

State Management



Nội dung

- State management
- provider package
- GetX

Giới thiệu pub.dev

- Trang web cung cấp các package hỗ trợ cho việc phát triển ứng dụng Flutter.
- Url: pub.dev
- Một số package/plugin:
 - equatable: Hỗ trợ việc so sánh các đối tượng trong Dart bằng toán tử "=="
 - provider: Quản lý app state
 - GetX: Quản lý app state và cung cấp rất nhiều tiện ích khác
 - BLoC: Quản lý app state
 - shared_preferences: Thư viện giúp đọc ghi các cặp key-value đơn giản.
 - flutter_local_notifications: platform đa nền tảng để hiển thị các thông báo local.
 - firebase_core: Thư viện cho phép kết nối nhiều ứng dụng với firebase

State Management



Flutter state

- Giao diện Flutter được xây dựng theo kiểu khai báo. Điều này có nghĩa là Flutter xây dựng giao diện người dùng của mình để phản ánh trạng thái hiện tại của ứng dụng của bạn.
- Khi state thay đổi → Giao diện người dùng được xây dựng lại từ đầu
- Hai trạng thái của Flutter: Trạng thái tạm thời và trạng thái ứng dụng

$$\text{UI} = f(\text{state})$$

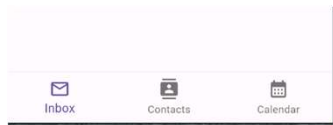
The layout on the screen Your build methods The application state

Trạng thái tạm thời

- Trạng thái tạm thời (Còn được gọi là trạng thái UI hay trạng thái cục bộ): Trạng thái có thể được chứa gọn gàng trong một widget. Ví dụ:
 - Trang hiện tại trong một PageView.
 - Tiến trình hiện tại trong một hoạt ảnh phức tạp.
 - Tab hiện tại được chọn trong BottomNavigationBar.
- Không cần phải sử dụng các kỹ thuật quản lý trạng thái đối với các loại trạng thái này. *Tất cả những gì cần thiết chỉ là một StatefulWidget.*
- Trạng thái tạm thời có thể được cài đặt bằng cách sử dụng `State` và phương thức `setState()`.

Trạng thái tạm thời

- Ví dụ:
 - `_index` là một trạng thái tạm thời, chỉ tồn tại trong `MyHomepage`. Các thành phần khác của ứng dụng không cần phải truy cập đến biến `_index` này.
 - `_index` được khởi tạo lại giá trị 0 khi `MyHomePage` được gọi.



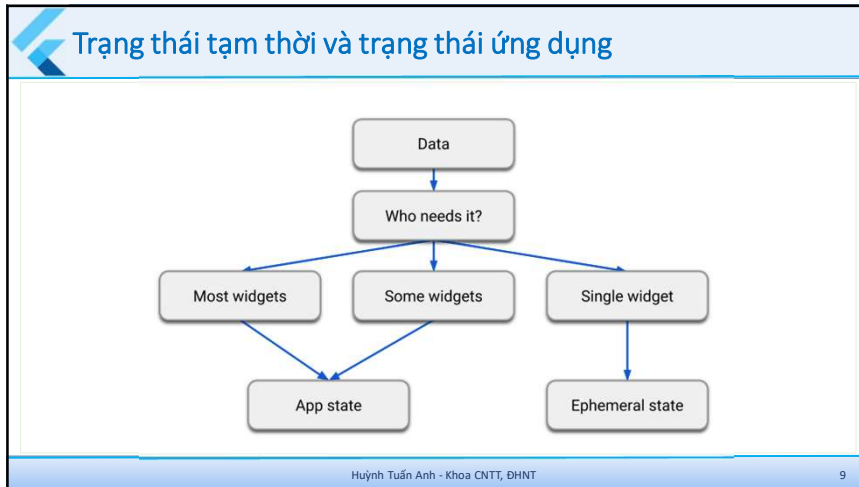
```
class MyHomepage extends StatefulWidget {
  @override
  _MyHomepageState createState() => _MyHomepageState();
}

class _MyHomepageState extends State<MyHomepage> {
  int _index = 0;

  @override
  Widget build(BuildContext context) {
    return BottomNavigationBar(
      currentIndex: _index,
      onTap: (newIndex) {
        setState(() {
          _index = newIndex;
        });
      },
      // ... items ...
    );
  }
}
```

Trạng thái ứng dụng (App State)

- Là trạng thái không phải tạm thời và được chia sẻ giữa các thành phần khác nhau, các trang của ứng dụng và có thể được lưu giữ giữa các phiên sử dụng của người dùng.
- Ví dụ:
 - Sở thích của người dùng
 - Thông tin đăng nhập
 - Thông báo trong ứng dụng mạng xã hội
 - Giỏ hàng trong ứng dụng thương mại điện tử
 - Trạng thái đọc và chưa đọc của các bài báo trong một ứng dụng tin tức.



Quản lý các trạng thái

- Trong Flutter, bạn nên giữ trạng thái trên các widget sử dụng nó vì:
 - Trong các Framework khai báo như Flutter, nếu muốn thay đổi giao diện thì phải xây dựng lại nó.
 - Khó thay đổi một widget từ bên ngoài bằng cách gọi một phương thức trên nó.** Nếu có thể làm được như vậy thì có khả năng sẽ xung đột với Flutter Framework thay vì tận dụng sự trợ giúp của Framework này.
 - Trong Flutter, **một Widget mới sẽ được tạo ra khi nội dung của nó thay đổi.** Do đó thay vì truyền trạng thái cần cập nhật cho đối số của một hàm cập nhật nào đó, **ta nên sử dụng hàm khởi tạo với đối số là trạng thái cần cập nhật.**

