

Hiểu về cách thêm thư viện bên ngoài vào dự án Flutter

Cách mà Dart tổ chức và chia sẻ các chức năng thông package . Dart Package là một thư viện hay mô hình đơn giản có thể chia sẻ. Nhìn chung, Dart package cũng giống như Dart Application ngoại trừ Dart Package không truy cập vào các điểm chính của ứng dụng

Cấu trúc chung của Package (ví dụ về package) dưới đây :

- **lib/src/*** : tệp Dart ở dạng private
- **lib/my_demo_package.dart** : phần code chính của Dart, có thể thêm một vài ứng dụng

```
1 | import 'package:my_demo_package/my_demo_package.dart'
```

- Một vài tệp ở dạng private có thể được xuất sang tệp chính (my_demo_package.dart) :

```
1 | export src/my_private_code.dart
```

- **lib/*** : Ta có thể truy cập vào bất kì tệp nào bên trong thư mục :

```
1 | import 'package:my_demo_package/custom_folder/custom_file.dart'
```

- **pubspec.yaml** : Được hiểu là trình quản lý thư mục của Package

Để tích hợp được các gói vào dự án thì ta cần phải có file pubspec.yaml

Các kiểu Package :

Kể từ khi Dart package là một collection có chức năng tương tự , nó có thể được phân loại dựa trên chức năng:

Dart Package

Chúng ta có thể sử dụng Dart trên cả 2 môi trường là web và android. Ví dụ , english_words là một package chứa khoảng 500 từ và có chức năng tiện ích cơ bản như danh từ (list các danh từ trong English), âm tiết (liệt kê ra các từ có âm tiết đặc biệt)

Flutter package

Phụ thuộc vào Flutter framework và có thể chỉ sử dụng trong môi trường mobile .

Flutter plugin

Phụ thuộc vào Flutter framework cũng như nền tảng cơ bản (Android SDK hay iOS SDK). Ví dụ Camera là một plugin (có thể hiểu là một phần mềm hỗ trợ) để tương tác với thiết bị camera. Nó sử dụng SDK để có quyền truy cập vào camera

Sử dụng Dart Package :

Dart package được lưu trữ và publish trên các máy chủ, <https://pub.dev> . Ngoài ra, Flutter cung cấp các tool, pub cơ bản để quản lý các Dart package trong ứng dụng. Các bước cần để sử dụng Package như sau :

-Nhập tên package và phiên bản phù hợp trong file pubspec.yaml như dưới đây :

```
1 | dependencies: english_words: ^3.1.5
```

-Bản mới nhất sẽ được cập nhật trên server

- Cài đặt package bằng lệnh :

```
1 | flutter packages get
```

- Khi chúng ta đang dùng Android studio, thì Android studio sẽ phát hiện bất kì thay đổi trong file pubspec.yaml và hiện thông báo để lập trình viên có thể biết

- Dart package có thể được cài đặt hoặc nâng cấp trong Android studio thông qua menu options/options .

- Thêm các file cần thiết sử dụng lệnh dưới đây và bắt đầu làm việc :

```
1 | import 'package:english_words/english_words.dart';
```

- Sử dụng bất kì phương thức có sẵn

```
1 | nouns.take(50).forEach(print);
```

- Ở trên ta đã dùng hàm nouns để lấy ra 50 từ đầu tiên

Quản lý version của thư viện trong Flutter

Ở bài này chúng ta sẽ cùng nhau đi học cách quản lý version của các package được sử dụng trong dự án của mình.

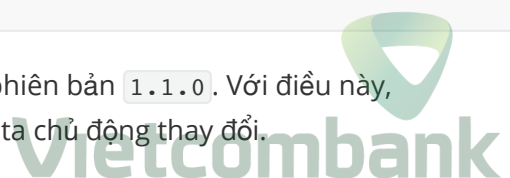
Có khá nhiều cách để `dependency` một package.

Cách 1: `dependency` cứng

```
1 | collection: "1.1.0"
```

Ở cách này thì chúng ta sẽ `dependency` package `collection` ở phiên bản `1.1.0` . Với điều này, `collection` sẽ được pull cố định ở phiên bản `1.1.0` , cho tới khi ta chủ động thay đổi.

Cách 2: `Semantic version`



Khi bạn `dependency` một package nào đó và bạn chỉ định Flutter có thể lấy bất kể phiên bản nào từ phiên bản bạn truyền vào. Trong trường hợp `author` của package bạn đang sử dụng upgrade package của họ vì một số lý do nào đó chẳng hạn như hotfix. Thì Flutter sẽ tự động lấy phiên bản phù hợp đó khi bạn gọi lệnh `Flutter pub get`

```
1 | collection: ^2.3.5
```

Và dãy version Flutter cho phép tải về là từ 2.3.5 đến dưới 3.0.0

Lưu ý: Ví dụ này sử dụng *cú pháp dấu mũ* để thể hiện một loạt các phiên bản. Chuỗi `^2.3.5` có nghĩa là "phạm vi của tất cả các phiên bản từ 2.3.5 đến 3.0.0, không bao gồm 3.0.0." Để biết thêm thông tin, hãy xem [Cú pháp con dấu](#).

Cách 3: Version constraints

```
1 | dependencies:  
2 |   collection: '>=2.3.5 <2.4.0'
```

Ở cách này, bạn cho Flutter biết là phạm vi mà Flutter có thể pull package `collection` về là từ 2.3.5 cho đến nhỏ hơn 2.4.0. Trong trường hợp `collection` có một phiên bản hotfix là `2.3.6` thì hiển nhiên nó sẽ nằm trong dự án của bạn.

Cách 4: Vạn sự tùy duyên

Ngoài các cách ở trên. Bạn cũng có thể sử dụng cách này

```
1 | dependencies:  
2 |   love:  
3 |     // or  
4 |     love: any
```

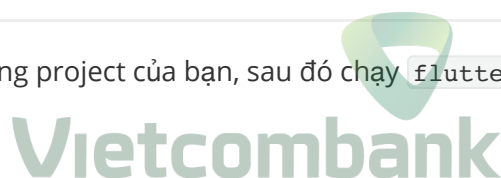
Khi mà bạn dùng cách này, mình biết bạn đã cố gắng rất nhiều. Bạn chán đời, bạn thất vọng, bạn mệt mỏi khi mãi vẫn không tìm ra được một version phù hợp. Nghĩ mà xem, điều này giống như cuộc đời bạn vậy. Tìm mãi, tìm mãi vẫn không thể tìm được một nửa bạn cần. Đến cuối cùng, bạn thở dài và nghĩ trong đầu là: "Thôi thì vạn sự tùy duyên", giống như cách bạn phó mặc cho Flutter tự tìm cho bạn một version phù hợp vậy.

Nhưng đây cũng là cách mình không khuyến cáo các bạn sử dụng vì không tương minh.

Hiểu về Dependencu Injection trong Flutter

Cài đặt

Như các library khác, bạn cần thêm nó vào file `pubspec.yaml` trong project của bạn, sau đó chạy `flutter packages get` để cài đặt.



```
1 dependencies:
2   ...
3   get_it: ^4.0.4
```

Sau đó trong project chúng ta sẽ tạo ra một file mới, mình đặt tên là `injection.dart`. Trong file này hãy tạo một hàm để lát sau chúng ta sẽ đăng kí các dependency trong đó. Nội dung file tương tự như sau:

```
1 import 'package:get_it/get_it.dart';
2
3 final getIt = GetIt.instance;
4
5 void configureDependencies() async {
6   // TODO: đăng kí các dependency trong này
7 }
```

Và cuối cùng mở file `main.dart`, gọi hàm mà chúng ta vừa tạo trước khi render UI:

```
1 import 'injection.dart';
2
3 void main() async {
4   await configureDependencies();
5
6   runApp(MyApp());
7 }
```

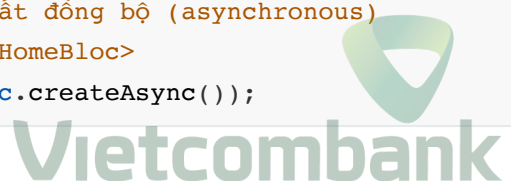
Tạo các instances

`get_it` cung cấp cho chúng ta gần như đầy đủ các pattern để tạo ra instance:

Factory

Factory được hiểu như một nhà máy sản xuất object. Mỗi khi bạn gọi đến để lấy object thì sẽ có một instance mới được tạo ra và trả về cho bạn. Cú pháp như sau:

```
1 // Dùng khi instance có thể khởi tạo được ngay
2 getIt.registerFactory<HomeBloc>(() => HomeBloc());
3
4 // Dùng khi instance bắt buộc phải tạo dưới dạng bất đồng bộ (asynchronous)
5 // hàm createAsync của HomeBloc trả về một Future<HomeBloc>
6 getIt.registerFactoryAsync<HomeBloc>(() => HomeBloc.createAsync());
```



Chúng ta dùng *Factory* khi luôn muốn nhận về một instance mới mỗi khi sử dụng mà không liên quan gì đến instance trước để tránh trường hợp dùng lại các data cũ đã init từ instance trước hoặc pointer cũ (điều này thể hiện rất rõ nếu như bạn đã từng sử dụng redux, phải luôn trả về một object mới để có thể render lại được).

Factory không nên dùng nếu như object của bạn có chứa các logic code quá phức tạp dẫn đến việc làm chậm quá trình khởi tạo và lãng phí tài nguyên do luôn phải tạo lại mới mỗi khi cần đến.

