**BỘ GIÁO DỤC & ĐÀO TẠO**

**TRƯỜNG ĐẠI HỌC SÀI GÒN**

**KHOA CÔNG NGHỆ THÔNG TIN**

--🙢🕮🙠--

**A picture containing text, device, gauge

Description automatically generated**

**BÁO CÁO MÔN HỌC**

**CÁC CÔNG NGHỆ LẬP TRÌNH HIỆN ĐẠI**

**ĐỀ TÀI**

**TÌM HIỂU FLUTTER**

**Nhóm 17**

3120410457 Lê Bảo Tài   
3120410467 Võ Minh Tấn  
3120410297 Trần Nguyên Lộc  
3120410471 Trịnh Hùng Thái

**Giảng viên phụ trách ThS. PHẠM THI VƯƠNG**

**TP.Hồ Chí Minh, tháng 5 năm 2024**

**MỤC LỤC**

[LỜI CẢM ƠN iii](#_Toc163136222)

[LỜI CAM ĐOAN iv](#_Toc163136223)

[DANH MỤC HÌNH ẢNH v](#_Toc163136224)

[DANH MỤC BẢNG BIỂU viii](#_Toc163136225)

[DANH MỤC THUẬT NGỮ ix](#_Toc163136226)

[LỜI MỞ ĐẦU 1](#_Toc163136227)

[CHƯƠNG 1. TỔNG QUAN VỀ FLUTTER 2](#_Toc163136228)

[1.1. Flutter là gì 2](#_Toc163136229)

[1.2. Lịch sử phát triển của Flutter 3](#_Toc163136230)

[1.3. Các tính năng chính của Flutter 4](#_Toc163136231)

[1.4. Kiến trúc của Flutter 7](#_Toc163136232)

[1.5. Ưu điểm và nhược điểm của Flutter 11](#_Toc163136233)

[1.6. Một số thông tin liên quan khác 12](#_Toc163136234)

[1.6.1. Các nguồn tài liệu khoá học Flutter 12](#_Toc163136235)

[1.6.1.1. Khoá học trả phí 12](#_Toc163136236)

[1.6.1.2. Khoá học miễn phí 15](#_Toc163136237)

[1.6.2. Các thông tin tuyển dụng về Flutter 17](#_Toc163136238)

[1.7. Cài đặt và cấu hình cần thiết cho việc lập trình với Flutter 19](#_Toc163136239)

[1.7.1. Flutter SDK 19](#_Toc163136240)

[1.7.2. IDE 22](#_Toc163136241)

[1.7.3. Khởi chạy chương trình đầu tiên 27](#_Toc163136242)

[CHƯƠNG 2. CHI TIẾT VỀ FLUTTER 32](#_Toc163136243)

[2.1. Những kiến thức cơ bản về Flutter 32](#_Toc163136244)

[2.1.1. Tổng quan về Widgets và một số Widgets cần phải biết 32](#_Toc163136245)

[2.1.1.1. Khái niệm về Widget trong Flutter 32](#_Toc163136246)

[2.1.1.2. Các loại Widget con 34](#_Toc163136247)

[2.1.1.3. Phân loại Widget State 44](#_Toc163136248)

[2.1.2. Bố cục giao điện (Layout) 48](#_Toc163136249)

[2.1.2.1. Khái niệm bố cục và bố trí trong Flutter 48](#_Toc163136250)

[2.1.2.2. Một số Widget bố cục 53](#_Toc163136251)

[2.1.2.3. Xây dựng thử một bố cục phức tạp 68](#_Toc163136252)

[2.1.3. Cử chỉ giao diện (Gestures) 74](#_Toc163136253)

[2.1.3.1. Con trỏ 74](#_Toc163136254)

[2.1.3.2. Cử chỉ 75](#_Toc163136255)

[2.1.3.3. Bắt sự kiện cử chỉ 77](#_Toc163136256)

[2.1.4. Quản lý trạng thái (State) 80](#_Toc163136257)

[2.1.4.1. Khái niệm 81](#_Toc163136258)

[2.1.4.2. Trạng thái tức thời (Ephemeral State) 82](#_Toc163136259)

[2.1.4.3. Trạng thái ứng dụng (App State) 83](#_Toc163136260)

[2.1.4.4. Quản lí trạng thái (State) bằng thư viện Provider 84](#_Toc163136261)

[2.1.4.5. So sánh Quản lý trạng thái bằng cách thông thường và Quản lý trạng thái bằng thư viện Provider 90](#_Toc163136262)

[2.1.5. Điều hướng màn hình 92](#_Toc163136263)

[2.1.5.1. Điều hướng bằng Routes 92](#_Toc163136264)

[2.1.5.2. Điều hướng bằng RouteName 96](#_Toc163136265)

[2.1.5.3. So sánh giữa Route và RouteName 98](#_Toc163136266)

[2.2. Những Widgets cần biết và phải hiểu 98](#_Toc163136267)

[CHƯƠNG 3. CÁC CHƯƠNG TRÌNH DEMO 100](#_Toc163136268)

[DANH MỤC THAM KHẢO 101](#_Toc163136269)

# LỜI CẢM ƠN

Để hoàn thành được báo cáo cáo cho môn Lập trình hiện đại này, chúng em xin chân thành cảm ơn thầy ThS.Phạm Thi Vương khoa Công Nghệ Thông Tin trường Đại học Sài Gòn đã tạo cơ hội cho chúng em được học tập, nghiên cứu, hợp tác làm việc nhóm và tích lũy kiến thức để thực hiện và hoàn thành báo cáo đồ án này. Xin cảm ơn thầy đã luôn nhận xét và theo dõi tiến trình trong suốt thời gian làm bài báo cáo.

Do kiến thức của các thành viên còn nhiều hạn chế và thiếu kinh nghiệm nên nội dung báo cáo khó tránh khỏi những thiếu sót. Chúng em rất mong nhận được những lời góp ý thêm đến từ thầy để được hoàn thiện hơn.

# LỜI CAM ĐOAN

Nhóm chúng em xin cam đoan đề tài “Tìm hiểu Flutter” là công trình nghiên cứu và tìm hiểu của riêng nhóm chúng em

# DANH MỤC HÌNH ẢNH

[Hình 1. 1. Flutter là gì? 2](#_Toc162851356)

[Hình 1. 2. Nguồn gốc hình thành và điểm độc đáo của Flutter 4](#_Toc162851357)

[Hình 1. 3. Các tính năng chính của Flutter 7](#_Toc162851358)

[Hình 1. 4. Kiến trúc tổng thể của của Flutter 8](#_Toc162851359)

[Hình 1. 5. Mô hình giải phẩu của một ứng dụng Flutter 9](#_Toc162851360)

[Hình 1. 6. Một số khoá học Flutter trả phí trên Udemy 13](#_Toc162851361)

[Hình 1. 7. Khoá học Flutter trả phí trên CodeGym 13](#_Toc162851362)

[Hình 1. 8. Khoá học lập trình Flutter miễn phí trên Youtube: TinCoder 14](#_Toc162851363)

[Hình 1. 9. Khoá học lập trình Flutter miễn phí trên Youtube: Dear Programmer 15](#_Toc162851364)

[Hình 1. 10. Khoá học lập trình Flutter miễn phí trên web: CafeDev 15](#_Toc162851365)

[Hình 1. 11. Một số tuyển dụng lập trình viên Flutter trên trang Topdev.vn 16](#_Toc162851366)

[Hình 1. 12. Một số tuyển dụng lập trình viên Flutter trên trang topcv.vn 17](#_Toc162851367)

[Hình 1. 13. Tải gói Flutter SDK mới nhất từ trang chủ chính thức của Flutter 18](#_Toc162851368)

[Hình 1. 14. Giải nén và cài đặt gói Flutter SDK tại một đường dẫn bất kỳ 19](#_Toc162851369)

[Hình 1. 15. Truy cập Enviroment Variables từ thanh tìm kiếm 19](#_Toc162851370)

[Hình 1. 16. Chọn Edit Path của User Variables 20](#_Toc162851371)

[Hình 1. 17. Thiết lập đường dẫn tới thư mục bin của Flutter SDK 20](#_Toc162851372)

[Hình 1. 18. Kiểm tra bằng lệnh Flutter doctor 21](#_Toc162851373)

[Hình 1. 19. Truy cập vào trang chủ android studio để cài đặt IDE 21](#_Toc162851374)

[Hình 1. 20. Cài đặt các Plugin cần thiết trên Android Studio 22](#_Toc162851375)

[Hình 1. 21. Kiểm tra Android SDK 22](#_Toc162851376)

[Hình 1. 22. Kiểm tra Dart SDK 23](#_Toc162851377)

[Hình 1. 23. Ấn vào nút Create Virtual Device để tạo máy ảo 23](#_Toc162851378)

[Hình 1. 24. Tuỳ chọn kích cỡ màn hình máy ảo 24](#_Toc162851379)

[Hình 1. 25. Tuỳ chọn phiên bản Android 24](#_Toc162851380)

[Hình 1. 26. Các tuỳ chọn còn lại và ấn Finish 25](#_Toc162851381)

[Hình 1. 27. Tạo mới một dự án Flutter 25](#_Toc162851382)

[Hình 1. 28. Thiết lập tên dự án, đường dẫn, mô tả, loại dự án và các thiết lập khác cho dự án Flutter đầu tiên 26](#_Toc162851383)

[Hình 1. 29. Những dòng code mặc định của một dự án Flutter 26](#_Toc162851384)

[Hình 1. 30. Khởi tạo máy ảo cho ứng dụng Flutter đầu tiên 27](#_Toc162851385)

[Hình 1. 31. Android Studio biên dịch code và cài đặt ứng dụng Flutter lên máy ảo 28](#_Toc162851386)

[Hình 1. 32. Máy ảo khởi động chương trình Flutter đầu tiên 29](#_Toc162851387)

[Hình 1. 33. Android Studio đặt chương trình đang chạy vào trạng thái Hot Reload 29](#_Toc162851388)

[Hình 2. 1. Một ví dụ về cây Widgets trong Flutter 32](#_Toc163135925)

