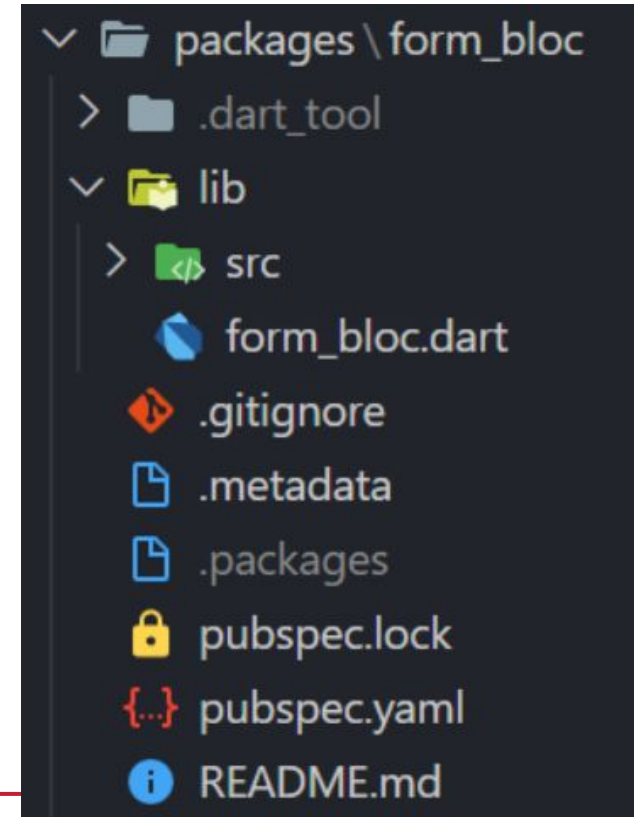
Mô-đun hóa trong Dự án này

Dart cung cấp khả năng mô-đun hóa một ứng dụng bằng cách trích xuất một chức năng vào một gói Dart riêng biệt.

Một gói Dart có cấu trúc rất giống với ứng dụng Flutter. Hình dưới đây cho thấy cấu trúc tệp của gói "Form BLoC" được tạo trong luận án này.

Bất kỳ tệp Dart nào cần quyền truy cập vào chức năng do gói cung cấp đều có thể thêm như được hiển thị trong đoạn mã sau:

```
import 'package:form_bloc/form_bloc.dart';
```



4. Chiến lược quyết định

4.9 Xác thực biểu mẫu

Xác thực biểu mẫu, trong bối cảnh phát triển phần mềm, mô tả quá trình xác thực nếu một tập hợp các giá trị được người dùng nhập vào Form tuân thủ các quy tắc được xác định trước.

Cách tiếp cận Inbuild của Flutter

Xác thực biểu mẫu trong bất kỳ ngữ cảnh nào phần lớn bao gồm ba bước:

1. Xác thực đầu vào của trường biểu mẫu
2. Xác nhận xem tất cả các trường của một biểu mẫu nhất định có hợp lệ hay không và do đó biểu mẫu hiện đã được điền hoàn toàn
3. Cung cấp tất cả các giá trị được nhập vào biểu mẫu để chúng có thể được xử lý thêm bởi các phần khác của ứng dụng.

4. Chiến lược quyết định

4.9 Xác thực biểu mẫu

Cách tiếp cận Inbuild của Flutter

Widget "Form Field" có thể được sử dụng để nhận ra đầu vào văn bản. Việc xác thực các trường biểu mẫu này được thực hiện thông qua các hàm "validator", lấy giá trị đầu vào trong kiểm tra xem giá trị đó có tuân thủ các quy tắc được xác định trước hay không.

```
1. TextFormField(  
2.   // Validator nhận được mà người dùng đã nhập  
3.   validator: (value) {  
4.     if (value.isEmpty) {  
5.       return 'Please enter some text'; //Invalid  
6.     }  
7.     return null; //Valid  
8.   },  
9. );
```

4. Chiến lược quyết định

4.10 Bản địa hóa

Bản địa hóa một ứng dụng cho nhiều quốc gia có tầm quan trọng rất quan trọng khi cố gắng tiếp cận đối tượng rộng nhất có thể với ứng dụng đó.

Năm 2012, công ty phân tích "Distimo" đã theo dõi việc tải xuống 200 ứng dụng trước và sau khi bản địa hóa cho một số quốc gia.

Kết quả: việc bản địa hóa một ứng dụng cho nhiều quốc gia có thể tăng lượt tải xuống ứng dụng đó trung bình 128% chỉ trong một tuần.

Bản địa hóa rất quan trọng đối với các ứng dụng quy mô lớn.

Phần này sẽ khám phá cách bản địa hóa đã được thực hiện trong dự án này và giới thiệu lý do tại sao phương pháp Flutter mặc định được chọn thay vì xây dựng lại tính năng này từ đầu với mô hình BLoC.

4. Chiến lược quyết định

4.10 Bản địa hóa

Bản địa hóa trong dự án

Flutter có một phương pháp sẵn có để xử lý nội địa hóa. Cách tiếp cận này sử dụng “Localization Delegate”, chịu trách nhiệm tạo các đối tượng chứa dữ liệu được bản địa hóa cho một quốc gia nhất định.