Singleton

Singleton trái ngược với *factory*, chỉ tạo ra một instance duy nhất kể từ khi app khởi động, sau đó nếu bất kì chỗ nào có dùng đến thì sẽ chỉ trả về instance đã tạo trước đó. Do đó xuyên suốt app, bạn sẽ chỉ sử dụng một instance của object đó mà thôi.

```
1 // Dùng khi instance có thể khởi tạo được ngay
2 getIt.registerSingleton<CounterRepository>(CounterRepository());
3
4 // Dùng khi instance bắt buộc phải tạo dưới dạng bất đồng bộ (asynchronous)
5 // hàm createAsync của CounterRepository trả về một Future<CounterRepository>
6 getIt.registerSingletonAsync<CounterRepository>(() =>
  CounterRepository.createAsync());
```

Trái ngược với *factory*, nên dùng *singleton* khi bạn chỉ muốn khởi tạo object một lần và dùng ở nhiều chỗ, tránh lãng phí tài nguyên. Không nên dùng nếu như nó phụ thuộc quá nhiều về mặt giá trị và pointer, dễ gây lỗi app về mặt logic nếu không xử lý cẩn thận.

Lazy-singleton

Lazy-singleton thì giống như *singleton*, chỉ khác là nó sẽ được khởi tạo vào **lần gọi lấy instance đầu tiên**, chứ không phải khi app khởi động. Sử dụng nó nếu như việc tạo instance này mất thời gian, bạn không muốn app dừng ở màn hình splash quá lâu để chờ khởi tạo instance, dẫn đến việc UX của app không tốt.

Ngoài ra nếu bạn nghĩ object này ở một trường hợp nào đó có thể sẽ không sử dụng thì cũng có thể dùng cách này để tránh lãng phí tài nguyên.

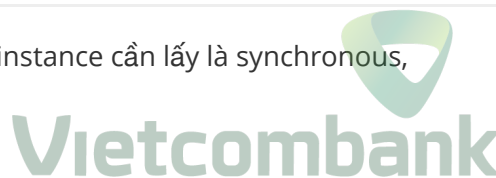
Ví dụ như khi mất mạng, user sẽ không cần gọi api, dẫn đến không cần instance network. Chỉ khi user có mạng, api được gọi lần đầu tiên thì instance được khởi tạo và sử dụng bình thường

```
1 getIt.registerLazySingleton<CounterRepository>(() => CounterRepository());
```

Sử dụng

Rất đơn giản, bạn chỉ cần dùng `getIt.get<T>()` với trường hợp instance cần lấy là synchronous, `getIt.getAsync<T>()` với trường hợp là asynchronous

Giải quyết dependency



Có những trường hợp object A cần cung cấp object B mới có thể hoạt động (A depends on B), vậy thì chúng ta sẽ phải làm như nào?

Factory/singleton A phụ thuộc vào factory/singleton B

```
1 class A {
2     final B b;
3
4     const A(this.b);
5 }
6
7 class B {
8
9 }
```

```
1 // đăng kí B trước tiên
2 getIt.registerSingleton<B>(B());
3
4 getIt.registerFactory<A>(() {
5     // lấy object B bên trên...
6     final b = getIt.get<B>();
7
8     // ...truyền vào constructor của A
9     return A(b);
10 });
```

Factory/singleton A phụ thuộc vào asynchronous factory/singleton B

Lúc này phải chuyển hàm khởi tạo A thành asynchronous chứ không còn dùng được synchronous nữa, cụ thể là phải dùng:

- `registerFactoryAsync` thay vì `registerFactory`
- `registerSingletonAsync` thay vì `registerSingleton`

```

1 class A {
2     final B b;
3
4     const A(this.b);
5 }
6
7 class B {
8     static Future<B> createAsync() {
9         // ... khởi tạo B
10    }
11 }

```

```

1 // đăng kí B trước tiên
2 getIt.registerSingletonAsync<B>(() => B.createAsync());
3
4 getIt.registerFactoryAsync<A>(() async {
5     // chờ và lấy object B bên trên...
6     final b = await getIt.getAsync<B>();
7
8     // ...truyền vào constructor của A
9     return A(b);
10 });

```

Asynchronous factory/singleton A phụ thuộc vào factory/singleton B

Phần này cũng khá giống với `Factory/singleton A phụ thuộc vào factory/singleton B`

```

1 class A {
2     final B b;
3
4     const A(this.b);
5
6     static Future<A> createAsync(B b) {
7         // ... khởi tạo A
8     }
9 }
10
11 class B {
12
13 }

```

```

1 // đăng kí B trước tiên
2 getIt.registerSingleton<B>(B())
3
4 getIt.registerFactoryAsync<A>(() async {
5     // lấy object B bên trên...
6     final b = getIt.get<B>();
7
8     // ...truyền vào hàm khởi tạo của A
9     return await A.createAsync(b);
10 });

```

Asynchronous factory/singleton A phụ thuộc vào asynchronous factory/singleton B

Phần này cũng khá giống với `Factory/singleton A phụ thuộc vào asynchronous factory/singleton B`

```

1 class A {
2     final B b;
3
4     const A(this.b);
5
6     static Future<A> createAsync(B b) {
7         // ... khởi tạo A
8     }
9 }
10
11 class B {
12     static Future<B> createAsync() {
13         // ... khởi tạo B
14     }
15 }

```

```

1 // đăng kí B trước tiên
2 getIt.registerSingletonAsync<B>(() => B.createAsync())
3
4 getIt.registerFactoryAsync<A>(() async {
5     // lấy object B bên trên...
6     final b = await getIt.getAsync<B>();
7
8     // ...truyền vào hàm khởi tạo của A
9     return await A.createAsync(b);
10 });

```


Truyền params vào factory

Với một số trường hợp, bạn muốn truyền tham số vào constructor khi khởi tạo object, ví dụ như `User(age: 12, name: 'Kevin')`, `get_it` cũng cho phép bạn truyền param với số lượng tối đa là 2 params.

```
1 class User {
2     final int age;
3     final String name;
4
5     const User({this.age, this.name});
6 }
```

```
1 // Khởi tạo
2 getIt.registerFactoryParam<User, int, String>((age, name) => User(age: age, name:
    name));
```

```
1 // Sử dụng
2 getIt.get<User>(param1: 5, param2: 'Kevin');
```

Nếu bạn muốn truyền nhiều hơn 2 params, có thể tạo một class đại diện cho các params và truyền vào như một param bình thường:

```
1 class UserParams {
2     final int age;
3     final String name;
4     final String address;
5
6     const UserParams({this.age, this.name, this.address});
7 }
8
9 class User {
10     final int age;
11     final String name;
12     final String address;
13
14     const User({this.age, this.name, this.address});
15
16     User.withParams(UserParams params) : this(age: params.age, name: params.name,
        address: params.address);
17 }
```

```
1 // Khởi tạo
2 getIt.registerFactoryParam<User, UserParams>((params) => User.withParams(params));
```

```
1 // Sử dụng
2 getIt.get<User>(param1: UserParams(age: 5, name: 'Kevin', address: 'Hanoi'));
```

Tự động đăng kí dependency với injectable

Mình đã từng code Java Spring và thấy cơ chế inject dependency của nó khá hay, chỉ cần thêm annotation trên đầu class cần inject và nó sẽ tự động tìm và inject luôn cho mình chứ không phải declare ra như bên trên.

Thật may là với `build_runner` và `injectable`, chúng ta có thể hoàn toàn tự động được công việc nhàm chán này.

Cài đặt

Đầu tiên chúng ta cần thêm vào `pubspec.yaml`, chạy `flutter packages get` để cài đặt.

```
1 dependencies:
2   ...
3   injectable: ^1.0.4
4
5 dev_dependencies:
6   ...
7   injectable_generator: ^1.0.4
8   build_runner: ^1.10.2
```

Bây giờ hãy mở file `injection.dart`, sửa lại thành như sau:

```
1 import 'package:get_it/get_it.dart';
2 import 'package:injectable/injectable.dart';
3
4 import 'injection.config.dart';
5
6 final getIt = GetIt.instance;
7
8 @InjectableInit()
9 Future<void> configureDependencies() async => await $initGetIt(getIt);
```



Vietcombank

Bạn sẽ thấy báo lỗi ở `import 'injection.config.dart';` và `$initGetIt`. Đừng lo, hãy mở terminal lên, cd đến project và chạy lệnh sau:

```
1 flutter packages pub run build_runner build
```

Sau khi terminal chạy hoàn tất, bạn sẽ thấy có một file mới tên `injection.g.dart` tạo bởi `injectable`, nằm cùng vị trí với `injection.dart` và lỗi bên trên cũng đã hết. Vậy là chúng ta đã setup xong.

Sử dụng

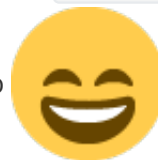
Giờ thì thay vì chúng ta viết mọi thứ ở trong `configureDependencies()`, hãy tạm quên nó đi và chuyển qua object bạn cần khởi tạo.

