GetX

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

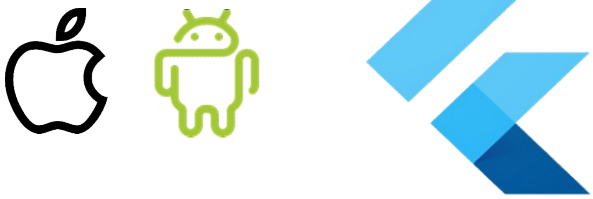
Giới thiệu về GetX

- 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.

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
  - ...

State Management



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
- So sánh về khả năng tiết kiệm bộ nhớ khi sử dụng:
  - GetBuilder > Obx > GetX

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 isLogged = 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 isLogged = 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 isLogged = 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 TextEditController, 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 sẽ 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}"),
),
```

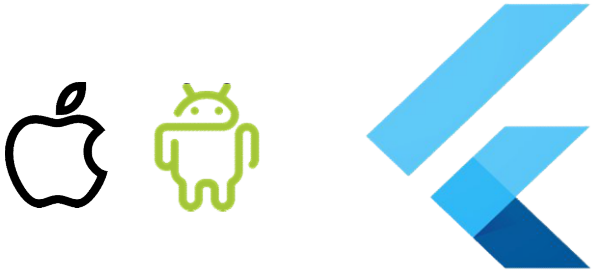

Sử dụng Simple State Manager

- 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();`

Route Management



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();`

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.

Dependency Management

Instancing 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");
```


Instancing 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");
```

Instancing 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() )
```


Instancing 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() )
```

Instancing 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(),
    )
  )
}
```

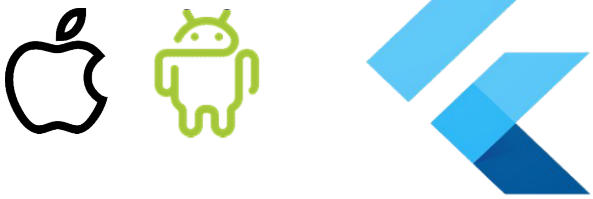
SmartManagement

- **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 dung 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 Phuong 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 nhập trạng thái lên màn hình trong trường hợp trạng thái là các Custome 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();
        });
  }}
```