Huỳnh Tuấn Anh - Khoa CNTT, ĐHNT

Một số cách tiếp cận

- Một số Framework để quản lý các trạng thái (<https://pub.dev/packages/>):
 - Provider** (khuyến dùng vì do chính google phát triển)
 - setState**: Được sử dụng cho các trạng thái tạm thời của một widget cụ thể.
 - InheritedWidget & InheritedModel**: Cách tiếp cận ở mức thấp được sử dụng để giao tiếp giữa các widget tổ tiên và các widget con cháu trong cây.
 - GetX**: Được đánh giá cao trên Pub.dev, các công ty ở VN thường sử dụng.
 - BLoC / Rx**: Một họ các mẫu Stream/Observable

```

graph TD
    MyApp[MyApp] --> MyCatalog[MyCatalog]
    MyApp --> MyCart[MyCart]
    MyCatalog --> MyAppBar[MyAppBar]
    MyCatalog --> MyListItem1[MyListItem]
    MyCatalog --> MyListItem2[MyListItem]
    MyCatalog --> MyListItem3[MyListItem]
    MyCart -- CART --> MyApp
  
```

Huỳnh Tuấn Anh - Khoa CNTT, ĐHNT

provider package

Huỳnh Tuấn Anh - Khoa CNTT, ĐHNT

Giới thiệu chung về provider

- Một trình bao bọc quanh **InheritedWidget** để giúp chúng dễ sử dụng hơn và có thể tái sử dụng nhiều hơn.
- Đơn giản hóa việc phân bổ và xử lý tài nguyên.
- lazy-loading
- Khuôn mẫu để giảm việc tạo các lớp mới.
- Công cụ phát triển thân thiện.
- Một cách phổ biến để sử dụng các InheritedWidget
 - Ba khái niệm: (**ChangeNotifier** / **ChangeNotifierProvider** / **Consumer** hay **Selector**)
- Tăng khả năng mở rộng cho các lớp có cơ chế lắng nghe phát triển theo độ phức tạp cấp số nhân

provider package

- Để sử dụng gói thư viện provider, cần phải khai báo nó trong tập tin pubspec.yaml.

```
dependencies:
  provider: ^4.3.2+2
```

- Ba khái niệm trong provider:
 - **ChangeNotifier**: Cung cấp cách thức để gói trạng thái của ứng dụng.
 - **ChangeNotifierProvider**: Là một widget cung cấp một thể hiện của một **ChangeNotifier** cho các con của nó là các **Consumer**.
 - **Consumer**: Sử dụng đối tượng **ChangeNotifier** do **ChangeNotifierProvider** cung cấp.
 - **Selector**: Tương đương với Consumer có thể lọc các cập nhật bằng cách chọn một số các giá trị giới hạn và *ngăn chặn rebuild nếu chúng không thay đổi*.

ChangeNotifier

- Là một lớp đơn giản trong Flutter SDK cung cấp sự thông báo thay đổi cho thành phần listener của nó. Nói cách khác, nếu thứ gì đó là ChangeNotifier, bạn có thể đăng ký để lắng nghe các thay đổi của nó (Observer Pattern).
- Trong provider, **ChangeNotifier** là một cách để đóng gói trạng thái ứng dụng của bạn. Đối với các ứng dụng phức tạp, sẽ có nhiều **ChangeNotifier** tương ứng với nhiều model.
- **notifyListeners()**: Gọi phương thức này khi mô hình của ứng dụng thay đổi và bạn muốn UI của ứng dụng cũng thay đổi theo.
- Cấu trúc thông thường của một lớp ChangeNotifier:
 - Thuộc tính state (của ứng dụng)
 - Các phương thức thay đổi state, trong phương thức này gọi notifyListeners() sau khi state bị thay đổi

ChangeNotifierProvider

- Là Widget cung cấp một instance của **ChangeNotifier** cho các con, là các **Consumer**, của nó.
- Nơi đặt ChangeNotifierProvider trong widget tree: Phía trên widget cần truy cập đến nó. *Không nên đặt ChangeNotifierProvider quá cao trong widget tree.*
 - context của ChangeNotifierProvider phải "cao hơn" context của Consumer.

```
class CounterProviderApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider(
      create: (context) => Counter(),
      child: MaterialApp(
        title: "Provider Demo",
        home: CounterPage(),
      ),
    );
  }
}
```

MultiProvider

- Một Provider hợp nhất nhiều Provider thành một single linear widget tree. Nó được sử dụng để cải thiện khả năng đọc và giảm mã lệnh sẵn có của việc phải lồng nhiều lớp của các Provider.

```
void main() {
  runApp(
    MultiProvider(
      providers: [
        ChangeNotifierProvider(create: (context) => CartModel()),
        Provider(create: (context) => SomeOtherClass()),
      ],
      child: MyApp(),
    ),
  );
}
```

Các dạng Provider

- Provider
- ChangeNotifierProvider
- ChangeNotifierProxyProvider: Một ChangeNotifierProvider build và đồng bộ một ChangeNotifier với các giá trị bên ngoài

```
ChangeNotifierProxyProvider<MyModel, MyChangeNotifier>(
  create: (_) => MyChangeNotifier(),
  update: (_, myModel, myNotifier) => myNotifier
    ..update(myModel),
  child: ...
);
```

- Nếu MyModel được cập nhật thì MyChangeNotifier sẽ được cập nhật một cách tự động.

Consumer

- Sử dụng đối tượng **ChangeNotifier** do **ChangeNotifierProvider** cung cấp.
- Cần phải chỉ định kiểu của model cần truy cập.
- builder**: là một hàm trong Consumer nhận 3 đối số và trả về một Widget (thường hiển thị trạng thái được gói trong ChangeNotifier.
 - context
 - Tham số thứ hai: là một thể hiện của ChangeNotifier
 - Tham số thứ ba, **child**: là một subtree bên dưới Consumer không thay đổi khi model thay đổi.