Giả sử mình có 2 class A và B, mình muốn thêm nó vào DI dưới dạng factory, A phụ thuộc vào B thì mình chỉ cần import và thêm annotation `@injectable` phía trên 2 class đó:

```
1 import 'package:injectable/injectable.dart';
2
3 @injectable
4 class A {
5     final B b;
6
7     const A(this.b);
8 }
9
10 @injectable
11 class B {
12
13 }
```

Bạn chạy lại lệnh `flutter packages pub run build_runner build` và mở file `injection.config.dart`

lên xem, nếu nó trông tương tự như này tức là chúng ta đã thành công rồi đó



```
1 // GENERATED CODE - DO NOT MODIFY BY HAND
2
3 // *****
4 // InjectableConfigGenerator
5 // *****
6
7 import 'package:get_it/get_it.dart';
8 import 'package:injectable/injectable.dart';
9
```



```

10 import 'models.dart';
11
12 /// adds generated dependencies
13 /// to the provided [GetIt] instance
14
15 GetIt $initGetIt(
16   GetIt get, {
17     String environment,
18     EnvironmentFilter environmentFilter,
19   }) {
20   final gh = GetItHelper(get, environment, environmentFilter);
21   gh.factory<B>(() => B());
22   gh.factory<A>(() => A(get<B>()));
23   return get;
24 }

```

Trong quá trình code, chúng ta có thể thay lệnh `flutter packages pub run build_runner build` bằng `flutter packages pub run build_runner watch`, và chỉ việc save lại file là `injectable` sẽ tự build lại file cho bạn luôn.

Tất nhiên ngoài factory ra, chúng ta cũng có thể dùng singleton và lazy-singleton bằng các annotation `@singleton` và `@lazySingleton`.

Với asynchronous factory bạn có thể dùng `@injectable` trên class và `@factoryMethod` trên hàm khởi tạo như sau:

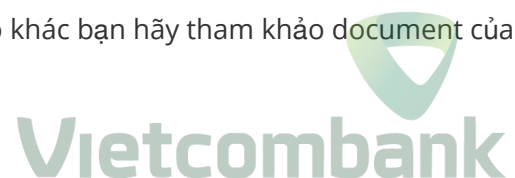
```

1 import 'package:injectable/injectable.dart';
2
3 @injectable
4 class A {
5   @factoryMethod
6   static Future<A> createAsync() {
7     ...
8   }
9 }

```

`injectable` còn giúp chúng ta tách các dependency theo các môi trường khác nhau để sử dụng, dễ dàng viết unit test, mock data,... Ví dụ như bạn có thể tạo riêng `DevRepository` với dev url, config riêng so với `StgRepository` hay `ProdRepository` để tránh việc dev nhầm môi trường, hay `TestRepository` gồm các sample data để tiện cho việc mock, unit test.

Vì bài này khá dài rồi nên tính năng này và các tính năng nâng cao khác bạn hãy tham khảo document của `injectable` để tìm hiểu thêm nhé, rất hay đó



** Tham khảo tài liệu tại viblo.asia

Làm việc với ListView & cách truyền data vào listview widget

Nếu bạn muốn có một giải pháp để hiển thị một danh sách các Widget và thậm chí có thể cuộn được theo chiều ngang hay dọc thì ListView chính là một lựa chọn vô cùng hiệu quả.

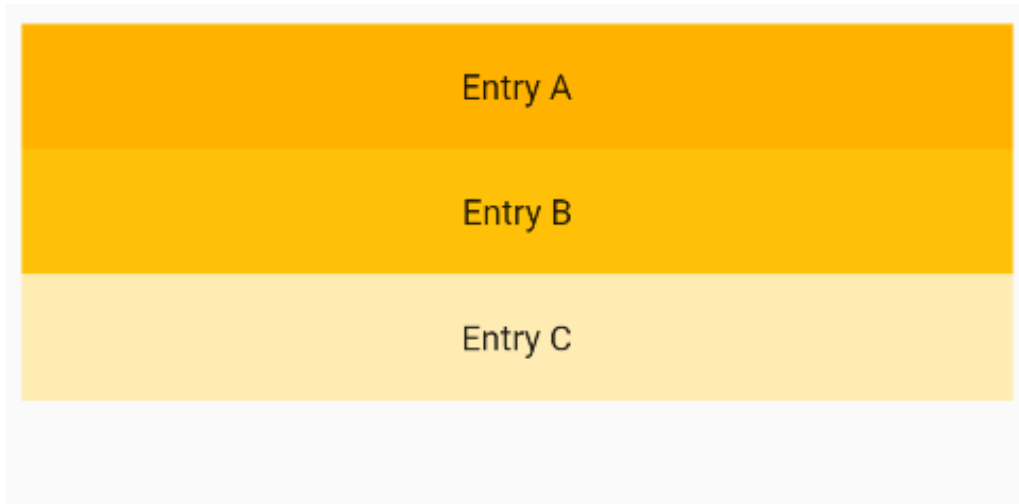
Dưới đây là một số cách để xây dựng một ListView:

Sử dụng List cho thuộc tính children

Đây là cách xây dựng mặc định của một ListView. Bằng cách xây dựng từng Widget cụ thể và đặt trong children của ListView, các Widget sẽ được hiển thị lần lượt theo trên giao diện người dùng.

Cách xây dựng này phù hợp với việc hiển thị một số lượng ít các Widget vì việc xây dựng một List yêu cầu cần phải làm việc với tất cả các thành phần con có thể được hiển thị kể cả khi các Widget chưa hiển thị lên màn hình.

```
ListView(  
  padding: const EdgeInsets.all(8),  
  children: [  
    Container(  
      height: 50,  
      color: Colors.amber[600],  
      child: const Center(child: Text('Entry A')),  
    ), // Container  
    Container(  
      height: 50,  
      color: Colors.amber[500],  
      child: const Center(child: Text('Entry B')),  
    ), // Container  
    Container(  
      height: 50,  
      color: Colors.amber[100],  
      child: const Center(child: Text('Entry C')),  
    ), // Container  
  ],  
) // ListView
```



Sử dụng ListView.builder

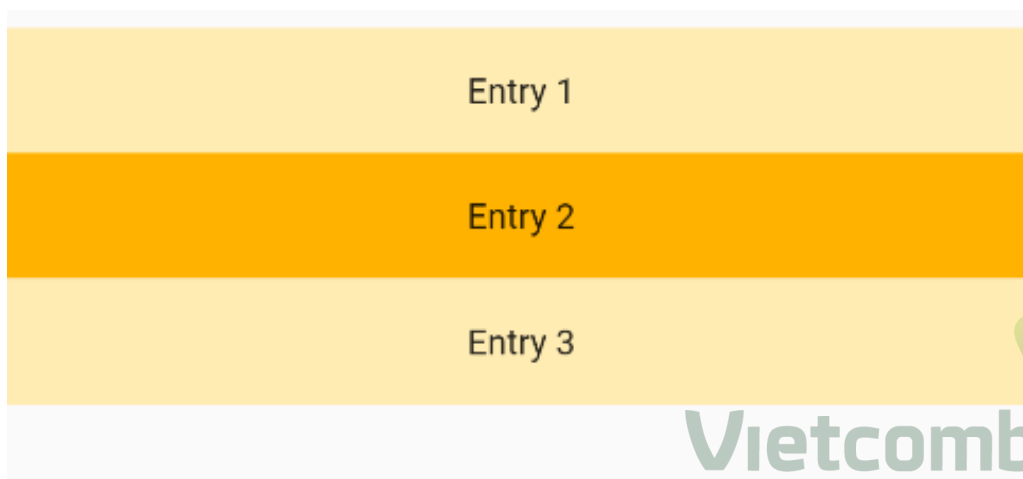
Đây là cách xây dựng ListView được áp dụng cho trường hợp cần hiển thị một lượng lớn (hay vô hạn) các Widget con vì builder chỉ được gọi cho những Widget thực sự được hiển thị lên màn hình.

```
List<int> listData = List.generate(100, (index) => index + 1);
```

Giả sử ta có một mảng dữ liệu 100 phần tử từ 1 đến 100

```
ListView.builder(  
  itemCount: 3,  
  itemBuilder: (context, index) {  
    return Container(  
      height: 50,  
      color: Colors.amber[index % 2 == 0 ? 100 : 600],  
      child: Center(child: Text('Entry ${listData[index]}')),  
    ); // Container  
  },  
) // ListView.builder
```

Ta có thể thay đổi số lượng item được hiển thị qua thuộc tính itemCount



Sử dụng ListView.separated

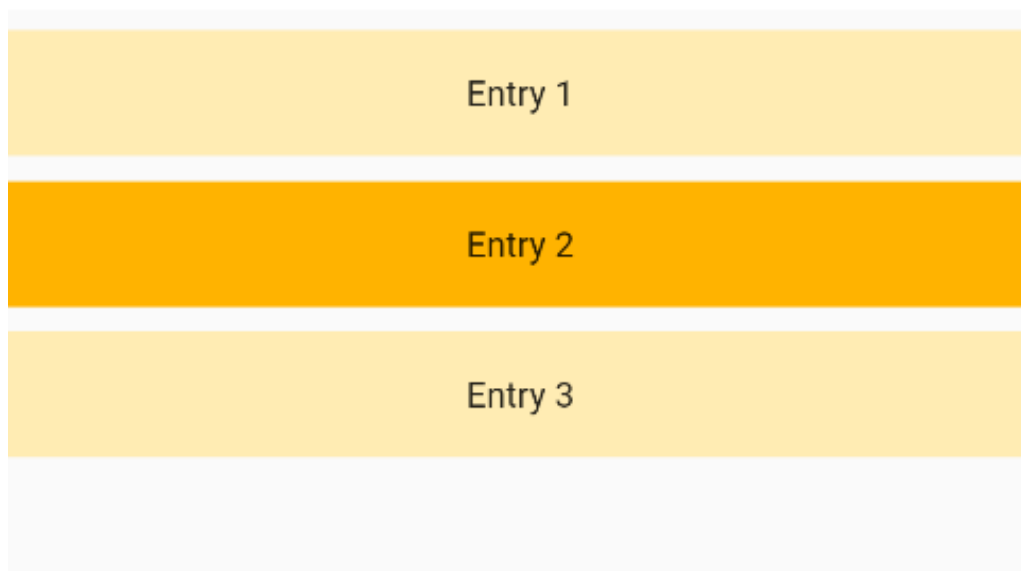
Đây là cách xây dựng ListView được áp dụng cho trường hợp khi cần hiển thị một số lượng lớn các Widget con và các Widget dùng để ngăn cách giữa các Widget đó vì builder chỉ được gọi cho những Widget thực sự được hiển thị lên màn hình.

```
List<int> listData = List.generate(100, (index) => index + 1);
```