[Hình 2. 2. Hình ảnh minh hoạ cho Widget Text 34](#_Toc163135926)

[Hình 2. 3. Hình ảnh minh hoạ cho một số Widget Button 36](#_Toc163135927)

[Hình 2. 4. Đường dẫn hình ảnh cấu hình trong pubspec.yaml trong dự án thực tế 37](#_Toc163135928)

[Hình 2. 5. Hình ảnh minh hoạ cho Widget Image 38](#_Toc163135929)

[Hình 2. 6. Hình ảnh minh hoạ cho Widget Icon 39](#_Toc163135930)

[Hình 2. 7. Hình ảnh minh hoạ cho Widget Column 41](#_Toc163135931)

[Hình 2. 8. Hình ảnh minh hoạ cho Widget Row 43](#_Toc163135932)

[Hình 2. 9. Sắp xếp bố cục trong Flutter 49](#_Toc163135933)

[Hình 2. 10. Minh hoạ về cây bố cục trong Flutter 49](#_Toc163135934)

[Hình 2. 11. Một ví dụ về xây dựng bố cục của một ứng dụng trong thực tế 53](#_Toc163135935)

[Hình 2. 12. Ví dụ về cách sử dụng Widget FittedBox 59](#_Toc163135936)

[Hình 2. 13. Ví dụ về cách sử dụng Widget OverflowBox 62](#_Toc163135937)

[Hình 2. 14. Kết hợp giữa Widget Row và Column 65](#_Toc163135938)

[Hình 2. 15. Một ví dụ minh hoạ của Widget Stack được sử dụng như thế nào 68](#_Toc163135939)

[Hình 2. 16. Bố cục của ví dụ ProductBox Widget 73](#_Toc163135940)

[Hình 2. 17. Kết quả chạy thử giả lập trong ví dụ bố cục 73](#_Toc163135941)

[Hình 2. 18. Hình ảnh minh hoạ cử chỉ trong Flutter 76](#_Toc163135942)

[Hình 2. 19. Giao diện bắt sự kiện cử chỉ 80](#_Toc163135943)

[Hình 2. 20. Một giải nghĩa cho Quản lý trạng thái trên Flutter 81](#_Toc163135944)

[Hình 2. 21. Ví dụ minh hoạ cho trạng thái ứng dụng 84](#_Toc163135945)

[Hình 2. 22. Demo minh hoạ cho quản lý trạng thái (State) bằng thư viện Provider 90](#_Toc163135946)

[Hình 2. 23. Màn hình thứ 1 có nút bấm để di chuyển tới màn hình thứ 2 95](#_Toc163135947)

[Hình 2. 24. Màn hình thứ 2 có nút bấm để quay ngượic về màn hình thứ 1 96](#_Toc163135948)

# DANH MỤC BẢNG BIỂU

# DANH MỤC THUẬT NGỮ

# LỜI MỞ ĐẦU

**Lý do chọn đề tài**

Hiện nay số lượng điện thoại ngày càng đông và công nghệ cũng ngày càng phát triển. Trong mỗi ngày thì mỗi người dành thời gian khá nhiều cho điện thoại từ giải trí cho đến làm việc. Từ đó ứng dụng phần mềm trên điện thoại cũng phát triển nhanh chóng. Từ đó nhóm em có quyết định chọn đề tài nghiên cứu về framework Fluter để thực hiện việc phát triển phần mềm trên điện thoại di động.

**Mục đích**

Học tập và phát triển bản thân với ngôn ngữ lập trình Dart và Framework Flutter. Xây dựng các giao diện cơ bản phục vụ cho mục đích học tập và tìm hiểu Flutter.

**Đối tượng và phạm vi**

Framework Flutter và ngôn ngữ lập trình Flutter

**Phương pháp thực hiện**

Sử dụng công nghệ Flutter, ngôn ngữ lập trình Dart.

Các công cụ như: Visual Studio Code, Android Studio.

**Cấu trúc báo cáo**

Cấu trúc của báo cáo gồm 3 phần chính:

* Chương 1. Tổng quan về Flutter
* Chương 2. Chi tiết về Flutter
* Chương 3. Các chương trình Demo

# CHƯƠNG 1. TỔNG QUAN VỀ FLUTTER

## 1.1. Flutter là gì

Nói chung, tạo một ứng dụng di động là một công việc rất phức tạp và đầy thử thách. Có rất nhiều framework có sẵn, cung cấp các tính năng tuyệt vời để phát triển các ứng dụng di động. Để phát triển các ứng dụng dành cho thiết bị di động, Android cung cấp một framework gốc dựa trên ngôn ngữ Java và Kotlin, trong khi iOS cung cấp một framework dựa trên ngôn ngữ Objective-C / Swift. Vì vậy, chúng ta cần hai ngôn ngữ và framework khác nhau để phát triển ứng dụng cho cả hai hệ điều hành. Ngày nay, để khắc phục sự phức tạp này, có một số framework đã được giới thiệu hỗ trợ cả hệ điều hành cùng với các ứng dụng dành cho máy tính để bàn. Những loại framework này được gọi là công cụ phát triển đa nền tảng.



Hình 1. 1. Flutter là gì?

Framework phát triển đa nền tảng có khả năng viết một code và có thể triển khai trên nhiều nền tảng khác nhau (Android, iOS và Máy tính để bàn). Nó tiết kiệm rất nhiều thời gian và nỗ lực phát triển của các nhà phát triển. Có một số công cụ có sẵn để phát triển đa nền tảng, bao gồm các công cụ dựa trên web, chẳng hạn như Ionic từ Drifty Co. vào năm 2013, Phonegap từ Adobe, Xamarin từ Microsoft và React Native form của Facebook. Mỗi framework này có mức độ thành công khác nhau trong ngành công nghiệp di động. Gần đây, một framework công tác mới đã được giới thiệu trong họ phát triển đa nền tảng có tên là Flutter được phát triển từ Google.

Flutter là một bộ công cụ giao diện người dùng để tạo các ứng dụng nhanh, đẹp, được biên dịch nguyên bản cho thiết bị di động, web và máy tính để bàn với một ngôn ngữ lập trình và cơ sở code duy nhất. Nó là miễn phí và code nguồn mở. Ban đầu nó được phát triển từ Google và bây giờ được quản lý theo tiêu chuẩn ECMA . Ứng dụng Flutter sử dụng ngôn ngữ lập trình Dart để tạo ứng dụng. Các phi tiêu chương trình cổ phiếu một số tính năng tương tự như ngôn ngữ lập trình khác, chẳng hạn như Kotlin và Swift, và có thể xuyên biên dịch thành code JavaScript.

Flutter chủ yếu được tối ưu hóa cho các ứng dụng di động 2D có thể chạy trên cả nền tảng Android và iOS. Chúng ta cũng có thể sử dụng nó để xây dựng các ứng dụng đầy đủ tính năng, bao gồm máy ảnh, bộ nhớ, vị trí địa lý, mạng, SDK của bên thứ ba, v.v.

## 1.2. Lịch sử phát triển của Flutter

Flutter đã trở thành framework nổi tiếng có tác dụng phát triển ứng dụng di động và web từ Google. Nền tảng được ra mắt lần đầu vào năm 2017 tại hội nghị Dart Developer Summit. Ban đầu, Flutter được thiết kế để giúp các nhà phát triển xây dựng ứng dụng di động đẹp và tương thích trên nhiều nền tảng với một mã nguồn duy nhất. Từ đó tiết kiệm thời gian và công sức cho người dùng trong quá trình phát triển.\

Ban đầu nó được biết đến với tên code “Sky” và có thể chạy trên hệ điều hành Android. Sau khi công bố Flutter, phiên bản Flutter Alpha đầu tiên (v-0.06) đã được phát hành vào tháng 5 năm 2017 .Sau đó, trong bài phát biểu quan trọng của ngày Nhà phát triển Google tại Thượng Hải, Google đã tung ra bản xem trước thứ hai của Flutter vào tháng 9 năm 2018 , đây là bản phát hành lớn cuối cùng trước phiên bản Flutter 1.0. Vào ngày 4 tháng 12 năm 2018 , phiên bản ổn định đầu tiên của framework Flutter đã được phát hành tại sự kiện Flutter Live, ký hiệu là Flutter 1.0. Bản phát hành ổn định hiện tại của framework là Flutter v1.9.1 + hotfix.6 vào ngày 24 tháng 10 năm 2019.



Hình 1. 2. Nguồn gốc hình thành và điểm độc đáo của Flutter

Tính đến thời điểm hiện tại, Flutter đã phát triển với tốc độ chóng mặt thông qua việc ra mắt các phiên bản mới. Công nghệ cung cấp các tính năng cải tiến và sửa lỗi để người dùng có được trải nghiệm tốt nhất khi phát triển ứng dụng di động và web.

## 1.3. Các tính năng chính của Flutter

**Hot Reload (Tải lại nóng):**

Một trong những tính năng nổi bật của Flutter là khả năng Hot Reload. Các nhà phát triển có thể ngay lập tức xem những thay đổi họ thực hiện trên code được phản ánh trên giao diện ứng dụng, mà không cần phải build lại toàn bộ ứng dụng. Điều này giúp tăng tốc đáng kể quá trình phát triển và khuyến khích thử nghiệm cũng như lặp lại.

**Widget (Các khối dựng):**

Flutter được xây dựng dựa trên khái niệm widget, là những khối xây dựng để tạo các thành phần UI. Mọi thứ trong Flutter đều là một widget, từ nút bấm và văn bản đến bố cục và hoạt ảnh. Cách tiếp cận dựa trên widget này giúp đơn giản hóa việc phát triển UI và cho phép các nhà phát triển dễ dàng tạo ra các giao diện người dùng phức tạp.

**Phát triển đa nền tảng:**

Flutter cho phép các nhà phát triển viết một codebase duy nhất chạy trên nhiều nền tảng, bao gồm iOS, Android, web và desktop. Điều này giúp loại bỏ sự cần thiết phải duy trì các codebase riêng biệt cho các nền tảng khác nhau, tiết kiệm thời gian và công sức.