Selector

- Tương đương với Consumer có thể lọc các cập nhật bằng cách chọn một số các giá trị giới hạn và ngăn chặn rebuild nếu chúng không thay đổi.
- Selector sẽ nhận được một giá trị bằng cách sử dụng Provider.of, sau đó chuyển giá trị đó cho selector. Selector callback này sau đó có nhiệm vụ trả về một đối tượng chỉ chứa thông tin cần thiết để buider hoàn thành.
- Theo mặc định, Selector xác định xem builder có cần được gọi lại hay không bằng cách so sánh kết quả trước đó và kết quả mới của bộ chọn bằng cách sử dụng DeepCollectionEquality từ bộ package selection.
 - Việc này có thể được ghi đè bằng cách viết callback cho tham số shouldRebuild

Selector

- Để chọn nhiều giá trị mà không cần phải viết một lớp thực thi ==, giải pháp đơn giản nhất là sử dụng "Tuple" từ [package tuple](https://pub.dev/packages/tuple) (<https://pub.dev/packages/tuple>):

```
Selector<Foo, Tuple2<Bar, Baz>>(
  selector: (_, foo) => Tuple2(foo.bar, foo.baz),
  builder: (_, data, _) {
    return Text('${data.item1} ${data.item2}');
  }
)
```

Kiểu đối tượng Kiểu giá trị được chọn trên đối tượng

Đối tượng Dữ liệu được chọn trên đối tượng

Huỳnh Tuấn Anh - Khoa CNTT, ĐHNT 21

Provider.of<T>(BuildContext context, {bool listen: true})

- Lấy Provide <T> gần nhất ở trên widget tree của nó và trả về giá trị của nó.
- listen: true**: các thay đổi giá trị sau này sẽ kích hoạt State.build mới cho các widget và State.didChangeDependencies cho StatefulWidget.
- Được sử dụng khi không cần phải thay đổi UI khi dữ liệu trong mô hình thay đổi (listen:false), và người dùng vẫn cần truy cập dữ liệu. Ví dụ: nút ClearCart cho phép người dùng xóa mọi thứ ra khỏi giỏ hàng và không cần hiển thị nội dung của giỏ hàng, chỉ cần gọi phương thức removeAll().
- Sử dụng Consumer trong trường hợp này có thể gây lãng phí bởi vì chúng ta sẽ yêu cầu framework rebuild lại những widget mà không cần rebuild.
- Trong trường hợp này, ta có thể sử dụng Provider.of

```
Provider.of<CartModel>(context, listen: false).removeAll();
```

Huỳnh Tuấn Anh - Khoa CNTT, ĐHNT 22

Ví dụ

Flutter Demo Home Page

You have pushed the button this many times:
4

+

```

graph TD
    ChangeNotifierProvider --> MyApp
    MyApp --> MaterialApp
    MaterialApp --> MyHomePage["MyHomePage (Stateless)"]
    MyHomePage --> Scaffold
    Scaffold --> AppBar
    Scaffold --> Center
    Scaffold --> FloatingActionButton
    Center --> Column
    Column --> Text1[Text]
    Column --> Consumer
    Consumer --> Text2[Text]
    FloatingActionButton --> Text3[Text]
    Text3 -.->|Thay đổi| Text2
  
```

Huỳnh Tuấn Anh - Khoa CNTT, ĐHNT 23

Ví dụ: Counter App viết theo Provider

- Sử dụng ChangeNotifier để gói trạng thái của ứng dụng

```

class Counter extends ChangeNotifier {
  int _value = 0;
  int get value => _value;
  void increment() {
    _value++;
    notifyListeners();
  }
}
  
```

Trạng thái của ứng dụng

Huỳnh Tuấn Anh - Khoa CNTT, ĐHNT 24

ChangeNotifierProvider

```
void main() {
  runApp(
    ChangeNotifierProvider(
      create: (context) => Counter(),
      child: MyApp(),
    ),
  );
}
```

Cung cấp model cho tất cả các Widget trong ứng dụng

Huỳnh Tuấn Anh - Khoa CNTT, ĐHNT 25

Consumer

```
floatingActionButton: FloatingActionButton(
  onPressed: () {
    var counter = context.read<Counter>();
    counter.increment();
  },
  tooltip: 'Increment',
  child: Icon(Icons.add),
),
```

```
Consumer<Counter>(
  builder: (context, counter, child) {
    return Text('${counter.value}',
      style: Theme.of(context).textTheme.headline4,);
  },
),
```

Huỳnh Tuấn Anh - Khoa CNTT, ĐHNT 26

Consumer

- **child**: Tham số dùng để tối ưu hóa. Tham số này dùng để "chứa" các widget không cần phải rebuild khi state của ứng dụng thay đổi

```
Consumer<Counter>(
  builder: (context, counter, child) => Column(
    children: [
      Text('${counter.value}',
        style: Theme.of(context).textTheme.headline4,),
      child,
    ],
  ),
  child: Text("This is not rebuild widget"),
),
```

Huỳnh Tuấn Anh - Khoa CNTT, ĐHNT 27

Selector

```
Selector<Counter, int>(
  selector: (context, data) => data.value,
  shouldRebuild: (pre, next) => next <= 10,
  builder: (context, counter, child) => Column(
    children: [
      Text(
        '$counter',
        style: Theme.of(context).textTheme.headline4,
      ),
      child,
    ],
  ),
  child: Text("This is not rebuild widget"),
),
```

Huỳnh Tuấn Anh - Khoa CNTT, ĐHNT 28

ChangeNotifierProvider

- Nếu cần cung cấp nhiều hơn một provider, sử dụng MultiProvider

```
void main() {
  runApp(
    MultiProvider(
      providers: [
        ChangeNotifierProvider(create: (context) => Counter() ),
        //Other Providers
      ],
      child: MyApp() );
}
```