Các văn bản dịch của ứng dụng này được lưu trữ trong các tệp JSON ánh xạ khóa tiếng Anh với bản dịch của nó bằng ngôn ngữ đã cho như được hiển thị trong đoạn mã sau.

```
"table": {  
  "rowsPage": "Zeilen pro Seite",  
  "of": "von",  
  "noResults": "Keine Ergebnisse"  
},  
...
```

Tất cả các tệp JSON này tuân theo cùng một cấu trúc và sử dụng cùng một khóa, nếu không việc truy cập các bản dịch khác nhau của cùng một văn bản là rất khó khăn.

4. Chiến lược quyết định

Bản địa hóa trong dự án

Ví dụ: khi người dùng chuyển ứng dụng từ tiếng Anh sang tiếng Đức. Hành vi này được minh họa trong đoạn mã sau:

```
1.  /// Tạo và sau đó cung cấp các đối tượng [Translation] đến cây widget
2.  ///
3.  /// Cách xử lý nội địa hóa của Flutter là thông qua [LocalizationsDelegate].
4.  /// Một tùy chỉnh có thể được tạo bởi mở rộng lớp [LocalizationsDelegate].
5.  @immutable
6.  class MtsLocalizationDelegate extends LocalizationsDelegate<Translation> {
7.    static const List<String> supportedLanguages =
8.      ['en', 'de', 'bg', 'es', 'fr', 'nl', 'pl', 'ru'];
9.
10.    ///Tạo một [Translation] cho một [Locale] đã cho
11.    @override
12.    Future<Translation> load(Locale locale) async {
13.      Map<dynamic, dynamic> translationMap = await _loadFromAssets(locale);
14.      return Translation(locale, translationMap);
15.    }
16.
17.    /// Tải tệp JSON dịch của một [Locale] nhất định trong một assets và phân tích
18.    /// nó thành một [Map<dynamic,dynamic>]
19.    static Future<Map<dynamic, dynamic>> _loadFromAssets(Locale locale) async {
20.      String jsonContent = await rootBundle
21.        .loadString('assets/languages/${locale.languageCode}.json');
22.      return json.decode(jsonContent);
23.    }
24.  }
```

4. Chiến lược quyết định

Bản địa hóa trong case study

“Localization delegates” được đăng ký trong tiện ích Material ở thư mục gốc của ứng dụng.

Tiện ích Material có thuộc tính "locale", xác định quốc gia hiện tại mà ứng dụng cần được bản địa hóa.

Khi thuộc tính locale của ứng dụng được thay đổi, chức năng tải của Localization Delegate được kích hoạt dựa trên thay đổi địa lý.

Sự thay đổi của Locales được xử lý bởi "Localization BLoC" trong dự án này.

5. Tổng kết

5. Tổng kết

- Chúng ta đã nắm được động lực chính của các dự án đa nền tảng
- Hiểu được nguyên nhân gây ra khuyết điểm của các công nghệ đa nền tảng
- Nắm được 10 chiến lược quyết định đến sự thành công của dự án Flutter có quy mô lớn (tham vọng phục vụ hàng trăm nghìn người trở lên)
- Đã được giới thiệu các minh họa bằng mã nguồn của dự án My Thai Star

Thank You For Your Attention

Appendix

Cây widget

```
1. class HomePage extends StatelessWidget {
2.     static const double _cardDisplayTopPadding = 170;

3.     @override
4.     Widget build(BuildContext context) {
5.         return Scaffold(
6.             appBar: Header(),
7.             drawer: AppDrawer(),
8.             body: SingleChildScrollView( //Làm cho Stack có thể cuộn được
9.                 child: Stack(
10.                    children: <Widget>[
11.                        ImageBanner(),
12.                        Padding(
13.                            child: CardDisplay(), //Giữ 2 cái HomeCards
14.                            padding: EdgeInsets.only(top: _cardDisplayTopPadding)),
15.                    ],
16.                ),
17.            ),
18.        );
19.    }
20. }
```

Minh họa Stateful Widgets

Stateful Widgets

```
1. class MyWidget extends StatefulWidget {
2.     ///Được gọi ngay lập tức khi lần đầu tiên xây dựng StatefulWidget
3.     @override
4.     State<StatefulWidget> createState() => MyState();
5. }

6. class MyState extends State<MyWidget>{
7.     ///Được gọi là một khởi tạo
8.     @override
9.     initState(){...}
10.    ///Được gọi nhiều lần trong một giây.
11.    ///
12.    ///Đây là nơi giao diện người dùng được xây dựng.
13.    @override
14.    Widget build(BuildContext context){...}
15.    ///Được gọi một lần trước khi state được loại bỏ(tắt ứng dụng)
16.    @override
17.    dispose(){...}
18. }
```


Minh họa một gói BLoC

Gói BLoC

Việc triển khai lớp BLoC có thể trông như thế này:

```
1. // xác định lớp của các sự kiện mà nó sử dụng và lớp trạng thái mà
   //nó tạo ra
2. class NewBloc extends Bloc<Event, State>{
3. // Khởi tạo trạng thái ban đầu từ [Bloc]
4. @override
5. State get initialState => State();
6. // Được gọi mỗi khi một sự kiện mới được gửi đến [Bloc].
7. // Xác định điều gì sẽ xảy ra khi một sự kiện mới được sử dụng.
8. @override
9. Stream<State> mapEventToState(Event event) async* {
10. //Phát ra trạng thái mới như thế này:
11. //"yield" thêm một giá trị mới cho stream nhưng không chấm dứt hàm.
12. yield State();
13. }
14. }
```