**Giao diện người dùng biểu cảm:**

Bộ widget được thiết kế sẵn phong phú và các thành phần có thể tùy chỉnh của Flutter cho phép các nhà phát triển tạo ra giao diện người dùng đẹp mắt và tương tác. Điều này giúp tạo ra các ứng dụng có giao diện người dùng nhất quán trên các nền tảng.

**Hiệu suất nhanh:**

Ứng dụng Flutter được biên dịch thành mã ARM gốc, đảm bảo hiệu suất cao và giảm thiểu chi phí. Việc không có cầu nối giữa ứng dụng và nền tảng cũng góp phần tăng tốc độ thực thi.

**Widget Material Design và Cupertino:**

Flutter cung cấp cả widget Material Design (dành cho Android) và widget Cupertino (dành cho iOS) để đảm bảo các ứng dụng tuân theo các nguyên tắc thiết kế riêng cho từng nền tảng. Điều này giúp đạt được trải nghiệm người dùng giống như bản địa.

**Hỗ trợ hoạt ảnh phong phú:**

Flutter cung cấp một thư viện hoạt ảnh mạnh mẽ cho phép các nhà phát triển dễ dàng tạo các hoạt ảnh mượt mà và phức tạp. Tính năng này rất cần thiết để nâng cao sự tương tác của người dùng và tạo giao diện ứng dụng năng động.

**Truy cập vào các tính năng gốc:**

Mặc dù Flutter là một framework độc lập, nó cũng cung cấp các plugin cho phép các nhà phát triển truy cập vào các tính năng của thiết bị gốc như camera, vị trí, cảm biến, v.v. Điều này đảm bảo rằng các nhà phát triển có thể tận dụng toàn bộ khả năng của nền tảng bên dưới.

**Hỗ trợ cộng đồng mạnh mẽ:**

Flutter có một cộng đồng các nhà phát triển đang phát triển nhanh chóng, những người tích cực đóng góp vào sự phát triển của nó. Điều này dẫn đến vô số tài nguyên, gói và plugin sẵn có, giúp việc giải quyết các nhu cầu phát triển khác nhau trở nên dễ dàng hơn.

**Quốc tế hóa và bản địa hóa:**

Flutter cung cấp hỗ trợ tích hợp cho quốc tế hóa và bản địa hóa, cho phép các ứng dụng dễ dàng được dịch sang nhiều ngôn ngữ và thích ứng với các khu vực khác nhau.

**Công cụ kiểm thử và gỡ lỗi:**

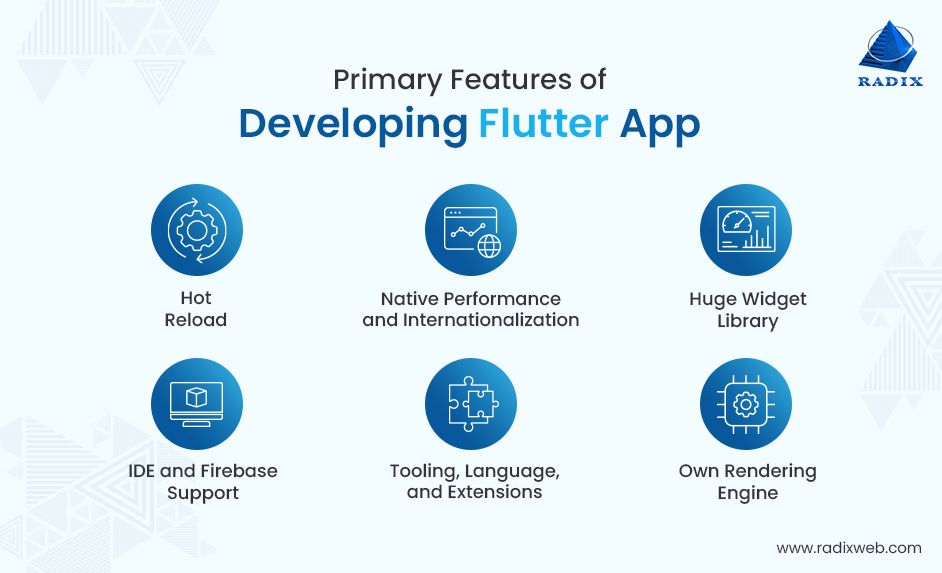
Flutter bao gồm các công cụ kiểm thử và gỡ lỗi mạnh mẽ giúp các nhà phát triển xác định các vấn đề và tối ưu hóa code của họ. Kết hợp với Hot Reload, điều này dẫn đến phát triển hiệu quả và khắc phục lỗi nhanh hơn.

**Khả năng mở rộng và bảo trì:**

Kiến trúc mô-đun và dựa trên widget của Flutter cho phép các nhà phát triển xây dựng các codebase có khả năng mở rộng và bảo trì. Điều này đặc biệt hữu lợi cho các dự án yêu cầu cập nhật và cải tiến liên tục.

**Kết luận:**

Các tính năng của Flutter giúp các nhà phát triển tạo ra các ứng dụng đa nền tảng chất lượng cao với hiệu suất ấn tượng và giao diện người dùng tuyệt đẹp. Tính linh hoạt, dễ sử dụng và cộng đồng hỗ trợ mạnh mẽ của nó biến Flutter trở thành lựa chọn lý tưởng cho các nhà phát triển đang tìm kiếm các giải pháp hiệu quả và thiết thực cho phát triển ứng dụng hiện đại. Khi framework Flutter tiếp tục phát triển, nó có khả năng giới thiệu nhiều tính năng thú vị hơn nữa, giúp nâng cao hơn nữa trải nghiệm phát triển và khả năng



Hình 1. 3. Các tính năng chính của Flutter

## 1.4. Kiến trúc của Flutter

Kiến trúc Flutter gồm 3 thành phần chính:

* 1. Framework
     + Là nền tảng cơ bản cung cấp các API, widget và công cụ cốt lõi để xây dựng ứng dụng Flutter.
     + Trách nhiệm phát triển và duy trì ứng dụng.
     + Cung cấp các widget để xây dựng giao diện người dùng.
     + Bao gồm các thư viện hỗ trợ cho các chức năng như:
       1. Networking
       2. State Management
       3. Animations
       4. Material Design
       5. Cupertino Design
  2. Engine
     + Là trình kết xuất chịu trách nhiệm hiển thị giao diện người dùng (UI) của ứng dụng.
     + Phân tích mã Dart và tạo các lệnh đồ họa tương ứng với nền tảng hệ điều hành cụ thể.
     + Cung cấp khả năng render giao diện người dùng.
     + Viết bằng C++ để đảm bảo hiệu suất cao.
     + Có thể sử dụng GPU để tăng tốc độ render.
  3. Platform
     + Cung cấp các API để truy cập các chức năng của thiết bị.
     + Là cầu nối giữa framework và nền tảng hệ điều hành cụ thể. Cho phép Flutter hoạt động trên nhiều nền tảng khác nhau như iOS, Android, Web, Windows, macOS, Linux

**Mối quan hệ tương tác:**

1. Framework tương tác với Engine:

* Framework cung cấp các widget và API để định nghĩa UI của ứng dụng.
* Engine phân tích các widget này và tạo các lệnh đồ họa tương ứng.

1. Engine tương tác với Embedder:

* Engine gửi các lệnh đồ họa cho Embedder để hiển thị trên màn hình thiết bị.
* Embedder sử dụng các API hệ điều hành để hiển thị các lệnh đồ họa này.

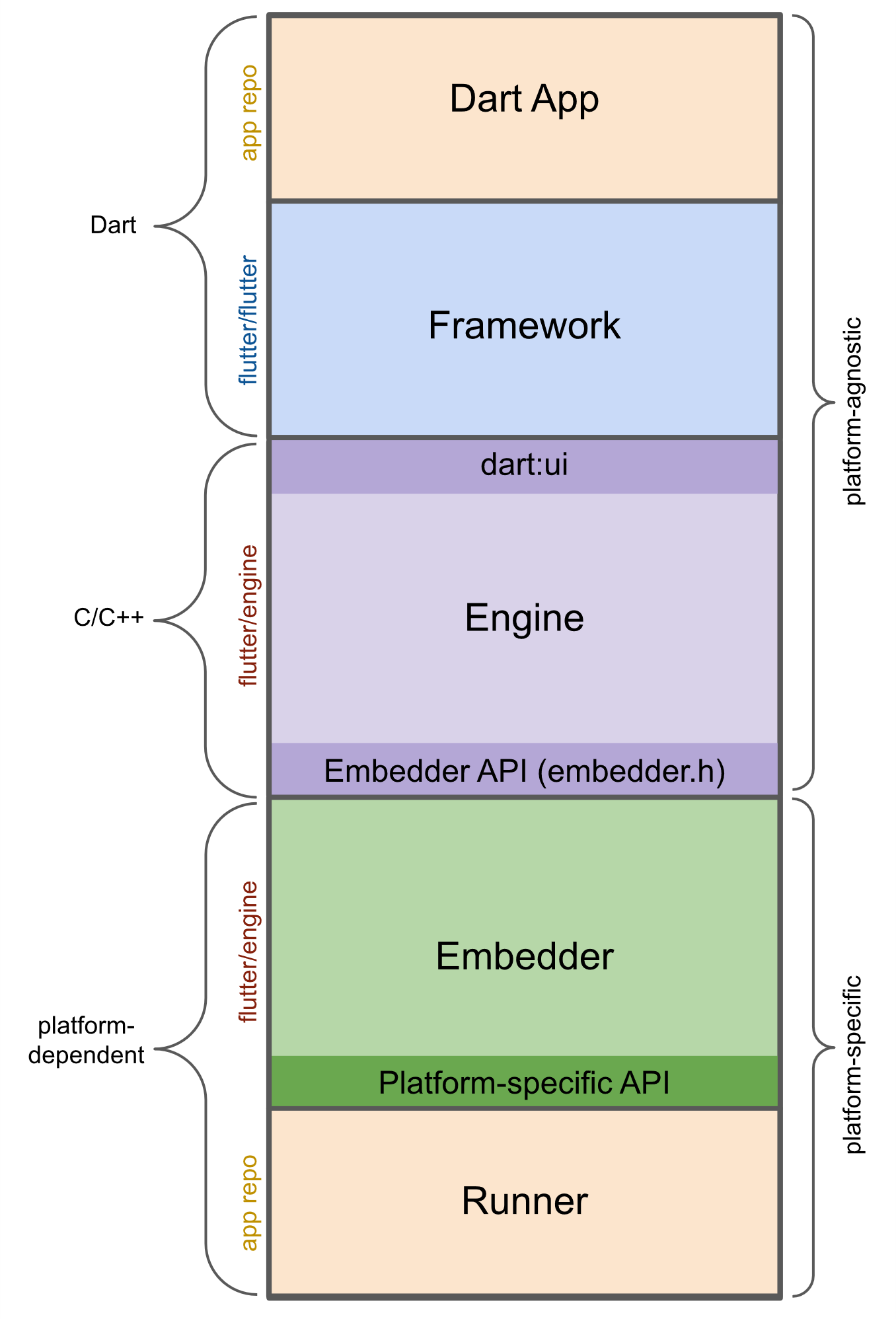
1. Embedder tương tác với hệ điều hành:

* Embedder truy cập các chức năng hệ điều hành thông qua các API được cung cấp.
* Ví dụ: truy cập camera, bộ nhớ, cảm ứng, v.v.