Các extension trên BuildContext của Provider

- WatchContext: phương thức: `T watch<T>()`
 - Nhận một giá trị từ một provider tổ tiên gần nhất thuộc loại [T] và đã đăng ký với provider.
 - Gọi phương thức này tương đương với gọi pt `Provider.of<T>(context)`.
 - Phương thức này chỉ được truy cập bên trong `StatelessWidget.build` và `State.build`. Nếu truy cập bên ngoài phạm vi các phương thức build này, sử dụng `Provider.of` để thay thế và sẽ không có các hạn chế này.
 - Nếu giá trị nhận được thay đổi (phương thức `notifyListeners()` được gọi) thì widget sẽ được rebuild
 - Không được gọi bên trong các phương thức build nếu các giá trị chỉ được sử dụng cho các events

Các extension trên BuildContext của Provider

- SelectContext: Phương thức: `R select<T, R>(R selector(T value))`
 - Xem (watch) một giá trị kiểu T từ một provider và chỉ lắng nghe một phần (dạng R) các thay đổi.
 - select phải được sử dụng bên trong phương thức build của một widget*, Nó sẽ không hoạt động trong các pt vòng đời khác, bao gồm `State.didChangeDependencies`.
 - Khi sử dụng select, thay vì xem toàn bộ đối tượng, trình lắng nghe sẽ chỉ rebuild nếu giá trị được trả về bởi selector thay đổi.
 - Khi một provider phát bản cập nhật, nó sẽ gọi đồng bộ tất cả các selector. Sau đó, nếu chúng trả về một giá trị khác với giá trị được trả về trước đó, thì phần phụ thuộc sẽ được đánh dấu là cần rebuild.
 - Các phương thức `watch()`; `Provider.of<T>(context)` không thể chỉ lắng nghe một phần đối tượng đã đăng ký với Provider.

Các extension trên BuildContext của Provider

- SelectContext: Phương thức: `R select<T, R>(R selector(T value))` tt...
 - Để xác định được một đối tượng thay đổi giá trị, các phương thức `==` và phương thức `hashCode` được sử dụng của đối tượng đó được sử dụng.
 - Ta cần override hai phương thức này trong một số trường hợp.
 - Giải pháp khác: Sử dụng gói: `equatable` trên `pub.dev`. Khi đó ta chỉ cần định nghĩa lớp extends lớp `Equatable`

```
class Credentials extends Equatable {
  const Credentials({this.username, this.password});
  final String username;
  final String password;

  @override
  List<Object> get props => [username, password];
}
```


Các extension trên BuildContext của Provider

- ReadContext: Phương thức `T read<T>()`
 - Nhận một giá trị từ provider tổ tiên gần nhất thuộc loại [T].
 - Không làm cho widget rebuild và không được gọi bên trong các phương thức `StatelessWidget.build/State.build`.
 - Có thể gọi bên ngoài những phương thức này.
 - Nếu không phù hợp với điều kiện sử dụng thì có thể sử dụng `Provider.of(context, listen: false)` để thay thế và cho kết quả tương tự nhưng không có những hạn chế trên.
 - KHÔNG gọi read bên trong build nếu giá trị chỉ được sử dụng cho các sự kiện.
 - Có thể gọi read bên trong các event handler (ví dụ như các sự kiện `onPressed`)
 - KHÔNG sử dụng read để tạo widget có giá trị không bao giờ thay đổi. Thay vào đó, hãy sử dụng `select` để lấy giá trị không thay đổi có đối tượng được cung cấp bởi provider.

Huỳnh Tuấn Anh - Khoa CNTT, ĐHNT 33

Bài tập

Huỳnh Tuấn Anh - Khoa CNTT, ĐHNT 34


Bài tập

- Yêu cầu:
 - Dữ liệu lưu thành file .json trong thư mục của ứng dụng
 - Version 1: Sử dụng `setState` để quản lý trạng thái.
 - Version 2: Sử dụng Provider để quản lý trạng thái.
 - Do đọc, ghi dữ liệu sử dụng các phương thức async nên các provider được sử dụng là: `MultiProvider`, `FutureProvider`, `ChangeNotifierProxyProvider`


Huỳnh Tuấn Anh - Khoa CNTT, ĐHNT 35

GetX

Huỳnh Tuấn Anh - Khoa CNTT, Đại Học Nha Trang 36



GetX State Manager | Navigation Manager
Dependencies Manager
Fast, Stable, Extra-light and Powerful Flutter Framework



**Không nên thần thánh hóa một API nào cả.
Nếu quá phụ thuộc vào nó bạn sẽ chết vì nó**

Huỳnh Tuấn Anh - Khoa CNTT, Đại Học Nha Trang 37

Giới thiệu về GetX

- GetX là một giải pháp "nhẹ" và mạnh mẽ cho Flutter. GetX tích hợp việc quản lý trạng thái hiệu suất cao, quản lý phụ thuộc thông minh, và quản lý điều hướng một cách nhanh chóng.
- 3 nguyên tắc cơ bản của GetX:
 - PERFORMANCE:** Tập trung vào hiệu năng và giảm thiểu việc sử dụng các tài nguyên
 - PRODUCTIVITY:** Sử dụng cú pháp thân thiện nhưng mang lại hiệu quả tối đa cho ứng dụng. Không cần phải xóa controller, GetX sẽ làm công việc này. Tuy nhiên, các controller cũng có thể được giữ lại lâu dài trong bộ nhớ bằng các khai báo đơn giản, "permanent: true" in trong dependency của bạn.
 - ORGANIZATION:** GetX allows the total decoupling of the View, presentation logic, business logic, dependency injection, and navigation. You do not need context to navigate between routes, so you are not dependent on the widget tree (visualization) for this.
- Các tính năng của GetX được tách biệt nhau và chỉ được khởi chạy sau khi nó được sử dụng. Có nghĩa là, các tính năng chỉ được biên dịch theo app nếu nó được sử dụng.