Làm thế nào BLoCs có thể được cung cấp và truy cập từ cây widget

```
1. class Page extends StatelessWidget{
2.   @override
3.   Widget build(BuildContext context) {
4.     return BlocProvider<NewBloc>(
5.       //Cung cấp NewBloc cho tất cả child trên cây
6.       builder: (BuildContext context) => NewBloc(),
7.       child: Container(
8.         child: BlocBuilder<NewBloc, State>(
9.           //Khởi tạo trạng thái phát ra từ NewBloc được cung cấp gần nhất phí trên
10.          //widget này trong cây. Xây dựng lại child mỗi lần có trạng thái mới từ
11.          BLoC
12.          builder: (context, state) {
13.            //Hiển thị UI dựa trên state phát ra từ NewBLoC
14.            if (state is StateA) return WidgetA();
15.            if (state is StateB) return WidgetB();
16.            else {
17.              //Truy cập newbloc được cung cấp gần nhất ở trên widget này trên cây
18.              //và gửi một event đến đó
19.              BlocProvider.of<NewBloc>(context).dispatch(Event());
20.              return Loading();
21.            }
22.          },
23.        ),
24.      );
25.    }
26.  }
```

làm thế nào BLoCs có thể được cung cấp và truy cập từ cây widget

Minh họa DishBLoC

1. `///Cung cấp khả năng yêu cầu [Dish]es từ [DishBloc].`
2. `/// Là một enum vì nó không cần phải mang theo bất kỳ [DishBloc]`
3. `///vì đã có mọi thứ cần thiết vì nó là truyền đối tượng`
`///của[CurrentSearchBloc] qua tham số`
4. `///khi khởi tạo.`
5. `enum DishEvent { request }`
6. `class DishBloc extends Bloc<DishEvent, DishState> {`
7. `final CurrentSearchBloc _searchBloc;`
8. `///Turns [Search] object into [List<Dish>]`
9. `final Service<Search, List<Dish>> _dishService;`
10. `///Tạo một[DishBloc].`
11. `///`
12. `///Sự phụ thuộc của[_dishService] &`
13. `///[CurrentSearchBloc] được truyền vào qua tham số.`
14. `DishBloc({@required searchBloc, @required dishService})`
15. `: _searchBloc = searchBloc,`
16. `_dishService = dishService;`
17. `@override`
18. `DishState get initialState => InitialDishState();`

Minh họa DishBLoC (tiếp)

```
19. @override
20. Stream<DishState> mapEventToState(DishEvent event) async* {
21.   if(currentState is LoadingDishState) return;
22.   Search currentSearch = searchBloc.currentState.search;
23.   yield LoadingDishState();
24.   try {
25.     yield await _loadDishes (currentSearch);
26.   } catch (e) {
27.     yield ErrorDishState(e.toString());
28.   }
29. }
30. Future<DishState> _loadDishes(Search currentSearch) async {
31.   List<Dish> newState = await _dishService.post(currentSearch);
32.   return ReceivedDishState(newState);
33. }
34. }
```

Minh họa cho trạng thái của Dish Bloc

Trạng thái của Dish BLoC được thực trong đoạn mã sau:

```
1.  /// Mô tả quá trình lấy [Dish] diễn ra như thế nào
2.  @immutable
3.  abstract class DishState extends Equatable {}
4.  @immutable
5.  class InitialDishState extends DishState {
6.      @override
7.      List<Object> get props => [toString()];
8.      @override
9.      String toString() => 'Initial';
10. }
11. @immutable
12. class ReceivedDishState extends DishState {
13.     final List<Dish> dishes;
14.     ReceivedDishState(this.dishes);
15.     @override
16.     List<Object> get props => [dishes];
17.     @override
18.     String toString() => 'Received/NumberOfDishes: '
19.     + dishes.length.toString();
```

Minh họa cho trạng thái của Dish BLoC (tiếp)

```
21. @immutable
22. class ErrorDishState extends DishState {
23.     final String errorMessage;
24.     ErrorDishState(this.errorMessage);
25.     @override
26.     List<Object> get props => [errorMessage];
27.     @override
28.     String toString() => 'Error/Message: ' + errorMessage;
29. }
30. @immutable
31. class LoadingDishState extends DishState {
32.     @override
33.     List<Object> get props => [toString()];
34.     @override
35.     String toString() => 'Loading';
36. }
```

Minh họa cây Widget với trang Menu

Đoạn mã sau đây minh họa cách các BLoCs này được khởi tạo trong Trang Menu

```
1. class MenuPage extends StatefulWidget {
2.   @override
3.   _MenuPageState createState() => _MenuPageState();
4. }
5. class _MenuPageState extends State<MenuPage> {
6.   CurrentSearchBloc _searchBloc = CurrentSearchBloc();
7.   DishBloc _dishBloc;
8.   @override
9.   void initState() {
10.    //Injecting dependencies
11.    _dishBloc = DishBloc(
12.      searchBloc: _searchBloc,
13.      dishService: RepositoryProvider.of<RepositoryBundle>(context)
14.        .dish,
15.      ...
16.    );
17.  }
18. }
```