Architectural
diagram

Hình 1. 4. Kiến trúc tổng thể của của Flutter

Bên cạnh đó, ta còn có cả mô hình giải phẫu của một ứng dụng Flutter được mô tả như sau:



Hình 1. 5. Mô hình giải phẩu của một ứng dụng Flutter

1. Dart App

* Ghép các widget thành giao diện người dùng mong muốn.
* Thực thi logic nghiệp vụ.
* Thuộc sở hữu của nhà phát triển ứng dụng.

1. Framework
   * Cung cấp API cấp cao để xây dựng các ứng dụng chất lượng cao (ví dụ: widget, kiểm tra nhấn, nhận dạng cử chỉ, khả năng truy cập, nhập văn bản).
   * Ghép cây widget của ứng dụng thành một cảnh.
2. Engine
   * Chịu trách nhiệm raster hóa các cảnh được hợp thành.
   * Cung cấp triển khai cấp thấp của các API cốt lõi của Flutter (ví dụ: đồ họa, bố cục văn bản, runtime Dart).
   * Công khai chức năng của nó cho framework bằng cách sử dụng API *dart:ui*.
   * Tích hợp với một nền tảng cụ thể bằng cách sử dụng API *Embedder* của Engine
3. Runner
   * Ghép các thành phần lại và nhúng thành các gói để có thể chạy trên được nền tảng đích
   * Là một phần của mẫu ứng dụng được tạo bởi *Flutter create*, thuộc sở hữu của nhà phát triển ứng dụng.

## 1.5. Ưu điểm và nhược điểm của Flutter

* **Ưu điểm**

Flutter đáp ứng các nhu cầu và yêu cầu tùy chỉnh để phát triển các ứng dụng di động. Nó cũng cung cấp nhiều lợi thế, được liệt kê dưới đây.

* Nó làm cho quá trình phát triển ứng dụng cực kỳ nhanh chóng vì tính năng tải lại nóng (Hot Reload). Tính năng này cho phép người lập trình thay đổi hoặc cập nhật code được phản ánh ngay sau khi các thay đổi được thực hiện.
* Nó cung cấp trải nghiệm cuộn mượt mà và liền mạch khi sử dụng ứng dụng mà không bị treo hoặc cắt nhiều, giúp chạy ứng dụng nhanh hơn so với các framework phát triển ứng dụng dành cho thiết bị di động khác.
* Flutter làm giảm thời gian và nỗ lực kiểm tra. Như chúng ta đã biết, các ứng dụng rung là đa nền tảng, do đó người thử nghiệm không cần phải luôn chạy cùng một nhóm thử nghiệm trên các nền tảng khác nhau cho cùng một ứng dụng.
* Nó có giao diện người dùng tuyệt vời vì nó sử dụng tiện ích tập trung vào thiết kế, các công cụ phát triển cao, API nâng cao và nhiều tính năng khác.
* Nó tương tự như một framework phản ứng trong đó các nhà phát triển không cần cập nhật nội dung giao diện người dùng theo cách thủ công.
* Nó phù hợp với các ứng dụng MVP (Sản phẩm khả thi tối thiểu) vì quá trình phát triển nhanh chóng và tính chất đa nền tảng của nó
* **Nhược điểm**

Trước đó chúng ta đã thấy rằng Flutter có nhiều ưu điểm, nhưng nó cũng chứa một số nhược điểm, được đưa ra dưới đây.

* Flutter là một ngôn ngữ tương đối mới cần được hỗ trợ tích hợp liên tục thông qua việc duy trì các tập lệnh.
* Nó cung cấp quyền truy cập rất hạn chế vào các thư viện SDK. Nó có nghĩa là một nhà phát triển không có nhiều chức năng để tạo một ứng dụng di động. Các loại chức năng như vậy cần được phát triển bởi chính nhà phát triển Flutter.
* Nó sử dụng lập trình Dart để viết code, vì vậy một nhà phát triển cần phải học các công nghệ mới. Tuy nhiên, nó rất dễ học đối với các nhà phát triển.

## 1.6. Một số thông tin liên quan khác

### 1.6.1. Các nguồn tài liệu khoá học Flutter

#### 1.6.1.1. Khoá học trả phí

Một số khoá học về Flutter trả phí nhưng đi kèm là sự đầy đủ và chất lượng như Udemy, CodeGym… Tuy mất phí nhưng sự đánh đổi lại không hề nhỏ, lợi ích mà chúng ta nhận được qua những khoá học trả phí như sau:

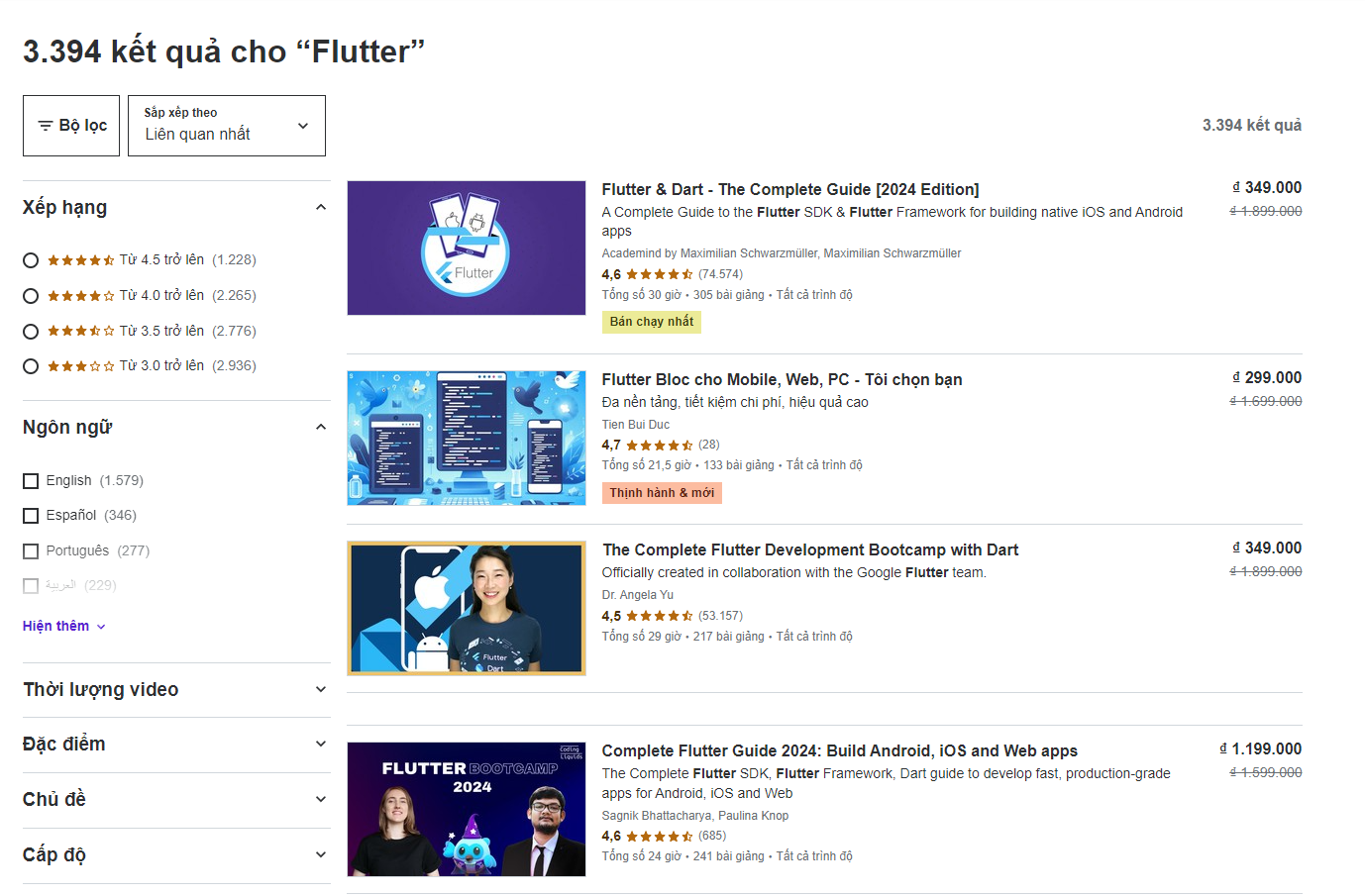
* **Nội dung chuyên sâu và cập nhật:** Khóa học trả phí thường được đầu tư kỹ lưỡng về nội dung, với các bài giảng chi tiết, cập nhật và đi sâu vào các chủ đề chuyên sâu.
* **Giảng viên uy tín:** Khóa học trả phí thường được giảng dạy bởi các giảng viên uy tín, có nhiều kinh nghiệm trong lĩnh vực giảng dạy và thực hành.
* **Hỗ trợ và tương tác:** Khóa học trả phí thường cung cấp dịch vụ hỗ trợ và tương tác tốt hơn, giúp ta giải đáp thắc mắc và trao đổi với giảng viên và học viên khác.
* **Chứng chỉ:** Một số khóa học trả phí cấp chứng chỉ sau khi hoàn thành, giúp ta nâng cao năng lực và tăng cơ hội việc làm.