Giả sử ta có một mảng dữ liệu 100 phần tử từ 1 đến 100

```
ListView.separated(  
  itemCount: 3,  
  itemBuilder: (context, index) {  
    return Container(  
      height: 50,  
      color: Colors.amber[index % 2 == 0 ? 100 : 600],  
      child: Center(child: Text('Entry ${listData[index]}')),  
    ); // Container  
  },  
  separatorBuilder: (context, index) => const SizedBox(height: 10)  
) // ListView.separated
```

Ta có thể thêm Widget dùng để tách các Widget con qua separatorBuilder



Sử dụng ListView.custom

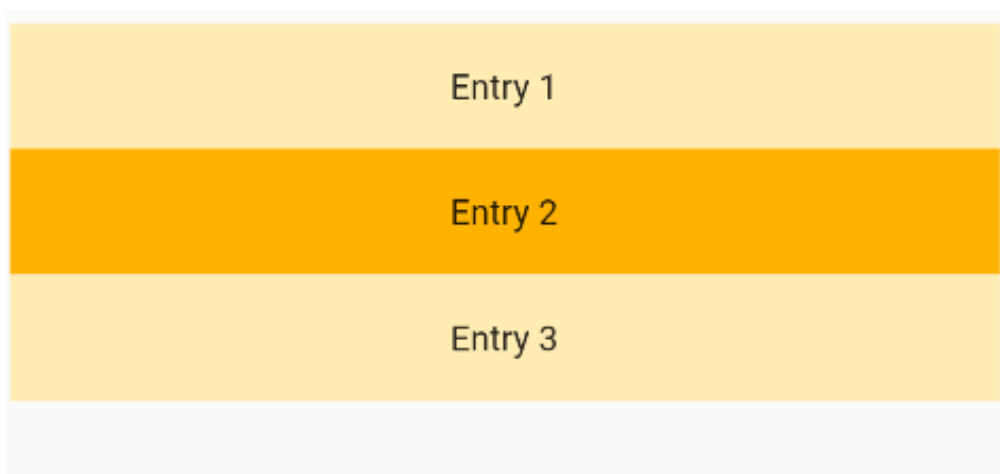
Đây là cách xây dựng ListView giúp bạn có thể tùy chỉnh nhiều hơn cho các model con. Ví dụ: một model con tùy chỉnh có thể kiểm soát thuật toán được sử dụng để ước tính kích thước của các mô hình con không thực sự hiển thị.

```
List<int> listData = List.generate(100, (index) => index + 1);
```

Giả sử ta có một mảng dữ liệu 100 phần tử từ 1 đến 100

```
ListView.custom(  
  childrenDelegate: SliverChildBuilderDelegate(  
    (context, index) => Container(  
      height: 50,  
      color: Colors.amber[index % 2 == 0 ? 100 : 600],  
      child: Center(child: Text('Entry ${listData[index]}')),  
    ), // Container  
    childCount: 3,  
  ), // SliverChildBuilderDelegate  
, // ListView.custom
```

Sử dụng SliverChildBuilderDelegate để xây dựng Widget con



Một số thuộc tính thường được sử dụng

padding

ListView có khoảng cách với Widget cha. Ví dụ: padding: const EdgeInsets.all(8).

scrollDirection

Thuộc tính scrollDirection xác định hướng cuộn của ListView, mặc định của ListView sẽ là vertical. Ví dụ: scrollDirection: Axis.horizontal.

reverse

ListView sẽ được hiển thị ngược chiều, đây là thuộc tính có kiểu bool, mặc định là false.

physics

Thuộc tính physics giúp bạn cài đặt ListView được cuộn như thế nào. Ví dụ: physics: const NeverScrollableScrollPhysics(),

Tài liệu tham khảo

ListView class trong Flutter: [ListView class - widgets library - Dart API](#)


Làm việc với ListTile Widget


Sử dụng `ListTile`, một widget giống `Row` từ thư viện `Material`, để dễ dàng tạo một hàng chứa tối đa 3 dòng văn bản và các biểu tượng đầu và cuối tùy chọn. `ListTile` được sử dụng phổ biến nhất trong `Card` hoặc `ListView`, nhưng có thể được sử dụng ở những nơi khác.


Tóm tắt (ListTile)

- Một widget giống `Row` chứa tối đa 3 dòng văn bản và các biểu tượng tùy chọn
- Ít cấu hình hơn `Row`, nhưng dễ sử dụng hơn
- Từ thư viện `Material`

Examples (ListTile)

 1625 Main Street
My City, CA 99984

 (408) 555-1212

 costa@example.com

Một `card` bao gồm 3 `ListTiles`.

A dropdown button displays a menu that's used to select a value from a small set of values. The button displays the current value and a down arrow.

Simple dropdown:

Free ▼


Dropdown with a hint:

Choose ▼

Scrollable dropdown:

Four ▼

Sử dụng `ListTile` để tạo danh sách gồm 3 button dạng dropdown.



Tích hợp thư viện firebase_core vào dự án


Tạo project Firebase:


Điều này là dĩ nhiên rồi không có nó làm sao mà liên kết được =)). Click vào **New project**.

Let's start with a name for your project[?]

Project name

FlutterApp

 flutterapp-d12a5

 You're 2 projects away from the project limit. Consider adding Firebase to an existing project or request an increased limit.


[Request an increase](#)


Continue

Let's start with a name for your project[?]

Project name

FlutterApp

 flutterapp-d12a5

 You're 2 projects away from the project limit. Consider adding Firebase to an existing project or request an increased limit.



Vietcombank

[Request an increase](#)

[Continue](#)

Nhập tên project mới và chọn **continue**.

Google Analytics for your Firebase project

Google Analytics is a free and unlimited analytics solution that enables targeting, reporting, and more in Firebase Crashlytics, Cloud Messaging, In-App Messaging, Remote Config, A/B Testing, Predictions, and Cloud Functions.

Google Analytics enables:

- ✕ A/B testing ?
- ✕ User segmentation & targeting across Firebase products ?
- ✕ Predicting user behavior ?
- ✕ Crash-free users ?
- ✕ Event-based Cloud Functions triggers ?
- ✕ Free unlimited reporting ?

☐ **Enable Google Analytics for this project**
Recommended

[Previous](#)

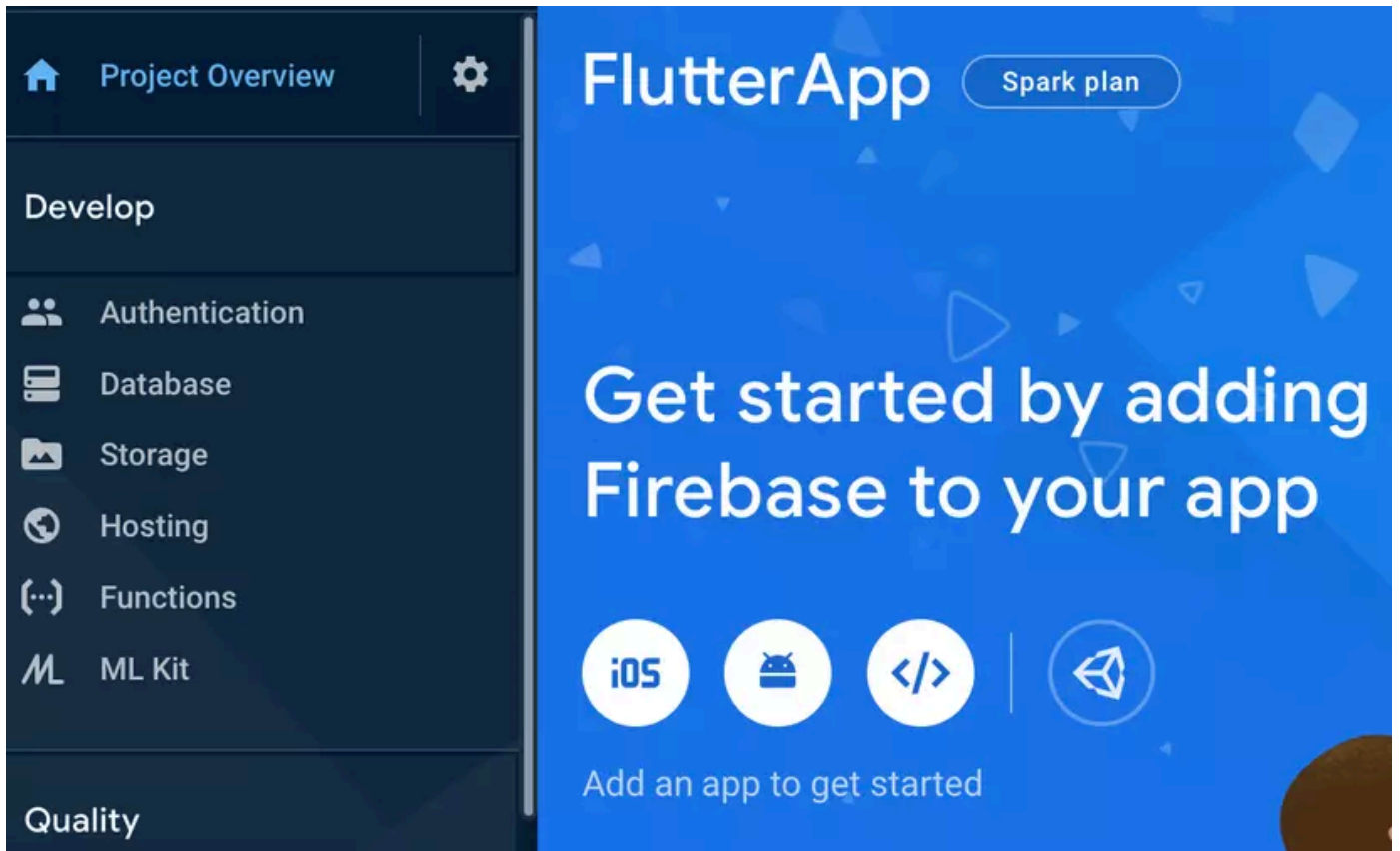
[Create project](#)