Minh họa cây Widget với trang Menu (tiếp)

```
1. @override
2. Widget build(BuildContext context) {
3.   return Scaffold(
4.     appBar: Header(),
5.     drawer: AppDrawer(),
6.     body: MultiBlocProvider(
7.       //Cung cấp BLoC cho các con của nó
8.       providers: [
9.         BlocProvider<CurrentSearchBloc>(
10.          builder: (BuildContext context) => _searchBloc,
11.        ),
12.         BlocProvider<DishBloc>(
13.          builder: (BuildContext context) => _dishBloc,
14.        ),
15.      ],
16.      child: CustomScrollView(
17.        // CustomScrollView vì nó chứa widget của các loại khác nhau
18.        slivers: <Widget>[
19.          SliverSearchHeader(),
20.          BlocBuilder<DishBloc, DishState>(
21.            builder: (context, DishState state) {
22.              // Danh sách hiển thị dựa trên State
23.              if (state is ErrorDishState) {
24.                return _error(state);
25.              } else if (state is ReceivedDishState) {
26.                return _list(state);
27.              } else {
28.                return _loading();
29.              }
30.            },
31.          ),
32.          ...
33.        ],
34.      ),
35.    );
36.  }
37. }
```


Minh họa cây Widget với trang Menu (tiếp)

```
38.  /// Hiển thị loading animation trong
    [SliverFillRemaining] Widget _loading() {...}
39.  /// Hiển thị thông báo lỗi được bao bọc trong
    [SliverFillRemaining]
40.  Widget _error(ErrorDishState state) {...}
41.  ///Hiển thị [SliverList] của [DishCard]s
42.  Widget _list(ReceivedDishState state) {
43.      return SliverList(
44.          delegate: SliverChildBuilderDelegate(
45.              (BuildContext context, int index) => DishCard
46.              (
47.                  dish: state.dishes[index],
48.                  childCount: state.dishes.length,
49.              ),
50.          );
51. }
```

Phần còn lại của class DishService

```
16. @override
17. Future<List<Dish>> post(Search input) async {
18.     // Biến Search thành một đối tượng có thể đọc được bằng API
19.     SearchRequest requestBody = SearchRequest.fromSearch(input);
20.     //Send it to the API
21.     http.Response response = await http
22.         .post(
23.             _baseUrl + _route,
24.             headers: _requestHeaders,
25.             body: jsonEncode(requestBody.toJson()),
26.         )
27.         .timeout(Duration(seconds: Configuration.defaultTimeout));
28.
29.     if (response.body == '') throw Exception("No dishes match that query");
30.     Map<dynamic, dynamic> respJson = json.decode(response.body);
31.     SearchResponse searchResponse = SearchResponse.fromJson(respJson);
32.
33.     if (searchResponse.content == null)
34.         throw Exception("Received invalid response from Server");
35.
36.     // Nếu câu trả lời hợp lệ, phản hồi dưới dạng List<Dish>
37.     return searchResponse.toDishList();
38. }
```

Minh họa lớp Model

4.3 Lớp Model

```
1. @immutable
2. class Dish extends Equatable {
3.     final String name;
4.     final String description;
5.     final double price;
6.     /// Image được mã hóa dưới dạng string
7.     final String encodedImage;
8.     /// Vị trí của một asset trong bộ nhớ cục bộ của ứng dụng
9.     final String assetImage;
10.    final int id;
11.    final Map<Extra, bool> extras;
12.
13.    Dish({
14.        @required this.name,
15.        @required this.description,
16.        @required this.price,
17.        @required this.extras,
18.        @required this.id,
19.        this.encodedImage,
20.        this.assetImage,
```

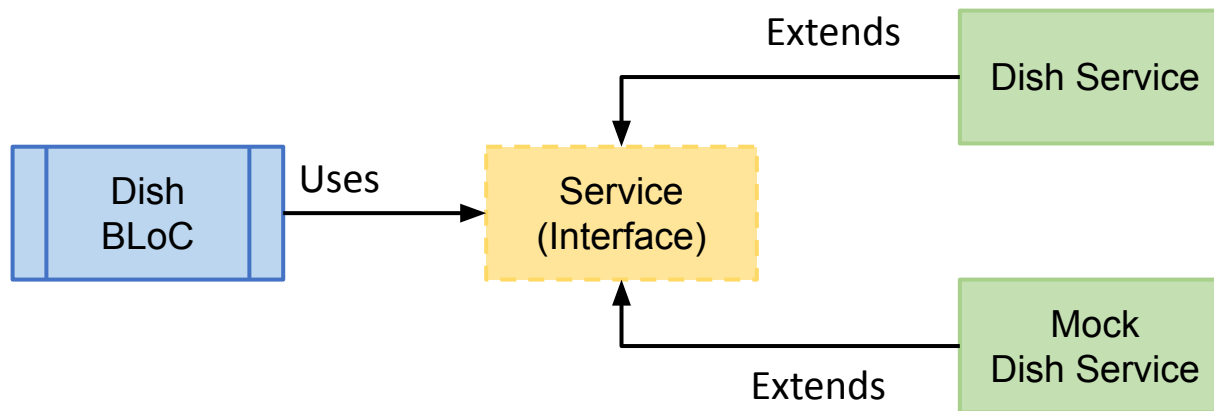
Minh họa lớp Model (tiếp)

```
21. @override
22. List<Object> get props => [name, id, extras, selectedExtras()];
23. @override
24. toString() {
25.     String res = '\'$name\'';
26.     res += hasSelectedExtras() ? ' with ' + selectedExtras() : '';
27.     return res;
28. }
29. bool hasSelectedExtras() => extras.values.contains(true);
30. ///Cung cấp [String] chứa [Extra]s với phân tách dấu phẩy
31. String selectedExtras() {
32.     if (!hasSelectedExtras()) return null;
33.     String res = '';
34.     extras.forEach((extra, picked) => res +=
35.         picked ?
36.         extra.name + ', ' :
37.         '');
38.     );
39.     //Loại bỏ dấu phẩy ở cuối
40.     return res.replaceRange(res.length - 2, res.length - 1, '');
41. }
42. Dish copyWith(...){...}
```