Có nhiều loại khóa học trả phí khác nhau, bao gồm:

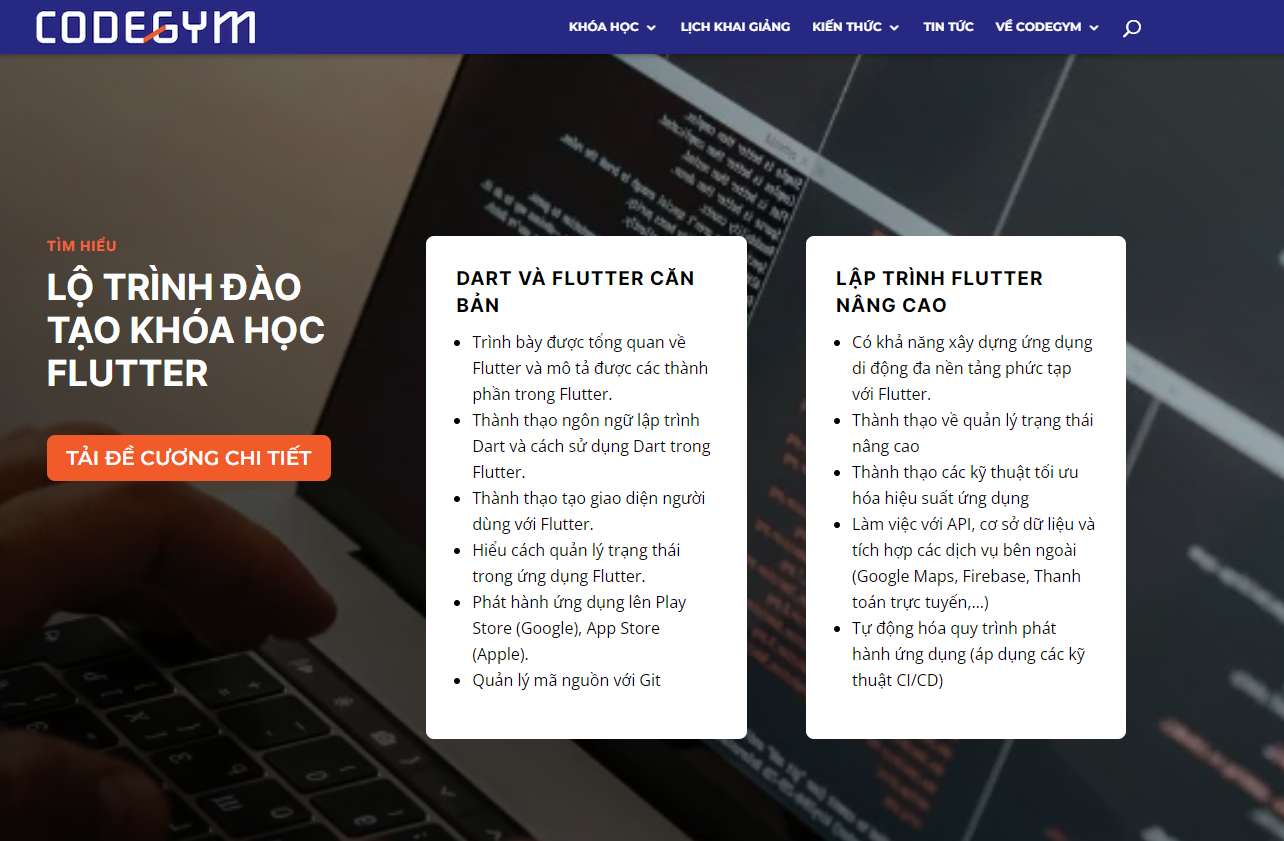
1. **Khóa học trực tuyến trên các nền tảng online:** Có thể học mọi lúc mọi nơi với các bài giảng video, bài tập thực hành và tài liệu học tập.
2. **Khóa học trực tiếp tại trung tâm:** Tham gia học tại một địa điểm cụ thể với giảng viên và các học viên khác.

Một số khoá học Flutter trả phí được nhiều người lựa chọn:

* Udemy: <https://www.udemy.com/courses/search/?src=ukw&q=Flutter>
* CodeGym: [https://codegym.vn/khoa-hoc/Flutter/](https://codegym.vn/khoa-hoc/flutter/)



Hình 1. 6. Một số khoá học Flutter trả phí trên Udemy



Hình 1. 7. Khoá học Flutter trả phí trên CodeGym

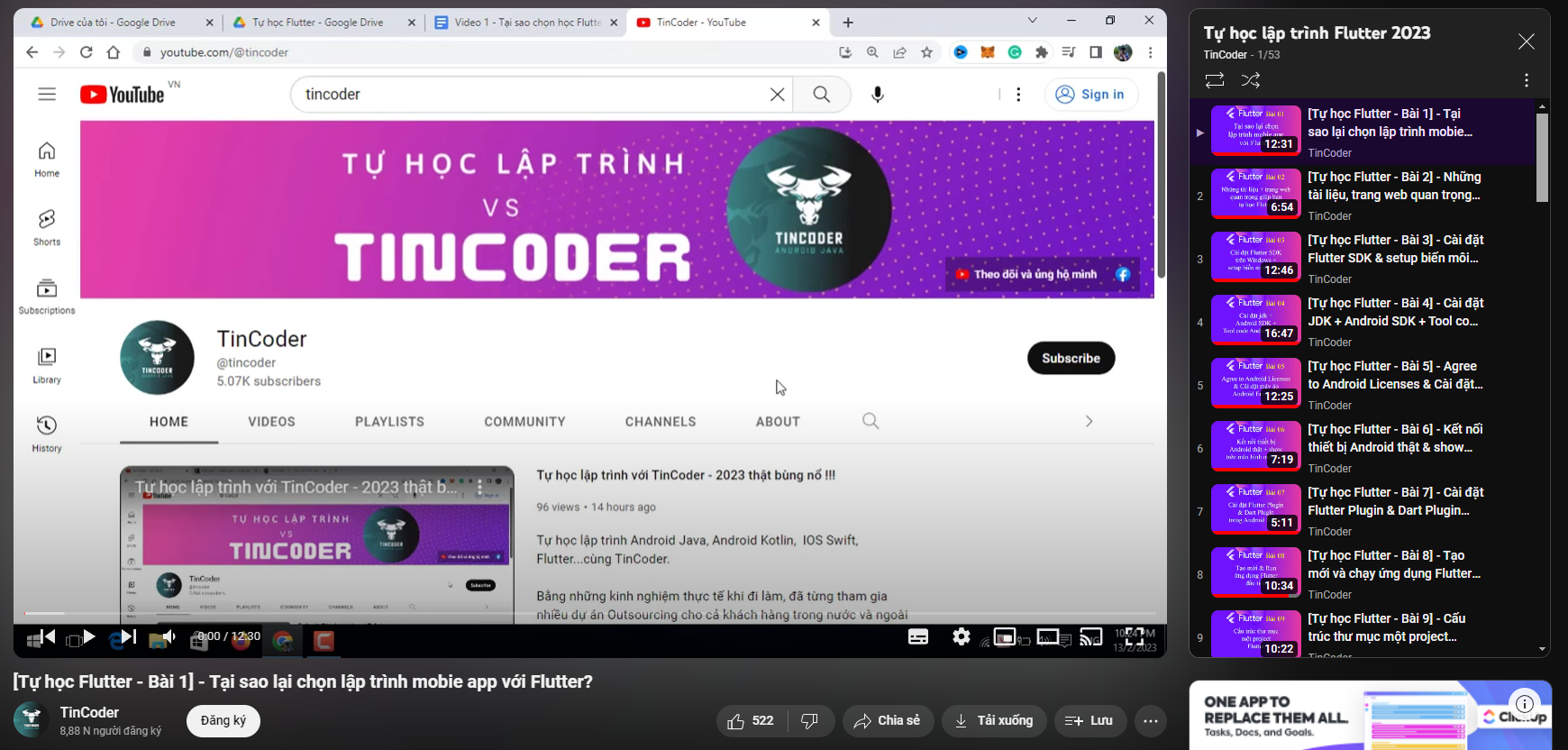
#### 1.6.1.2. Khoá học miễn phí

Ngoài việc phải trả phí để học lập trình ra thì ta cũng có sự lựa chọn khác là lựa chọn các khoá học lập trình Flutter miễn phí trên các nền tảng khác, ví dụ: Youtube, Các trang web phi lợi nhuận. Ưu điểm của những khoá học này là:

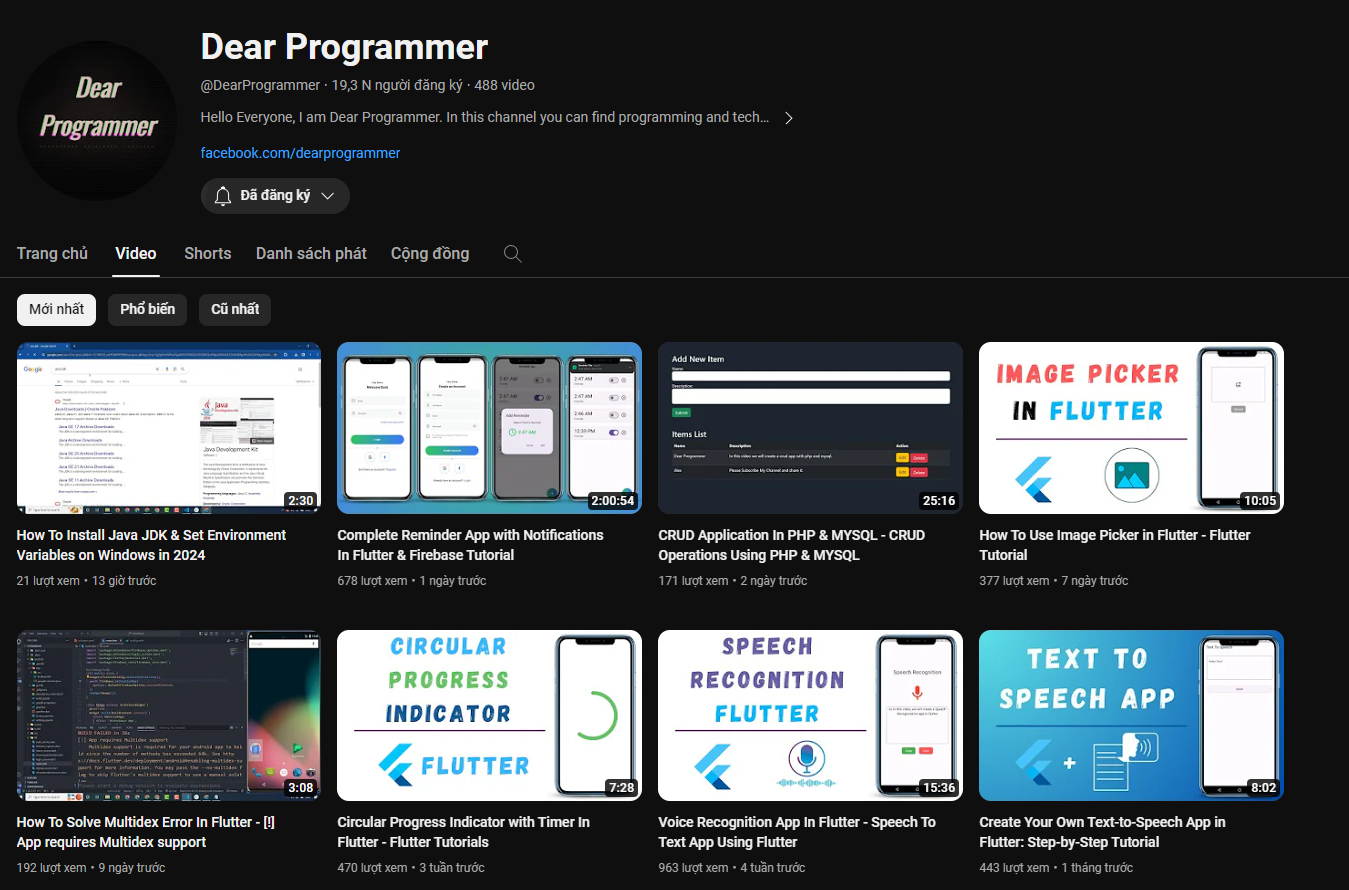
* **Tiếp cận kiến thức dễ dàng:** Có thể học tập mọi lúc mọi nơi mà không cần lo lắng về chi phí.
* **Thử nghiệm lĩnh vực mới:** Có thể khám phá các lĩnh vực mới mà không cần đầu tư nhiều tiền.
* **Phát triển kỹ năng:** Có thể trau dồi kỹ năng và kiến thức mới để nâng cao bản thân.

Một số khoá học Flutter miễn phí:

* TinCoder: <https://www.youtube.com/@tincoder>
* DearProgrammer: <https://www.youtube.com/@DearProgrammer>
* CafeDev: [https://cafedev.vn/series-tu-hoc-Flutter-tu-co-ban-toi-nang-cao](https://cafedev.vn/series-tu-hoc-flutter-tu-co-ban-toi-nang-cao)



Hình 1. 8. Khoá học lập trình Flutter miễn phí trên Youtube: TinCoder



Hình 1. 9. Khoá học lập trình Flutter miễn phí trên Youtube: Dear Programmer

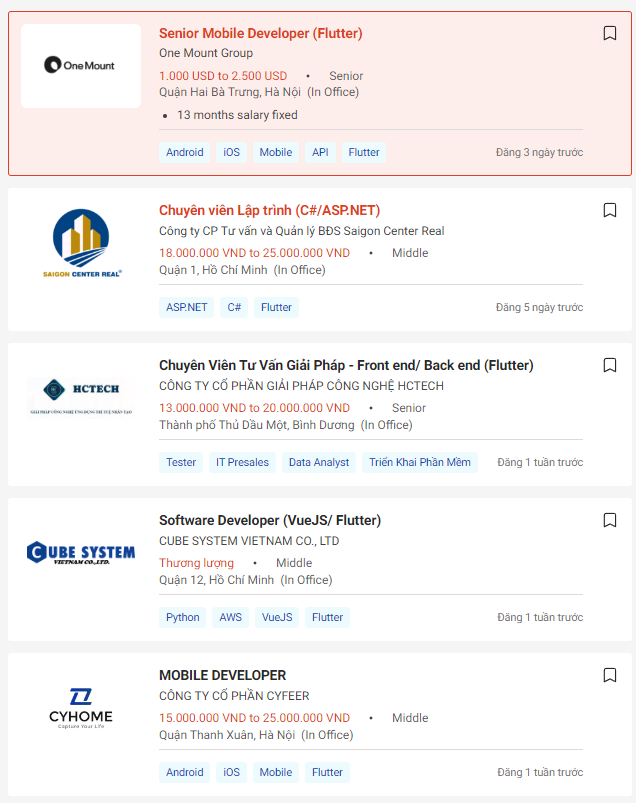


Hình 1. 10. Khoá học lập trình Flutter miễn phí trên web: CafeDev

### 1.6.2. Các thông tin tuyển dụng về Flutter

Một số mức lương mà các nhà tuyển dụng đưa ra để tuyển dụng các lập trình viên Flutter trên các nền tảng như sau:

* **Topdev.vn**

****

Hình 1. 11. Một số tuyển dụng lập trình viên Flutter trên trang Topdev.vn

* **Topcv.vn**

**A screenshot of a chat

Description automatically generated**

Hình 1. 12. Một số tuyển dụng lập trình viên Flutter trên trang topcv.vn

Nhìn chung, mức lương của các lập trình viên Flutter tương đối cao và nhu cầu tuyển dụng lập trình viên tăng cao. Mức lương dao dộng từ 15 – 40 triệu tuỳ vào kinh nghiệm và năng lực, ta có một bảng lương trung bình từng Level như sau:

|  |  |
| --- | --- |
| **Level (Kinh nghiệm)** | **Mức lương trung bình (VND)** |
| Junior Developer (0-1 năm kinh nghiệm) | 10.000.000 - 15.000.000 |
| Mid-level Developer (1-3 năm kinh nghiệm) | 15.000.000 - 25.000.000 |
| Senior Developer (3-5 năm kinh nghiệm) | 25.000.000 - 35.000.000 |
| Team Lead/Architect (5+ năm kinh nghiệm) | 35.000.000 - 50.000.000 |

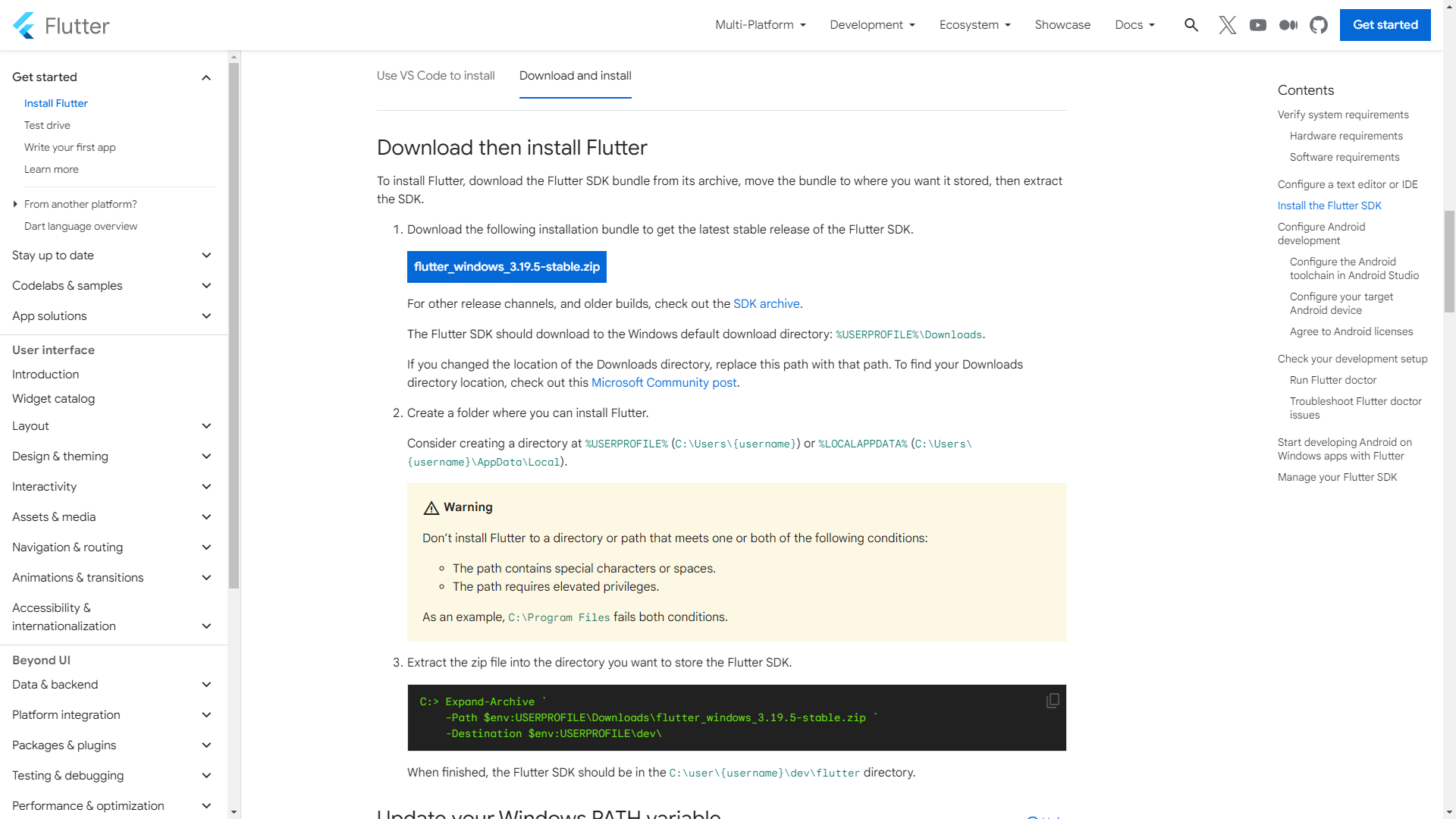
## 1.7. Cài đặt và cấu hình cần thiết cho việc lập trình với Flutter

### 1.7.1. Flutter SDK

Trước tiên ta cần cài đặt Flutter SDK. Vậy Flutter SDK là gì ? Flutter SDK là bộ công cụ phát triển phần mềm (SDK) mã nguồn mở, miễn phí được tạo bởi Google để xây dựng các ứng dụng di động đa nền tảng chất lượng cao với hiệu suất cao và giao diện người dùng đẹp mắt.