Ở phần này anh em có thể enable hoặc không chế độ Analys project của Google và sau đó nhấn chọn **Create project**. Đơn giản phải không anh em, tiếp theo mình sẽ tiến hành thêm Android và iOS vào trong project.

Thêm Android vào project Firebase:




Anh em chọn vào icon Android để thêm Android vào project Firebase mới tại nhé.



Sau đó điền các thông tin lấy ra từ project Flutter và điền vào thôi.

Register app

Android package name 

com.company.appname

App nickname (optional)

My Android App

Debug signing certificate SHA-1 (optional)

00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:(

Required for Dynamic Links, Invites, and Google Sign-In or phone number support in Auth. Edit SHA-1s in Settings.

Register app

Bao gồm các thông tin về package, SHA1 và nick name của Project sau khi hoàn thành thì click vào Register app. Bước tiếp theo thì anh em sẽ phải tải file google-services.json và import vào trong project Flutter như ảnh bên dưới.




Register app

Android package name: com.xnn.demo, App nickname: Demo

2

Download config file

Instructions for Android Studio below | [Unity](#) [C++](#)

 Download google-services.json

Switch to the **Project** view in Android Studio to see your project root directory.

Move the google-services.json file you just downloaded into your Android app module root directory.



google-services.json

[Previous](#)

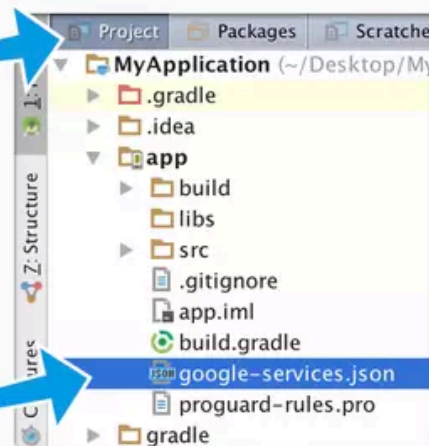
[Next](#)

3

Add Firebase SDK

4

Read the Get Started Guide for Android



Tiếp đến là cấu hình build.gradle theo hướng dẫn.

Project-level build.gradle (<project>/build.gradle):

```
buildscript {
    repositories {
        // Check that you have the following line (if not, add it):
        google() // Google's Maven repository
    }
    dependencies {
        ...
        // Add this line
        classpath 'com.google.gms:google-services:4.3.3'
    }
}

allprojects {
    ...
    repositories {
        // Check that you have the following line (if not, add it):
        google() // Google's Maven repository
        ...
    }
}
```

App-level build.gradle (<project>/<app-module>/build.gradle):

```
apply plugin: 'com.android.application'
// Add this line
apply plugin: 'com.google.gms.google-services'

dependencies {
    // add SDKs for desired Firebase products
    // https://firebase.google.com/docs/android/setup#available-libraries
}
```

Finally, press "Sync now" in the bar that appears in the IDE:

Gradle files have changed since last sync

[Sync now](#)

Chọn **Next** và **Continue** ở màn hình sau đó, thế là xong phần cấu hình cho Android.

Thêm iOS vào project Firebase:



Ở bước đầu cũng giống như Android anh em cũng click vào icon iOS thay vì icon Android.

×

Add Firebase to your iOS app

✓

Register app

iOS bundle ID: com.androidfactorem.flutterNotifications

✓

Download config file

3

Add Firebase SDK

Instructions for CocoaPods | [Download ZIP](#) [Unity](#) [C++](#)

Google services use [CocoaPods](#) to install and manage dependencies. Open a terminal window and navigate to the location of the Xcode project for your app.

Create a Podfile if you don't have one:

```
$ pod init
```

Open your Podfile and add:

```
pod 'Firebase/Core'
```

Includes Analytics by default

Save the file and run:

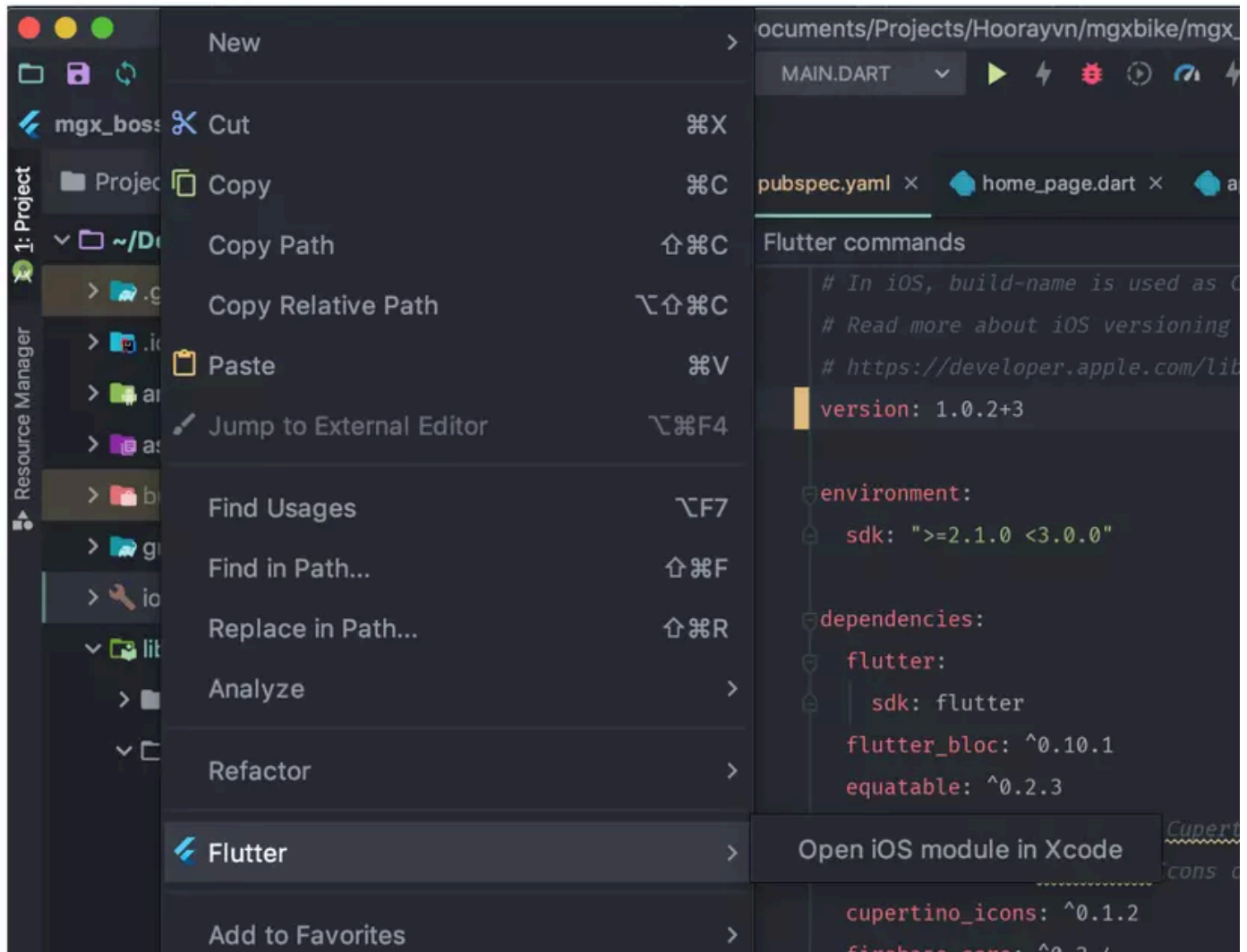
```
$ pod install
```

This creates an .xcworkspace file for your app. Use this file for all future development on your application.