Huỳnh Tuấn Anh - Khoa CNTT, Đại Học Nha Trang 38

Các tính năng chính của GetX

- 3 tính năng chính
 - State Management
 - Route Management
 - Dependency Management
- Các tiện ích
 - Internationalization
 - Change Theme
 - GetConnect: cung cấp cách thức dễ dàng để giao tiếp giữa back end và front end bằng http hoặc websockets
 - GetPage Middleware
 - ...

Huỳnh Tuấn Anh - Khoa CNTT, Đại Học Nha Trang 39

State Management

Huỳnh Tuấn Anh - Khoa CNTT, Đại Học Nha Trang 40

State Management

- GetX có hai trình quản lý trạng thái khác nhau:
 - Trình quản lý trạng thái đơn giản: GetBuilder
 - Trình quản lý trạng thái phản ứng (reactive state manager): GetX/Obx
- Obx: Được sử dụng khi cập nhật tất cả các widget liên quan tới một observed state.
- GetX: Được sử dụng khi chỉ cần cập nhật một observed state cho một số widget cụ thể thông qua các tag.
- GetBuilder: Sử dụng khi chỉ muốn cập nhật một số ít các widget liên quan đến một trạng thái thay đổi thông qua các id. Việc cập nhật chỉ được thực hiện thông qua việc gọi hàm update.

Reactive State Manager

- GetX cho phép lập trình react theo cách đơn giản:
 - Không cần phải tạo StreamControllers
 - Không cần phải tạo một StreamBuilder cho mỗi biến
 - Không cần phải tạo một lớp cho mỗi state
 - Không cần phải tạo một thuộc tính get cho một giá trị khởi tạo
- Lập trình react trong GetX chỉ đơn giản gồm 2 bước:
 - Khai báo các biến reactive (**reactive variable**)
 - Truy xuất giá trị được bọc trong biến reactive: `tên_biến.value`
 - Sử dụng các giá trị của các biến react trên View
- Một biến reactive có thể xem tương đương với một Stream. Khi giá trị của biến thay đổi thì các thành phần phụ thuộc (thành phần sử dụng biến này) sẽ được thay đổi theo.
 - Bản chất của reactive state manager chính là quản lý các Stream

3 cách khai báo biến reactive

- Cách 1: Sử dụng Rx{Type}. Nên khởi tạo giá trị cho biến, nhưng không bắt buộc
 - `final name = RxString("");`
 - `final isLoggedIn = RxBool(false);`
 - `final count = RxInt(0);`
 - `final balance = RxDouble(0.0);`
 - `final items = RxList<String>([]);` // hoặc `final items = RxList<String>();`
 - `final myMap = RxMap<String, int>({});`

3 cách khai báo biến reactive

- Cách 2: Sử dụng Rx và sử dụng Darts Generics, Rx<Type>
 - `final name = Rx<String>("");`
 - `final isLoggedIn = Rx<Bool>(false);`
 - `final count = Rx<Int>(0);`
 - `final balance = Rx<Double>(0.0);`
 - `final number = Rx<Num>(0);`
 - `final items = Rx<List<String>>([]);`
 - `final myMap = Rx<Map<String, int>>({});`
 - // Custom classes - it can be any class, literally
 - `final user = Rx<User>();`

3 cách khai báo biến reactive

- Cách 3: Cách tiếp cận dễ dàng hơn và ưa thích hơn, chỉ cần thêm .obs (Observer) làm thuộc tính giá trị của biến:

- final name = "".obs;
 - final isLoggedIn = false.obs;
 - final count = 0.obs;
 - final balance = 0.0.obs;
 - final number = 0.obs;
 - final items = <String>[].obs;
 - final myMap = <String, int>{}.obs;
- Custom classes - it can be any class, literally
- final user = User().obs;

Sử dụng các biến reactive trên view

- Bước 1: Viết Controller, class mở rộng từ lớp GetXController. Controller có nhiệm vụ thực hiện các xử lý logic và quản lý các state cho các route

```
class Controller extends GetXController{
    final count1 = 0.obs;
    final count2 = 0.obs;
    int get sum => count1.value + count2.value;
    increment(){
        count1.value++;
        count2.value++;
    }
}
```

- Phương thức **refresh**: Trong trường hợp các biến reactive có kiểu là các Custome Type, việc thay đổi giá trị sẽ không làm *UI cập nhật*. Phương thức refresh sẽ đưa giá trị của biến reactive vào Stream và làm cho UI cập nhật. Ví dụ: `count1.refresh()`;

Sử dụng các biến reactive trên view

- Bước 2: Sử dụng Controller trên view

- Mỗi đối tượng Controller được tạo trên view thường là một singleton được truy cập ở mức toàn cục, mỗi controller có thể có một giá trị tag (giá trị tag là một tùy chọn nếu có nhiều controller được tạo ứng với một lớp Controller đã cài đặt).
- Tạo Controller nếu controller đã được tạo thì lấy controller này:
 - final Controller c = Get.put(Controller());
 - final Controller c = Get.put(Controller(), tag: "my_controller");
- Lấy một controller đã được tạo
 - final Controller c = Get.find();
 - final Controller c = Get.find(tag: "my_controller");
- Chú ý: Nếu controller chưa được tạo ra, khi gọi phương thức find() ==> Lỗi

Sử dụng các biến reactive trên view

- GetX Widget

- Ví dụ:

```
GetX<Controller>(<
    tag: "my_controller",
    builder: (controller) => Text("${controller.count1.value}"),
), // GetX
```

- Obx Widget

- Ví dụ:

```
Obx(() => Text("${c.count2.value}")),
```

- Gọi một phương thức của controller:

- Ví dụ: `c.increment()`;