Ví dụ Dependency Injection

Ưu điểm

Trong dự án này, có hai phiên bản Service có thể được tiêm vào Dish BLoC. Dish Service giao tiếp với backend. Trong khi Mock Dish Service, trả về dữ liệu mock cục bộ cho chạy test. Bất kể loại Service nào trong số đó được tiêm vào Dish BLoC, nó hoạt động theo cùng một cách. Ví dụ này được minh họa trong hình dưới đây:



Tóm lại, sử dụng DI trong một dự án mang lại hai lợi thế sau:

1. Mã có thể tái sử dụng hơn.
2. Mã dễ kiểm tra hơn vì phụ thuộc được truyền vào có thể dễ dàng tìm thấy

Ví dụ Dependency Injection (tiếp)

Việc thực hiện Repository Bundle được hiển thị trong đoạn mã sau:

```
1.  /// Tạo và giữ tất cả các repositories của ứng dụng.
2.  ///
3.  /// [RepositoryBundle] sẽ tạo ra tất cả các repositories có liên quan một lần khi tạo
4.  /// và sau đó giữ chúng trong các biến thành viên của nó.
5.  /// [RepositoryBundle] tồn tại để có tất cả các repositories lưu trữ ở một vị trí
6.  /// trung tâm. Nó sẽ được cung cấp trên toàn cục bằng cách sử dụng
7.  /// [RepositoryProvider] vì vậy nó có thể dễ dàng được sử dụng để tiêm dependency vào [Bloc]
8.  /// của ứng dụng.
9.  @immutable
10. class RepositoryBundle {
11.     final Service dish;
12.     final Service booking;
13.     final Service order;
14.
15.     RepositoryBundle({@required bool mock, @required String baseUrl})
16.         : dish = _buildDishRepository(mock, baseUrl),
17.           booking = _buildBookingRepository(mock, baseUrl),
18.           order = _buildOrderRepository(mock, baseUrl);
19.
20.     static Service _buildDishRepository(bool mock, String baseUrl) {
21.         if (mock) {
22.             return MockDishService();
23.         } else {
24.             return DishService(baseUrl: baseUrl);
25.         }
26.     }
27. }
```

Ví dụ Dependency Injection (tiếp)

```
25. static Service _buildBookingRepository(bool mock, String baseUrl) {
26.     if (mock) {
27.         return MockBookingService();
28.     } else {
29.         return BookingService(baseUrl: baseUrl);
30.     }
31. }
32.
33. static Service _buildOrderRepository(bool mock, String baseUrl) {
34.     if (mock) {
35.         return MockOrderService();
36.     } else {
37.         return OrderService(baseUrl: baseUrl);
38.     }
39. }
```

Repository Bundle này quyết định phiên bản nào của repository sẽ được sử dụng cho ứng dụng dựa trên tham số mock mà nó được truyền vào.

Tham số được định nghĩa trong tệp cấu hình, cùng với một số cấu hình chung khác cho ứng dụng.

Ví dụ Dependency Injection (tiếp)

Khi Repository Bundle được khởi tạo, nó được cung cấp trên toàn cục cho phần còn lại của cây widget như được hiển thị trong đoạn mã tiếp theo:

```
1. @override
2. Widget build(BuildContext context) {
3.     return RepositoryProvider<RepositoryBundle>(
4.         //Cung cấp repositories toàn cục
5.         builder: (context) => RepositoryBundle(
6.             mock: Configuration.useMockData,
7.             baseUrl: Configuration.baseUrl,
8.         ),
9.         child: MyThaiStar(), // Tạo phần còn lại của cây widget
10.     );
11. }
```


Ví dụ Dependency Injection (tiếp)

Bất cứ khi nào một BLoC có phụ thuộc vào repository được khởi tạo, widget có liên quan có thể truy cập repository đó và tiêm nó như trong đoạn mã sau:

```
1. class _OrderFormButtonsState extends State<OrderFormButtons> {
2.   OrderBloc _orderBloc;
3.   @override
4.   void initState() {
5.     _orderBloc = OrderBloc(
6.       currentOrderBloc: BlocProvider.of<CurrentOrderBloc>(context),
7.       orderService: RepositoryProvider.of<RepositoryBundle>(context)
8.     ).order,
9.   );
10.   _setUpOrderResponses(_orderBloc);
11.   super.initState();
12.   ...
13. }
```