Previous

Next

Để lấy Bundle id app anh em làm như sau: vào Android studio (nếu đang dùng để code Flutter) chuột phải vào folder iOS và chọn **Flutter** -> **Open module in Xcode**



Ở Xcode, chọn thư mục Runner ở Root -> trong General, các bạn sẽ tìm thấy bundle identifier, nó chính là bundle id. Sau đó quay lại Firebase console, nhập bundle Id và tìm được -> click **Register app** Tiếp theo tải file GoogleService-Info.plist ở bước kế tiếp.

× Add Firebase to your iOS app



Register app

iOS bundle ID: com.hires.leverage

2

Download config file

Instructions for Xcode below | [Unity](#) [C++](#)

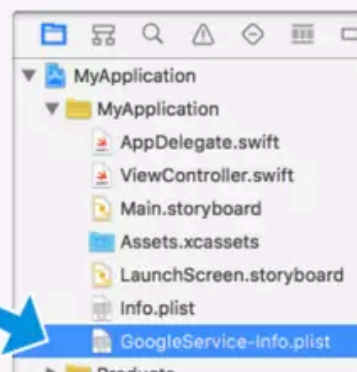


Download GoogleService-Info.plist

Move the GoogleService-Info.plist file you just downloaded into the root of your Xcode project and add it to all targets.



GoogleService-Info.plist



Previous

Next

File tải về phải được bỏ vào thư mục Runner/Runner.

× Add Firebase to your iOS app



Register app

iOS bundle ID: com.androidfactorem.flutterNotifications



Download config file



Add Firebase SDK

Instructions for CocoaPods | [Download ZIP](#) [Unity](#) [C++](#)

Google services use [CocoaPods](#) to install and manage dependencies. Open a terminal window and navigate to the location of the Xcode project for your app.

Create a Podfile if you don't have one:

```
$ pod init
```



Open your Podfile and add:

```
pod 'Firebase/Core'
```



Includes Analytics by default [?](#)

Save the file and run:

```
$ pod install
```



This creates an `.xcworkspace` file for your app. Use this file for all future development on your application.

[Previous](#)

[Next](#)

Anh em tiến hành cấu hình như hình trên.

Ở bước add SDK anh em nên build App lên điện thoại để quá trình kết nối giữa iOS và Firebase diễn ra, nếu thành công thì sẽ hiển thị theo bên dưới.

× Add Firebase to your iOS app

- ✓ Register app
iOS bundle ID: com.hires.leverage
- ✓ Download config file
- ✓ Add Firebase SDK
- ✓ Add initialization code
- 5 Run your app to verify installation

✓ Congratulations, you've successfully added Firebase to your app!

Previous

Continue to console

** Tham khảo tài liệu viblo.asia

Textfield & Năm cách sử dụng Textfield

Một widget TextField cho phép thu thập thông tin từ người dùng. Code cho TextField cơ bản đơn giản như:

```
1 | TextField()
```

Điều này tạo ra một TextField cơ bản:



Lấy thông tin từ TextField

Vì TextField không có ID như trong Android, văn bản không thể được truy xuất theo yêu cầu và thay vào đó phải được lưu trữ trong một biến khi thay đổi hoặc sử dụng bộ điều khiển.

1. Cách dễ nhất để làm điều này là sử dụng phương thức `onChanged` và lưu trữ giá trị hiện tại trong một biến đơn giản. Đây là mã mẫu cho nó:

```
1 String value = "";
2 TextField(
3   onChanged: (text) {
4     value = text;
5   },
6 )
```

1. Cách thứ hai để làm điều này là sử dụng `TextEditingController`. Bộ điều khiển được gắn vào TextField và cũng cho phép chúng tôi nghe và điều khiển văn bản của TextField.

```
1 TextEditingController controller = TextEditingController();
2 TextField(
3   controller: controller,
4 )
```

Và chúng ta có thể lắng nghe những thay đổi bằng cách sử dụng

```
1 controller.addListener(() {
2   // Do something here
3 });
```

Và nhận hoặc đặt giá trị bằng cách sử dụng

```
1 print(controller.text); // Print current value
2 controller.text = "Demo Text"; // Set new value
```

Các Callback khác từ TextField

Tiện ích TextField cũng cung cấp các callback khác như

1. `onEditingCompleted`
2. `onSubmitted`



```
1 onEditingComplete: () {},  
2 onSubmitted: (value) {},
```

Đây là các callback được gọi trên các hành động như khi người dùng nhấn vào nút "Done" button trên bàn phím iOS.

Làm việc với focus trong TextFields

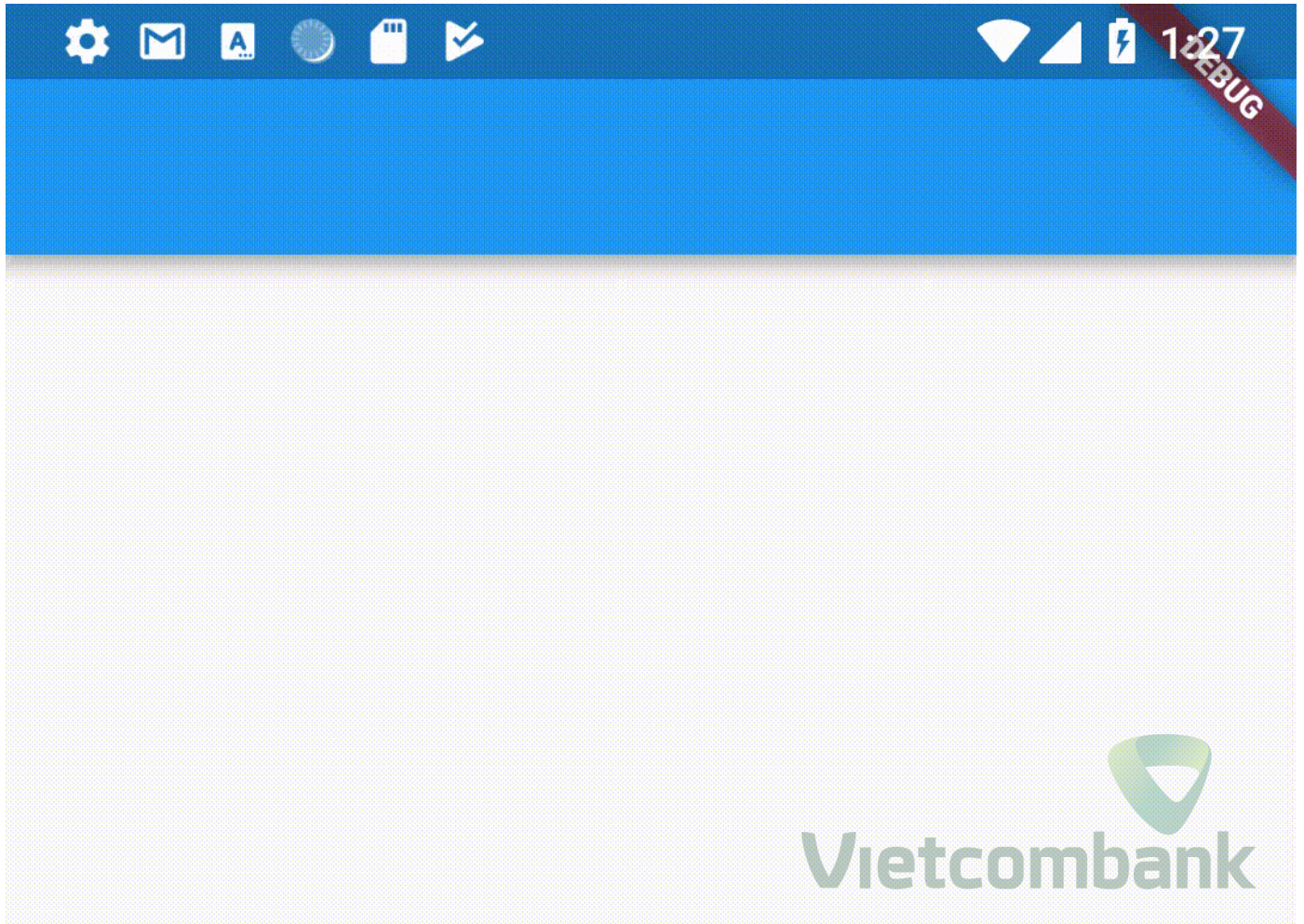
Có một TextField được `focus` có nghĩa là có một TextField hoạt động và mọi đầu vào từ bàn phím sẽ dẫn đến dữ liệu được nhập vào TextField đã `focus` đó.

1. Làm việc với autofocus

Để `autofocus` trên TextField khi tiện ích được tạo, hãy đặt trường `autofocus` thành `true`.

```
1 TextField(  
2   autofocus: true,  
3 ),
```

Điều này đặt focus vào TextField theo mặc định.