Simple State Manager – Ưu điểm

- Chỉ cập nhật đúng các widget theo yêu cầu
- Không sử dụng `changeNotifier`, là cách quản lý trạng thái ít tốn bộ nhớ
- Không cần phải sử dụng đến các `StatefulWidget`. Chỉ cần tạo một `StatelessWidget`, nếu cần cập nhật một component, chỉ cần bọc nó trong `GetBuilder`
- Tổ chức dự án thực tế hơn, các bộ điều khiển không được đặt trong UI, hãy đặt `TextEditingController`, hay bất kỳ controller nào trong lớp `Controller` của bạn
- Không cần phải phát sinh một sự kiện để cập nhật một widget. Thuộc tính `initState` đóng vai trò như một `StatefulWidget`, và bạn có thể gọi các sự kiện trực tiếp từ controller, không cần phải đặt các sự kiện đặt trong `initState`.
- Không cần phải phát sinh các action như đóng stream, timers... Chỉ cần gọi các event trên `callback dispose` của `GetBuilder` ngay khi widget bị hủy

Simple State Manager – Ưu điểm

- Chỉ sử dụng stream nếu cần thiết. Có thể sử dụng `StreamController` một cách bình thường bên trong controller, và sử dụng `StreamBuilder` một cách bình thường, nhưng chú ý rằng việc sử dụng stream sẽ tốn kém bộ nhớ, lập trình reactive rất đẹp nhưng không nên lạm dụng nó. 30 stream được mở đồng thời sẽ cho kết quả rất tồi tệ hơn `changeNotifier`.
- Cập nhật các widget mà tốn kém ít bộ nhớ ram cho việc này. Các `GetBuilder` có thể chia sẻ các state thông qua các `GetBuilder` có id chung. Hầu hết các widget của ứng dụng là `stateless`
- Tự quản lý bộ điều khiển, controller, trong bộ nhớ.

Sử dụng Simple State Manager

- Cài đặt controller
 - Không cần sử dụng các biến reactive
 - Phương thức `get({String? tag})`: Được sử dụng để truy xuất Controller trên view
 - Gọi phương thức `update` để cập nhật View. Các đối số tùy chọn gồm:
 - Danh sách các id của các `GetBuilder` sẽ được cập nhật
 - Điều kiện để các `GetBuilder` cập nhật

```
class SimpleCounter extends GetxController{
  int counter = 0;
  static SimpleCounter get({String? tag}) => Get.find(tag: tag);
  void increment(){
    counter++;
    update(["01"], counter<=10);
  }
}
```

Sử dụng Simple State Manager

- Sử dụng `GetBuilder` để hiển thị state. Một số tham số của state:
 - `init`: Khởi tạo Controller, là một singleton, mỗi controller có một giá trị tag tùy chọn. Nếu hai Controller ở hai `Getbuilder` khác nhau nhưng có cùng giá trị tag thì chúng là một.
 - `id` (tùy chọn): Định danh của `GetBuilder`, dùng để xác định `GetBuilder` có được cập nhật hay không khi state của controller thay đổi.
 - `tag` (tùy chọn): Dùng để định danh Controller

```
GetBuilder<SimpleCounter>(
  init: SimpleCounter(),
  id: "01",
  tag: "my_simple_state",
  builder: (controller) => Text("${controller.counter}"),
),
```


Sử dụng Simple State Manager

- Gọi các phương thức của Controller trong các sự kiện:
 - Ví dụ:


```
floatingActionButton: FloatingActionButton(
    child: Icon(Icons.add),
    onPressed: (){
      SimpleCounter.get(tag: "my_simple_state").increment();
    },
  ),
```
 - Có thể thay SimpleCounter.get(tag: "my_simple_state").increment(); bằng:
 - Get.find<SimpleCounter>(tag: "my_simple_state").increment();

Huỳnh Tuấn Anh - Khoa CNTT, Đại Học Nha Trang 53

Route Management



Huỳnh Tuấn Anh - Khoa CNTT, Đại Học Nha Trang 54

Route Management

- Thay MaterialApp bằng GetMaterialApp:


```
GetMaterialApp( // Before: MaterialApp(
  home: MyHome(),
)
```
- Navigate tới NextScreen: `Get.to(NextScreen());`
- Navigate tới NextScreen nhưng không có tùy chọn quay trở lại màn hình trước:


```
Get.off(NextScreen());
```
- Thay thế Navigate.pop(context): `Get.back();`

Huỳnh Tuấn Anh - Khoa CNTT, Đại Học Nha Trang 55

Route Management

- Điều hướng đến route tiếp theo và nhận hoặc cập nhật dữ liệu ngay sau khi bạn trở về từ route đó:


```
var data = await Get.to(Payment());
```
- Trên màn hình khác, gửi dữ liệu cho route trước đó:


```
Get.back(result: 'success');
```
- Khi một route bị xóa khỏi stack, các dependency của nó thường sẽ bị xóa theo.

Huỳnh Tuấn Anh - Khoa CNTT, Đại Học Nha Trang 56

Dependency Management



Instanting Method – Phương thức tạo thể hiện của lớp

- **Get.put():** Tạo một singleton là một Controller trên View

```
S.put<S>(  
    S dependency,  
    {String? tag,  
    bool permanent = false,  
    InstanceBuilderCallback<S>? builder}  
)
```

- Ví dụ:

```
Get.put<SomeClass>(SomeClass());  
Get.put<LoginController>(LoginController(), permanent: true);  
Get.put<ListItemController>(ListItemController, tag: "some unique string");
```

Instanting Method

- **Get.lazyPut:** Để tạo các dependency mà sẽ chỉ được khởi tạo khi nó được sử dụng lần đầu tiên. Rất hữu ích cho các lớp tính toán tốn nhiều chi phí hoặc nếu bạn muốn khởi tạo một số lớp chỉ ở một nơi (như trong lớp Bindings) và bạn biết rằng mình sẽ không sử dụng lớp đó tại thời điểm đó.

```
Get.lazyPut<Controller>(() => Controller(), tag: "abc");  
.....  
var c = Get.find<Controller>(tag: "abc");
```