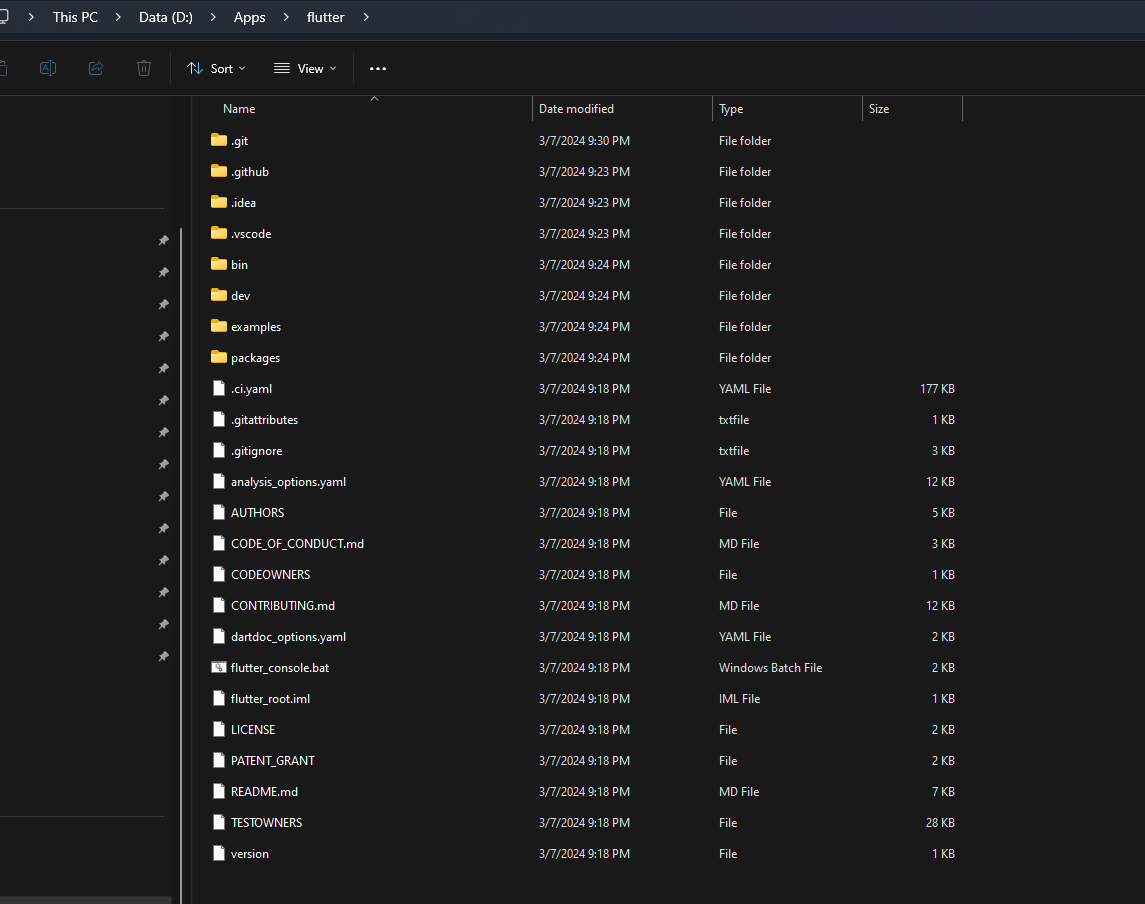
Các bước cài đặt được mô tả như sau (Lưu ý: Các bước hướng dẫn chỉ được áp dụng trên máy tinh Windows, không bao gồm MacOS và Linux):

**Bước 1: Truy cập vào trang web docs.Flutter.dev để tải Flutter SDK**



Hình 1. 13. Tải gói Flutter SDK mới nhất từ trang chủ chính thức của Flutter

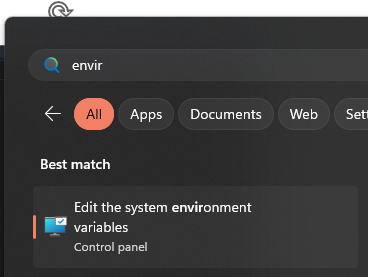
**Bước 2: Sau khi tải xong, giải nén và đặt nó ở một thư mục đường dẫn bất kỳ mà ta mong muốn** (Lưu ý: Không nên đặt Flutter SDK ở nơi cần sự cho phép (cấp quyền permission từ người dùng))



Hình 1. 14. Giải nén và cài đặt gói Flutter SDK tại một đường dẫn bất kỳ

**Bước 3: Cấu hình biến môi trường (Enviroment Variables)**

* Ta truy cập Enviorment Variables từ thanh tìm kiếm Windows



Hình 1. 15. Truy cập Enviroment Variables từ thanh tìm kiếm

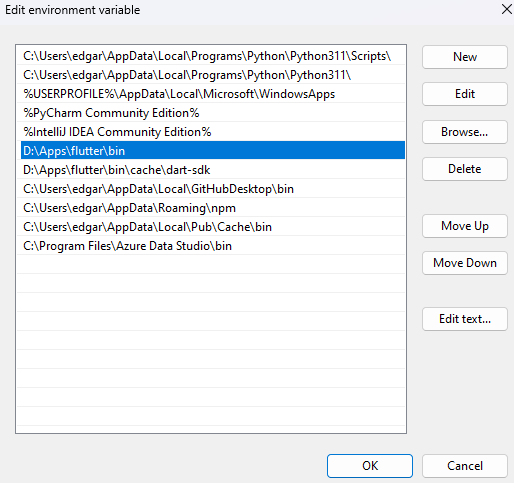
* Chọn vào Enviroment Variables -> ấn vào Nút Edit của User Variables

A screenshot of a computer

Description automatically generated

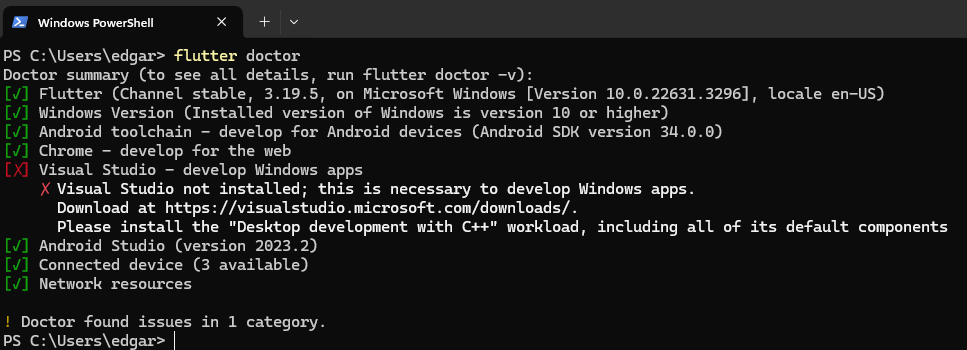
Hình 1. 16. Chọn Edit Path của User Variables

* Thiết lập đường dẫn Flutter vào thư mục bin của Flutter SDK như minh hoạ



Hình 1. 17. Thiết lập đường dẫn tới thư mục bin của Flutter SDK

* Kiểm tra tất cả trạng thái trước bằng cách chạy lên **Flutter doctor** trên CMD hoặc Windows Terminal để đảm bảo rằng tất cả đã được cài đặt thành công và có thể thực hiện lập trình Flutter (Lưu ý: Chỉ cần tích xanh đầy đủ như ở hình dưới là đã có thể lập trình Flutter rồi)