Next Page

2. Làm việc với các thay đổi focus tùy chỉnh

Điều gì sẽ xảy ra nếu chúng ta muốn thay đổi `focus` theo nhu cầu và không chỉ `autofocus`? Vì chúng ta cần một số cách để `focus` `TextField` tiếp theo mà chúng ta muốn, chúng ta đính kèm `FocusNode` vào `TextField` và sử dụng nó để chuyển `focus`.

```
1 // Initialise outside the build method
2 FocusNode nodeOne = FocusNode();
3 FocusNode nodeTwo = FocusNode();
4 // Do this inside the build method
5 TextField(
```

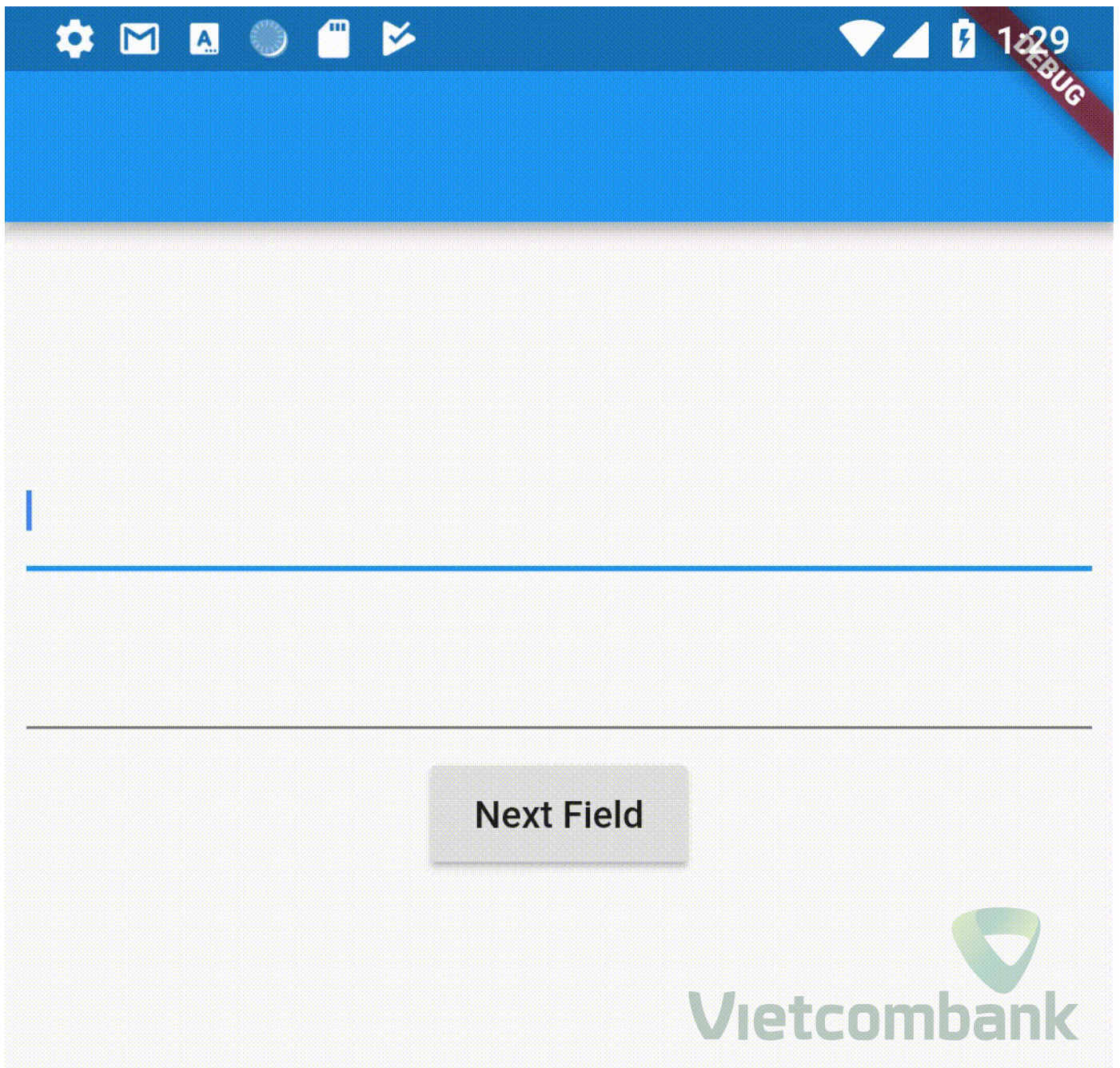


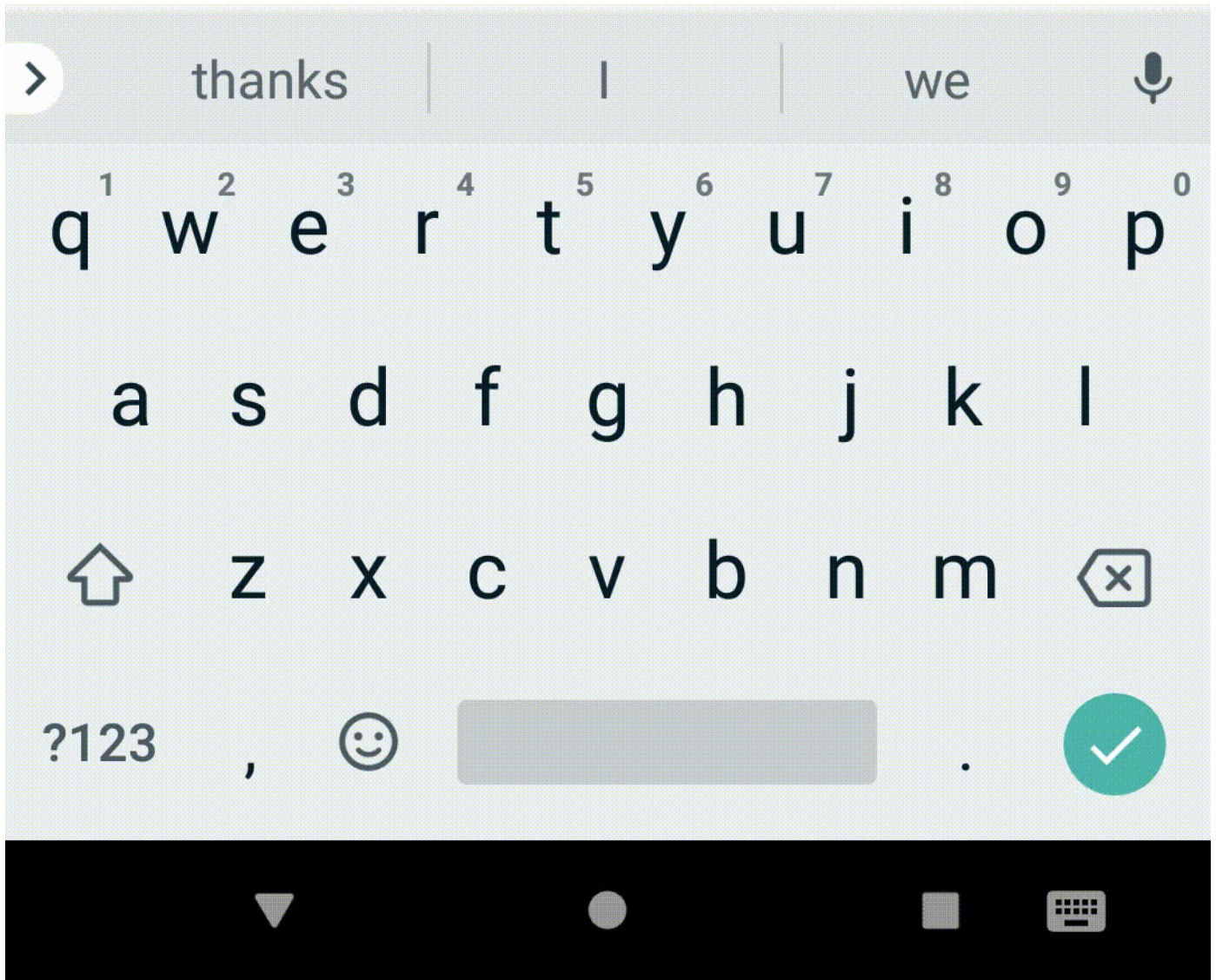

```

6     focusNode: nodeOne,
7   ),
8   TextField(
9     focusNode: nodeTwo,
10  ),
11  RaisedButton(
12    onPressed: () {
13      FocusScope.of(context).requestFocus(nodeTwo);
14    },
15    child: Text("Next Field"),
16  ),

```

Chúng ta tạo hai nút `focus` và đính kèm chúng vào TextFields. Khi nhấn nút, chúng tôi sử dụng `FocusScope` để yêu cầu `focus` cho TextField tiếp theo.





Thay đổi thuộc tính bàn phím cho TextFields

TextField trong Flutter cũng cho phép bạn tùy chỉnh các thuộc tính liên quan đến bàn phím.

1. Loại Keyboard

TextField cho phép bạn tùy chỉnh loại bàn phím hiển thị khi TextField được đưa vào tiêu điểm. Chúng tôi thay đổi thuộc tính keyboardType cho điều này.

```
1 TextField(  
2   keyboardType: TextInputType.number,  
3 ),
```

Các loại là:

1. **TextInputType.text** (Normal complete keyboard)
2. **TextInputType.number** (A numerical keyboard)
3. **TextInputType.emailAddress** (Normal keyboard with an "@")
4. **TextInputType.datetime** (Numerical keyboard with a "/" and ":")

5. **TextInputType.numberWithOptions** (Numerical keyboard with options to enable signed and decimal mode)
6. **TextInputType.multiline** (Optimises for multi-line information)

2. TextInputAction

Thay đổi `textInputAction` của `TextField` cho phép bạn thay đổi nút hành động của chính bàn phím. Ví dụ:

```
1 TextField(  
2   textInputAction: TextInputAction.continueAction,  
3 ),
```

Trường hợp này nút "Done" được thay thế bằng nút "Continue".



hoặc:

```
1 TextField(  
2   textInputAction: TextInputAction.send,  
3 ),
```

kết quả:



3. Autocorrect

Bật hoặc tắt tự động sửa cho TextField cụ thể. Sử dụng trường tự động sửa để đặt cái này.

```
1 TextField(  
2     autocorrect: false,  
3 ),
```

Điều này cũng sẽ vô hiệu hóa các đề xuất.