Minh họa Validator

Gói Form BLoC

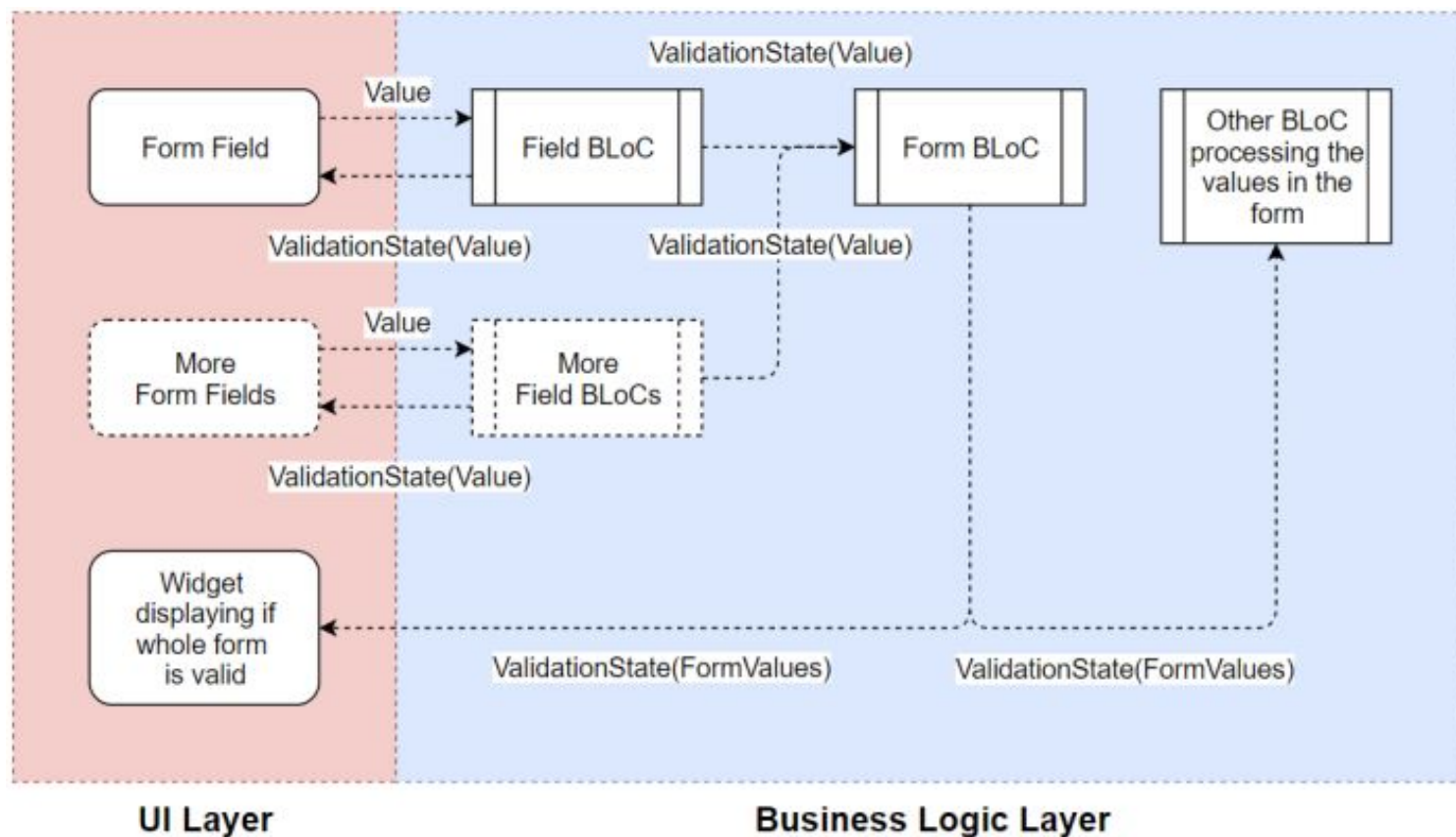
Gói Form BLoC cung cấp hai loại BLoCs: "Field BLoCs" và "Form BLoCs", được mô tả trong bảng sau:

BLoC	Mô tả
Field	<p>Chịu trách nhiệm về hai điều:</p> <ul style="list-style-type: none">• Xác thực nếu đầu vào của Trường Biểu mẫu hợp lệ• Giữ giá trị đã nhập đó. <p>Dùng giá trị được nhập vào Trường Biểu mẫu và phát ra trạng thái mô tả liệu đầu vào hiện tại có hợp lệ hay không và đầu vào đó là gì.</p>
Form	<p>Có một tập hợp các BLoCs trường liên quan. Chịu trách nhiệm về hai điều: khẳng định xem toàn bộ biểu mẫu có hợp lệ hay không và cung cấp tất cả các giá trị hiện đang được nắm giữ bởi mỗi BLoCs Trường liên quan.</p> <p>Phát ra trạng thái mô tả nếu toàn bộ biểu mẫu hợp lệ và giá trị nào hiện có trong biểu mẫu. Trạng thái được phát ra mỗi khi bất kỳ BLoCs trường nào thay đổi trạng thái. Nếu tất cả các BLoCs trường được liên kết là hợp lệ, Mẫu BLoC xem xét toàn bộ biểu mẫu hợp lệ.</p>

Minh họa Validator (tiếp)

Gói Form BLoC

Luồng dữ liệu của các BLoCs được cung cấp bởi gói Form BLoC:

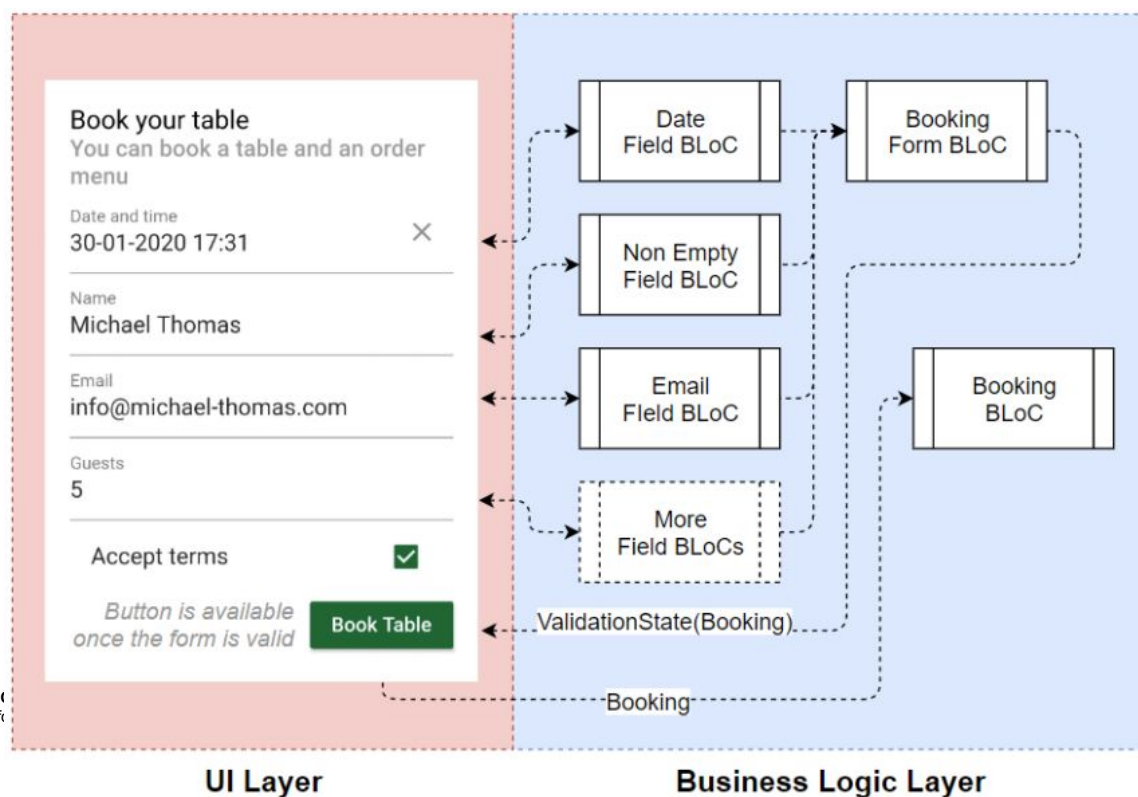


Minh họa Validator (tiếp)

Xác thực biểu mẫu trong Case-study

Hai loại BLoCs được cung cấp bởi gói Form BLoC trong một ví dụ cụ thể: Mẫu Booking của việc thực hiện My Thai Star Flutter. Mẫu đặt chỗ được điền khi người dùng muốn Booking tại nhà hàng My Thai Star.

Mẫu Booking sử dụng gói Form BLoC:



Minh họa Validator (tiếp)

Xác thực biểu mẫu trong Dự án này

Biểu mẫu BLoC mới được tạo cho trường hợp cụ thể. Điều này là do Form BLoCs phát ra tất cả các giá trị hiện tại của biểu mẫu dưới dạng đối tượng. Cách các giá trị đó được biến thành một đối tượng khác nhau ở các trường hợp sử dụng

Mẫu Booking BLoC :

```
1. class BookingFormBloc extends FormBaseBloc<Booking> {
2.   final EmailFieldBloc _emailBloc;
3.   final DateFieldBloc _dateBloc;
4.   final NonEmptyFieldBloc _nameBloc;
5.   final NumberFieldBloc _guestBloc;
6.
7.   BookingFormBloc({
8.     @required emailBloc,
9.     @required dateBloc,
10.    @required nameBloc,
11.    @required guestBloc,
12.    @required termsBloc,
13.  }) : _emailBloc = emailBloc,
14.       _dateBloc = dateBloc,
15.       _nameBloc = nameBloc,
16.       _guestBloc = guestBloc,
17.       super([emailBloc, dateBloc, nameBloc, guestBloc, termsBloc]);
```

Minh họa Validator (tiếp)

```
17. @override
18. ValidationState<Booking> get initialState =>
19.     InitialState(Booking());
20.
21. /// Được gọi mỗi lần một trong [FieldBloc]s
22. /// của trạng thái thay đổi [FormBaseBloc]
23. @override
24. Stream<ValidationState<Booking>> mapEventToState(FormEvent event)
25.     async* {
26.         ...
27.         if (isFormValid()) {
28.             yield ValidState(Booking(
29.                 name: _nameBloc.currentState.data,
30.                 organizerEmail: _emailBloc.currentState.data,
31.                 date: date,
32.                 guests: guests,
33.             ));
34.         } else {
35.             yield InvalidState(Booking(
36.                 name: _nameBloc.currentState.data,
37.                 organizerEmail: _emailBloc.currentState.data,
38.                 date: date,
39.                 guests: guests,
40.             ));
41.         }
42.     }
```

Minh họa Validator (tiếp)

Cách tất cả các BLoCs về xác thực biểu mẫu cho Booking được khởi tạo trong lớp UI được hiển thị trong đoạn mã sau:

```
1. //Xác thực: FieldBlocs
2. EmailFieldBloc _emailBloc = EmailFieldBloc();
3. DateFieldBloc _dateBloc = DateFieldBloc(DateFormat('dd-MM-yyyy HH:mm'));
4. NonEmptyFieldBloc _nameBloc = NonEmptyFieldBloc();
5. NumberFieldBloc _guestBloc = NumberFieldBloc();
6. CheckboxFieldBloc _termsBloc = CheckboxFieldBloc();
7. // Xác thực : FormBuilder
8. YourFormBloc _formBloc;
9. @override
10. void initState() {
11.   _formBloc = YourFormBloc(
12.     emailBloc: _emailBloc,
13.     dateBloc: _dateBloc,
14.     nameBloc: _nameBloc,
15.     guestBloc: _guestBloc,
16.     termsBloc: _termsBloc,
17.   );
18.   super.initState();
19. }
20. ...
```