Instanting Method - Get.lazyPut

```
Get.lazyPut<ApiMock>(() => ApiMock());  
  
Get.lazyPut<FirebaseAuth>(  
    () {  
        // ... some logic if needed  
        return FirebaseAuth();  
    },  
    tag: Math.random().toString(),  
    fenix: true  
)  
  
Get.lazyPut<Controller>(() => Controller())
```

Instantiating Method

- **Get.putAsync:** register an asynchronous instance
- Ví dụ:

```
Get.putAsync<SharedPreferences>(() async {
    final prefs = await SharedPreferences.getInstance();
    await prefs.setInt('counter', 12345);
    return prefs;
});

Get.putAsync<YourAsyncClass>(() async => await YourAsyncClass() )
```

Instantiating Method

- **Get.create:**

```
void create<S>(
    InstanceBuilderCallback<S> builder,
    {String? tag,
    bool permanent = true}
)
```
- Tạo một thể hiện mới S bằng cách gọi callback builder mỗi khi phương thức `Get.find<S>()` được gọi. Nó đảm bảo rằng vòng đời của mỗi thể hiện S chỉ nằm trong một route nếu tham số `permanent` không được thiết lập `true`.

```
Get.Create<SomeClass>(() => SomeClass());
Get.Create<LoginController>(() => LoginController());
```

Instantiated Method/Class

- Các phương thức dùng để truy cập các controller đã được tạo:

```
final controller = Get.find<Controller>();
// OR
Controller controller = Get.find();
```

- Giá trị trả về có thể là đối tượng thuộc một lớp bình thường đã được tạo bởi các phương thức `put`

```
int count = Get.find<SharedPreferences>().getInt('counter');
print(count); // out: 12345
```

- Xóa một Controller

```
Get.delete<Controller>(); //usually you don't need to do this because GetX already
```

Bindings

- Khi một route bị xóa khỏi Ngăn xếp, tất cả các controller, variables và các thể hiện của các đối tượng liên quan đến nó sẽ bị xóa khỏi bộ nhớ. Nếu bạn đang sử dụng Stream hoặc bộ Timer, chúng sẽ tự động bị đóng và bạn không phải lo lắng về bất kỳ điều gì.
- Binding class là một lớp tách riêng dependency injection, trong khi nó "binding" các route tới các "state manager" và "dependency manager". Điều này cho phép Get biết được màn hình nào đang được hiển thị khi một controller cụ thể được sử dụng và cách loại bỏ nó.
 - Lớp Binding là lớp có nhiệm vụ tạo ra các Controller.
- Ngoài ra lớp Binding sẽ cho phép bạn có "SmartManager configuration control". Bạn có thể cấu hình các dependency được sắp xếp khi loại bỏ một route khỏi ngăn xếp, hoặc khi widget đã sử dụng nó được layout hoặc không. Bạn sẽ có quản lý phụ thuộc thông minh làm việc cho bạn, nhưng ngay cả như vậy, bạn có thể định cấu hình nó theo ý muốn.

Binding class

- Một lớp Binding là một lớp thực thi giao diện Bindings. Ví dụ:

```
class HomeBinding implements Bindings {
    @override
    void dependencies() {
        Get.lazyPut<HomeController>(() => HomeController());
        Get.put<Service>(()=> Api());
    }
}

class DetailsBinding implements Bindings {
    @override
    void dependencies() {
        Get.lazyPut<DetailsController>(() => DetailsController());
    }
}
```

Binding

- Thông báo cho route của mình, rằng bạn sẽ sử dụng binding đó để tạo kết nối giữa route manager, dependencies and states.

```
Get.to(Home(), binding: HomeBinding());
Get.to(DetailsView(), binding: DetailsBinding())
```

- Sau đó, bạn không phải lo lắng về việc quản lý bộ nhớ của ứng dụng của mình nữa, Get sẽ thay bạn làm điều đó.
- Lớp Binding được gọi khi một route được gọi, bạn có thể tạo một "InitialBinding" trong GetMaterialApp của mình để đưa vào tất cả các dependency muốn khởi tạo cho ứng dụng.

```
GetMaterialApp(
    initialBinding: SampleBind(),
    home: Home(),
);
```

SmartManagement

- Lớp dùng để thiết lập việc quản lý các dependency

```
void main () {
    runApp(
        GetMaterialApp(
            smartManagement: SmartManagement.onlyBuilders //here
            home: Home(),
        )
    )
}
```

SmartManagement

- SmartManagement.full:** được thiết lập Mặc định. Loại bỏ các dependency không được sử dụng và không được thiết lập permanent (true). Được sử dụng trong phần lớn các trường hợp.
- SmartManagement.onlyBuilders:**
 - Chỉ những controller được bắt đầu trong init: hoặc được tải vào Binding bằng Get.lazyPut() mới được loại bỏ.
 - Nếu bạn sử dụng Get.put() hoặc Get.putAsync() hoặc bất kỳ cách tiếp cận nào khác, SmartManagement sẽ không có quyền loại trừ sự phụ thuộc này
- SmartManagement.keepFactory:** Cũng giống như SmartManagement.full, nó sẽ loại bỏ các dependency của nó khi nó không được sử dụng nữa. Tuy nhiên, nó sẽ giữ nguyên factory của chúng, có nghĩa là nó sẽ tạo lại các dependency đó nếu bạn cần lại phiên bản đó.



Chú ý

- **KHÔNG SỬ DỤNG** `SmartManagement.keepFactory` nếu bạn đang sử dụng nhiều Binding. Nó được thiết kế để sử dụng mà không có Bindings, hoặc với một Binding duy nhất được liên kết trong `initialBinding` của `GetMaterialApp`.
- Sử dụng Bindings là hoàn toàn tùy chọn, nếu muốn, bạn có thể sử dụng `Get.put()` và `Get.find()` trên các lớp sử dụng một controller đã cho mà không gặp bất kỳ vấn đề gì. Tuy nhiên, nếu bạn làm việc với các Service hoặc bất kỳ abstraction nào khác, nên sử dụng Binding để tổ chức tốt hơn.