4. Text Capitalization

TextField cung cấp một vài tùy chọn về cách viết hoa chữ cái trong đầu vào từ người dùng.

```
1 TextField(  
2     textCapitalization: TextCapitalization.sentences,  
3 ),
```

Các loại là:

1. TextCapitalization.sentences

Đây là loại viết hoa bình thường mà chúng tôi mong đợi, chữ cái đầu tiên của mỗi câu được viết hoa.

Hello world. Demo text.

2. TextCapitalization.characters

Viết hoa tất cả các ký tự trong câu.

THE NEW VERSION

3. TextCapitalization.words

Viết hoa chữ cái đầu tiên của mỗi từ.

Demo Text

Tùy chọn kiểu văn bản, căn chỉnh và con trỏ

Flutter cho phép tùy chỉnh liên quan đến kiểu dáng và căn chỉnh văn bản bên trong TextField cũng như con trỏ bên trong TextField.

Căn chỉnh text bên trong TextField

Sử dụng thuộc tính `textAlign` để điều chỉnh vị trí con trỏ bên trong TextField.

```
1 TextField(  
2   textAlign: TextAlign.center,  
3 ),
```

Điều này khiến con trỏ và văn bản bắt đầu ở giữa TextField.



Hello

Điều này có các thuộc tính căn chỉnh thông thường: **start, end, left, right, center, justify**.

Tạo kiểu văn bản bên trong TextField

Chúng ta sử dụng thuộc tính kiểu để thay đổi cách văn bản bên trong TextField. Sử dụng nó để thay đổi màu sắc, kích thước phông chữ, v.v ... Điều này tương tự với thuộc tính kiểu trong tiện ích Văn bản, vì vậy chúng tôi sẽ không mất quá nhiều thời gian để khám phá nó.

```
1 TextField(  
2   style: TextStyle(color: Colors.red, fontWeight: FontWeight.w300),  
3 ),
```

Hello World

Thay đổi con trỏ trong TextField

Con trỏ được tùy chỉnh trực tiếp từ tiện ích TextField.

Bạn được phép thay đổi màu con trỏ, chiều rộng và bán kính của các góc. Ví dụ, ở đây tôi tạo một con trỏ màu đỏ hình tròn mà không có lý do rõ ràng.

```
1 TextField(  
2   cursorColor: Colors.red,  
3   cursorRadius: Radius.circular(16.0),  
4   cursorWidth: 16.0,  
5 ),
```



Kiểm soát kích thước và độ dài tối đa trong TextField

TextFields có thể kiểm soát số lượng ký tự tối đa được viết bên trong nó, số lượng dòng tối đa và mở rộng khi văn bản được nhập.

Kiểm soát ký tự tối đa

```
1 TextField(  
2   maxLength: 4,  
3 ),
```

abc|

3/4

Bằng cách đặt thuộc tính `maxLength`, độ dài tối đa được thi hành và bộ đếm được thêm theo mặc định vào TextField.

Tạo một TextField có thể mở rộng

Đôi khi, chúng ta cần một TextField mở rộng khi một dòng kết thúc. Trong Flutter, nó hơi kỳ lạ (nhưng dễ) để làm. Để làm điều này, chúng tôi đặt `maxLines` thành `null`, theo mặc định là 1. Cài đặt thành `null` không phải là điều mà chúng tôi rất quen thuộc nhưng tuy nhiên nó rất dễ thực hiện.

Lorem ipsum dolor sit amet, consectetur
adipiscing elit, sed do eiusmod tempor|

Lưu ý: Đặt `maxLines` thành giá trị trực tiếp sẽ mở rộng nó thành số dòng đó theo mặc định.

Văn bản che khuất

Để che khuất văn bản trong TextField, hãy đặt `obscureText` thành `true`. Thuộc tính này dùng cho các TextField nhập password

.....

Và cuối cùng, trang trí TextField

Cho đến bây giờ chúng ta tập trung vào các tính năng Flutter cung cấp cho đầu vào. Bây giờ chúng ta sẽ chuyển sang thực sự thiết kế một TextField ưa thích và không nói không với nhà thiết kế của bạn. Để trang trí TextField, chúng ta sử dụng thuộc tính trang trí cần lấy `InputDecoration`. Vì lớp `InputDecoration` là rất lớn, chúng ta sẽ cố gắng nhanh chóng vượt qua hầu hết các thuộc tính quan trọng.

Sử dụng các thuộc tính gợi ý và nhãn để cung cấp thông tin cho người dùng

Cả gợi ý và nhãn là các chuỗi giúp người dùng hiểu thông tin được nhập vào TextField. Sự khác biệt là một gợi ý sẽ biến mất khi người dùng bắt đầu nhập trong khi nhãn nổi trên TextField.

Demo Text

Hint

Demo Text

|

Label

Bạn có thể thêm các “icon”, “prefixIcon” and “suffixIcon”

Bạn có thể thêm các biểu tượng trực tiếp vào TextFields. Bạn cũng có thể sử dụng `prefixText` và hậu tố cho văn bản thay thế.

```
1 TextField(  
2   decoration: InputDecoration(  
3     icon: Icon(Icons.print)  
4   ),  
5 ),
```



Icon using the icon property

```
1 TextField(  
2   decoration: InputDecoration(  
3     prefixIcon: Icon(Icons.print)  
4   ),  
5 ),
```



Vietcombank

Icon using the prefixIcon property

Tương tự cho bất kỳ widget nào khác, dùng “prefix” thay thế cho “prefixIcon”

Để sử dụng tiện ích chung thay vì biểu tượng, hãy sử dụng trường tiền tố. Một lần nữa, không có lý do rõ ràng, hãy để Thêm một chỉ báo tiến trình vòng tròn trong TextField.

```
1 TextField(  
2   decoration: InputDecoration(  
3     prefix: CircularProgressIndicator(),  
4   ),  
5 ),
```



Mỗi thuộc tính như gợi ý, nhãn, v.v đều có các trường kiểu tương ứng

Để tạo kiểu cho một gợi ý, hãy sử dụng một gợi ý. Để tạo kiểu cho nhãn, hãy sử dụng nhãnStyle.

```
1 TextField(  
2   decoration: InputDecoration(  
3     hintText: "Demo Text",  
4     hintStyle: TextStyle(fontWeight: FontWeight.w300, color: Colors.red)  
5   ),  
6 ),
```

Demo Text



Lưu ý: Mặc dù tôi đã thực hiện nó trong ví dụ này, nhưng nhìn chung không thay đổi màu gợi ý vì nó gây nhầm lẫn cho người dùng.

Sử dụng `helperText` của người dùng nếu bạn không muốn có nhãn nhưng bạn muốn có một thông điệp bền bỉ cho người dùng.

```
1 TextField(  
2   decoration: InputDecoration(  
3     helperText: "Hello"  
4   ),  
5 ),
```

Hello

Sử dụng “`decoration: null`” hoặc `InputDecoration.collapsed` để loại bỏ underline mặc định của `TextField`

Sử dụng chúng để xóa phần gạch chân mặc định trên `TextField`.

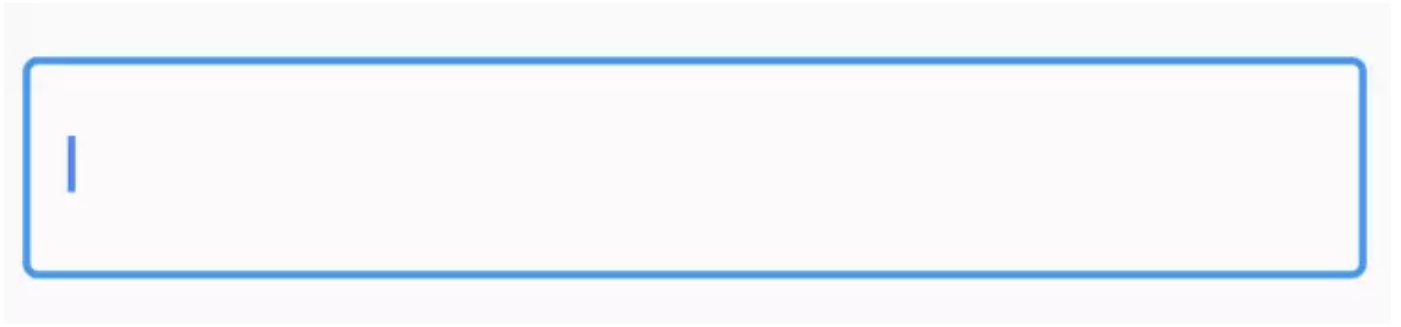
```
1 TextField(  
2   decoration: InputDecoration.collapsed(hintText: "")  
3 ),
```

abcd|

Thêm border vào `TextField`

```
1 TextField(  
2   decoration: InputDecoration(  
3     border: OutlineInputBorder()  
4   )  
5 ),
```





Có một số lượng lớn trang trí bạn có thể làm hơn nữa, nhưng chúng ta có thể đi sâu vào mọi thứ trong một bài viết. Nhưng tôi hy vọng điều này làm cho nó rõ ràng để hiểu cách dễ dàng tùy chỉnh Flutter TextFields.

** tham khảo tài liệu tại viblo.asia