Minh họa Validator (tiếp)

Tiện ích trường biểu mẫu thực hiện gói này được hiển thị trong đoạn mã sau. Xác thực không còn diễn ra trong giao diện người dùng. Mỗi Trường Biểu mẫu có một Trường BLoC chuyên dụng chịu trách nhiệm xác thực.

```
1.  BlocBuilder<FieldBloc, ValidationState>(  
2.    bloc: _formFieldBloc,  
3.    builder: (context, ValidationState state) {  
4.      return TextFormField(  
5.        decoration: InputDecoration(  
6.          labelText: _label,  
7.          errorText: validate(state),  
8.        ),  
9.        onChanged: (String input) => _formFieldBloc.dispatch(input),  
10.      );  
11.    },  
12.  );  
13.  String validate(ValidationState state) {  
14.    if (state is InvalidState)  
15.      return _errorHint;  
16.    else  
17.      return null;  
18.  }  
19.
```


Minh họa bản địa hóa

```
1. class MyThaiStar extends StatelessWidget {
2.   static const String title = 'My Thai Star';
3.
4.   @override
5.   Widget build(BuildContext context) {
6.     return RootProvider(
7.       child: BlocBuilder<LocalizationBloc, Locale>(
8.         builder: (context, locale) => MaterialApp(
9.           // Xây dựng lại bất cứ khi nào Localization BLoC phát ra trạng thái mới
10.          title: title,
11.          theme: MtsTheme.data,
12.          locale: locale, //Update locale
13.          initialRoute: Router.home,
14.          onGenerateRoute: (RouteSettings settings) =>
15.            Router.generateRoute(settings),
16.          localizationsDelegates: _buildLocalizationDelegates(),
17.          supportedLocales: MtsLocalizationDelegate.supportedLanguages
18.            .map((code) => Locale(code))
19.            .toList(),
20.        ),
21.      );
22.   }
23.   ///Xây dựng các [LocalizationsDelegate].
24.   ///
25.   /// Danh sách này chứa cả [LocalizationsDelegate] riêng của Flutter cho widget, v.v. và
26.   /// đại biểu cụ thể của My Thai Star.
27.   Iterable<LocalizationsDelegate<dynamic>> _buildLocalizationDelegates() => [
28.     GlobalMaterialLocalizations.delegate,
29.     GlobalWidgetsLocalizations.delegate,
30.     // Đây là đại biểu cụ thể của My Thai Star để xử lý văn bản được bản địa hóa
31.     MtsLocalizationDelegate(),
32.   ];
33. }
```

Minh họa bản địa hóa (tiếp)

Lớp Translation

```
1.  /// Giữ một bản dịch của tất cả các văn bản My Thai Star cho một văn bản được
    ///đưa ra [Locale]
2.  @immutable
3.  class Translation {
4.      ///JSON giữ các bản dịch
5.      final Map<dynamic, dynamic> _translationMap;
6.      final Locale locale;
7.
8.      const Translation(this.locale, this._translationMap);
9.
10.     String get languageCode => locale.languageCode;
11.
12.     /// Trả về giá trị trong [_translationMap] của một đường dẫn đã cho
13.     String get(String path) {
14.         String result;
15.         try {
16.             List<String> query = path.split('/');
17.             result = _findInMap(_translationMap, query);
18.         } catch (e) {
19.             result = 'Did not find \'$path\'';
20.         }
21.         if (result == null) result = 'Did not find \'$path\'';
22.         return result;
23.     }
24. }
```

Minh họa bản địa hóa (tiếp)

```
21.    /// Cho một giá trị JSON khớp với [query] đã cho.
22.    ///
23.    /// Tìm kiếm đệ quy thông qua map.
24.    String _findInMap(Map<dynamic, dynamic> map, List<String> query) {
25.    if (query.length == 1) {
26.        return map[query.first];
27.    } else {
28.        return _findInMap(map[query.first], query.sublist(1, query.length));
29.    }
30.    }

31.    /// Cho phép truy cập ngắn hơn [Translation]
32.    static Translation of(BuildContext context) =>
33.        Localizations.of<Translation>(context, Translation);
34.    }
```

Minh họa bản địa hóa (tiếp)

Hàm "of" rút ngắn lời gọi cần thiết để truy cập đối tượng Translation hiện tại.

Chức năng "get" cung cấp một giao diện đơn giản để truy cập dữ liệu trong map JSON được cung cấp bởi đối tượng Translation.

Sự khác biệt về kích thước trong lời gọi được làm rõ bằng đoạn mã sau.

1. `Localizations.of<Translation>(context, Translation);`
`//Without "of"`
2. `Translation.of(context);`
`//With "of"`
3. `Translation.of(context).map['menu']['header']['title'];`
`//Without "get"`
4. `Translation.of(context).get('menu/header/title');`
`//With "get"`
5. `Localizations.of<Translation>(context, Translation)`
`//No helpers`
6. `.map['menu']['header']['title'];`
7. `Translation.of(context).get('menu/header/title');`