GetX với dữ liệu không đồng bộ



GetX với dữ liệu Stream

- Nguyên tắc:
 1. Viết Controller quản lý trạng thái ứng dụng có kiểu dữ liệu `Rx{Type}`.
 2. Gắn kết (Bind) trạng thái ứng dụng với Stream thông qua phương thức `bindStream` trong phương thức `onInit` hay `onReady` tùy theo yêu cầu khởi tạo Stream.
 3. Viết lớp Binding để tách biệt các Controller với các View.
 4. Sử dụng các widget `Obx` hay `GetX` để hiển thị dữ liệu.



Ví dụ

- Truy vấn dữ liệu từ Firebase và hiển thị danh dữ liệu, các sinh viên, lên màn hình. Nếu dữ liệu trên Firebase thay đổi, dữ liệu hiển thị trên màn hình cũng tự động cập nhật

Truy vấn dữ liệu từ Firebase

Viết phương thức truy vấn dữ liệu từ Firebase:

```
static Stream<List<SinhVienSnapshot>> dsSVTuFirebase(){
    Stream<QuerySnapshot> qs = FirebaseFirestore.instance.
        collection("SinhVien").snapshots();
    if(qs == null)
        return Stream.empty();
    Stream<List<DocumentSnapshot>> listDocSnap =
        qs.map((querySn) => querySn.docs);
    return listDocSnap.map((listDocSnap) =>
        listDocSnap.map((docSnap) => SinhVienSnapshot.fromSnapshot(docSnap))
        .toList());
}
```

Viết Controller quản lý trạng thái

- Viết Controller, khai báo một danh sách Rx{Type}, thực hiện gắn kết (bind) danh sách Rx với một Stream trong phương thức Init

```
class Controller_SinhVienFirebase extends GetxController{
    final list = RxList<SinhVienSnapshot>([]);
    @override
    void onInit() {
        list.bindStream(SinhVienSnapshot.dsSVTuFirebase());
        super.onInit();
    }
    void getSinhviens() {...}}
}
```

Viết lớp Binding để quản lý các Controller

- Sử dụng lazyPut để tạo Controller và truy vấn dữ liệu khi Controller được gọi lần đầu tiên

```
class Binding_SinhVienFirebase implements Bindings{
    @override
    void dependencies() {
        Get.lazyPut(() => Controller_SinhVienFirebase(), tag: "dssv");
    }
}
```

Hiển thị dữ liệu trên màn hình ứng dụng

```
class Page_GetXFirebaseApp extends StatelessWidget {
    const Page_GetXFirebaseApp({Key? key}) : super(key: key);
    @override
    Widget build(BuildContext context) {
        return MyFirebaseConnect(
            builder:(context) {
                return GetMaterialApp(
                    initialBinding: Binding_SinhVienFirebase() ,
                    title: "GetX Firebase App",
                    home: const PageGetXSinhViens() ,
                ); // GetMaterialApp
            },
            errorMessage: "Lỗi khi kết nối Firebase",
            connectingMessage: "Đang kết nối với Firebase"); //
    }
}
```

Khởi tạo Binding ở mức ứng dụng do trong ứng dụng màn hình hiển thị DSSV được khai báo cho thuộc tính home. Có thể khởi tạo Binding khi gọi các Phương thức Get.to để điều hướng màn hình.

Hiển thị dữ liệu lên màn hình

```
Widget build(BuildContext context) {
  final c = Get.find<Controller_SinhVienFirebase>(tag: "dssv");
  return Scaffold(
    appBar: AppBar(...), // AppBar
    body: Obx((){
      return ListView.separated(
        itemBuilder: (context, index) => ListTile(
          leading: Text(c.list?.value[index].sinhVien?.id?? "No ID"),
          title: Text(c.list?.value[index].sinhVien?.ten?? "No Name"),
          subtitle: Text(c.list?.value[index].sinhVien?.que_quan?? ""),
          separatorBuilder: (context, index) => const Divider(thickness: 2),
          itemCount: c.list?.value.length?? 1,
        );
      ); // ListView.separated, Obx, Scaffold
    })
  );
}
```

Làm việc với dữ liệu Future

- Nguyên tắc:
 - Viết Controller quản lý trạng thái ứng dụng.
 - Viết các phương thức truy xuất dữ liệu bất đồng bộ
 - Sử dụng phương thức **then** để gán dữ liệu bất đồng bộ vào các trạng thái ứng dụng
 - Sử dụng refresh hay update nếu cần để cập nhật trạng thái lên màn hình trong trường hợp trạng thái là các Custom Type
 - Sử dụng Obx/GetX/GetBuilder để hiển thị trạng thái của ứng dụng

Ví dụ:


- Truy vấn dữ liệu từ Firebase bằng phương thức get

```
static Future<List<SinhVienSnapshot>> dsSVTuFirebaseOneTime() async{
  QuerySnapshot qs = await FirebaseFirestore.instance.
    collection("SinhVien").get();
  return qs.docs.map((docSnap) => SinhVienSnapshot.
    fromSnapshot(docSnap)).toList();
}
```

Ví dụ

- Viết Controller để quản lý các trạng thái

```
class Controller_SinhVienFirebase extends GetxController{
  RxList<SinhVienSnapshot>? list = RxList<SinhVienSnapshot>([]);
  @override
  void onInit() {...}
  void getSinhviens() {
    SinhVienSnapshot.dsSVTuFirebaseOneTime().then(
      (value) {
        list?.value = value;
        list?.refresh();
      });
  }
}
```

 Tài liệu tham khảo

- https://github.com/jonataslaw/getx/blob/master/documentation/en_US/state_management.md

Huỳnh Tuấn Anh - Khoa CNTT, Đại Học Nha Trang

81