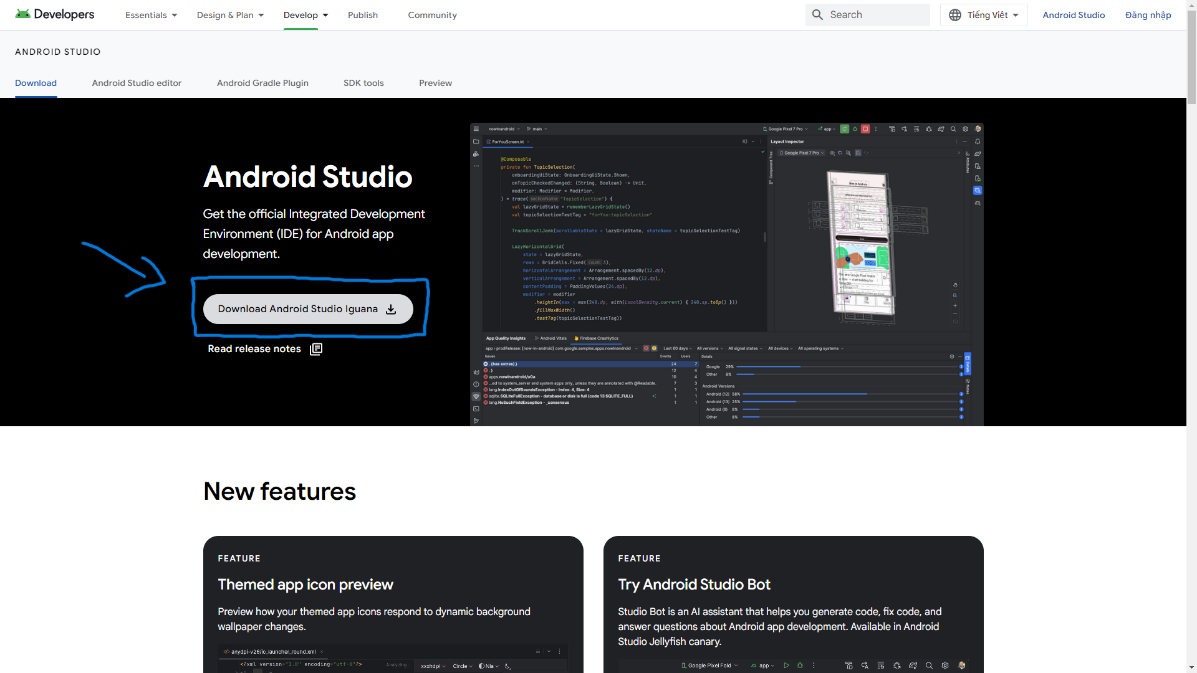


Hình 1. 18. Kiểm tra bằng lệnh Flutter doctor

### 1.7.2. IDE

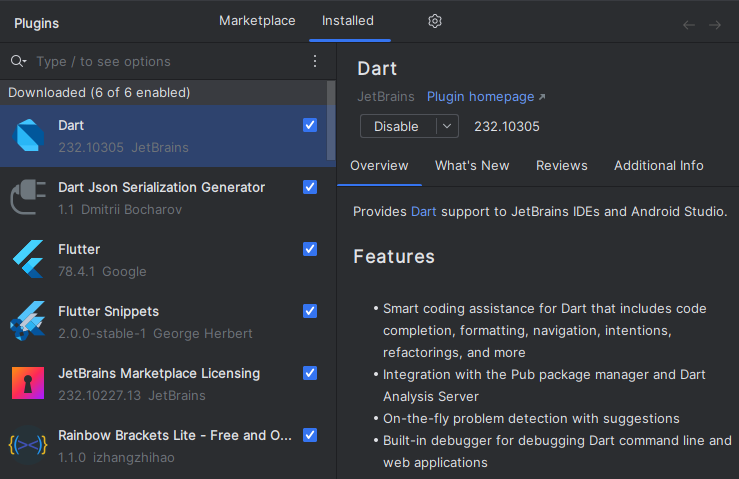
Việc lập trình thì không thể thiếu IDE hoặc Code Editor. Trong bài báo cáo này, chúng em chỉ sử dụng **Android Studio** để lập trình Flutter. Sau đây là các bước hướng dẫn cho việc thiết lập **Android Studio** để có thể lập trình được Flutter:

**Bước 1: Truy cập vào trang chủ developer.android.com để cài đặt Android Studio**



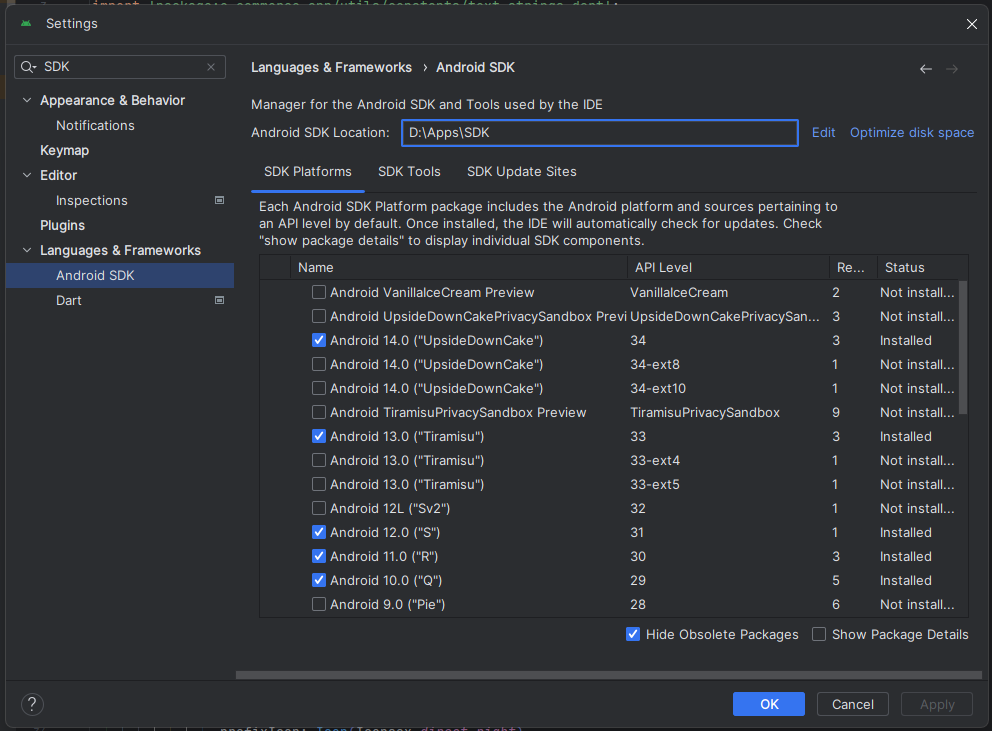
Hình 1. 19. Truy cập vào trang chủ android studio để cài đặt IDE

**Bước 2: Sau khi cài đặt IDE xong, ta tiến hành cài đặt các plugin cần thiết như: Dart, Flutter…..**

****

Hình 1. 20. Cài đặt các Plugin cần thiết trên Android Studio

**Bước 3: Kiểm tra Android SDK và Dart SDK đã được thiết lập trên Android Studio hay chưa. Setting -> Android SDK, Setting -> Dart**

****

Hình 1. 21. Kiểm tra Android SDK

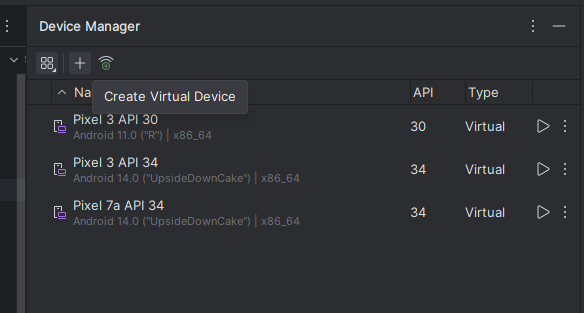
A screenshot of a computer

Description automatically generated

Hình 1. 22. Kiểm tra Dart SDK

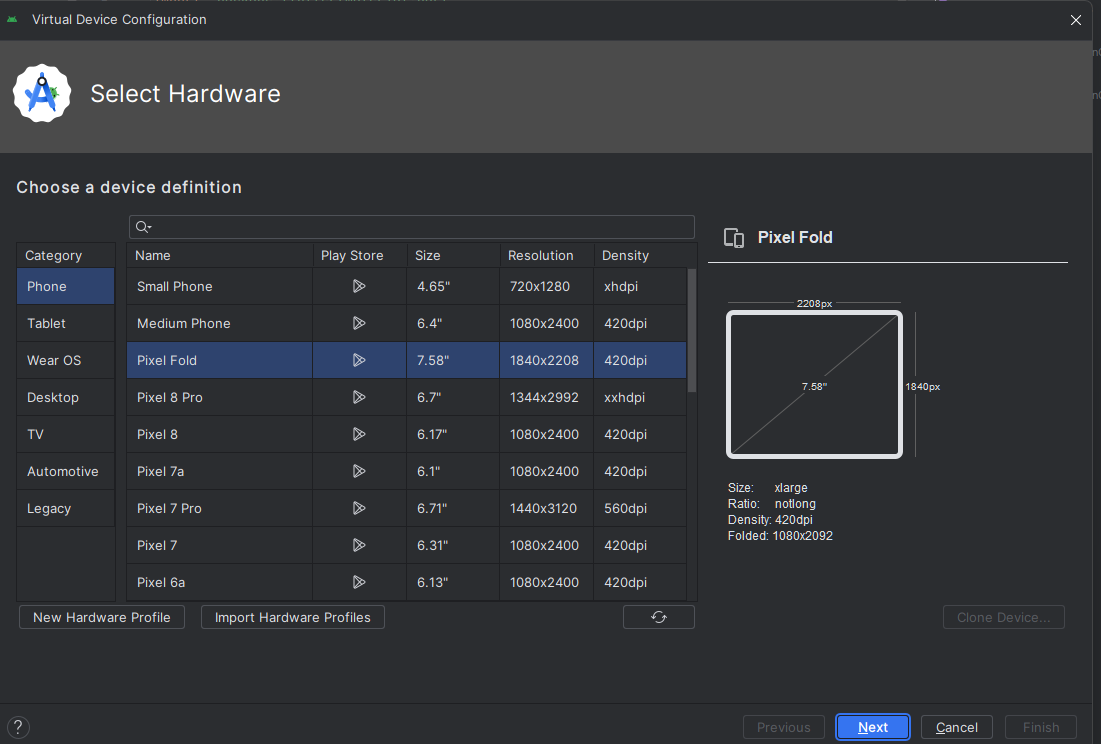
**Bước 4: Tạo thử máy ảo Emulator trên Android Studio**

* Ta tạo một máy ảo bằng cách ấn vào nút *Create Virtual Device*

****

Hình 1. 23. Ấn vào nút Create Virtual Device để tạo máy ảo

* Tiếp sau đó ta chọn kích thước màn hình, phiên bản android của máy ảo, các tuỳ biến khác và ấn Finish để hoàn thành



Hình 1. 24. Tuỳ chọn kích cỡ màn hình máy ảo

A screenshot of a computer

Description automatically generated

Hình 1. 25. Tuỳ chọn phiên bản Android

A screenshot of a computer

Description automatically generated

Hình 1. 26. Các tuỳ chọn còn lại và ấn Finish

### 1.7.3. Khởi chạy chương trình đầu tiên

Sau khi hoàn tất quá trình thiết lập môi trường và IDE cho Flutter. Tiếp sau đó ta tiến hành khởi tạo thử chương trình đầu tiên để kiểm tra xem mọi thứ có hoạt động ổn định hay không. Các bước thực hiện được mô tả như sau:

**Bước 1: Ấn vào nút tạo mới Project và chọn vào mục Flutter. Sau đó là bấm next**

**A screenshot of a computer

Description automatically generated**

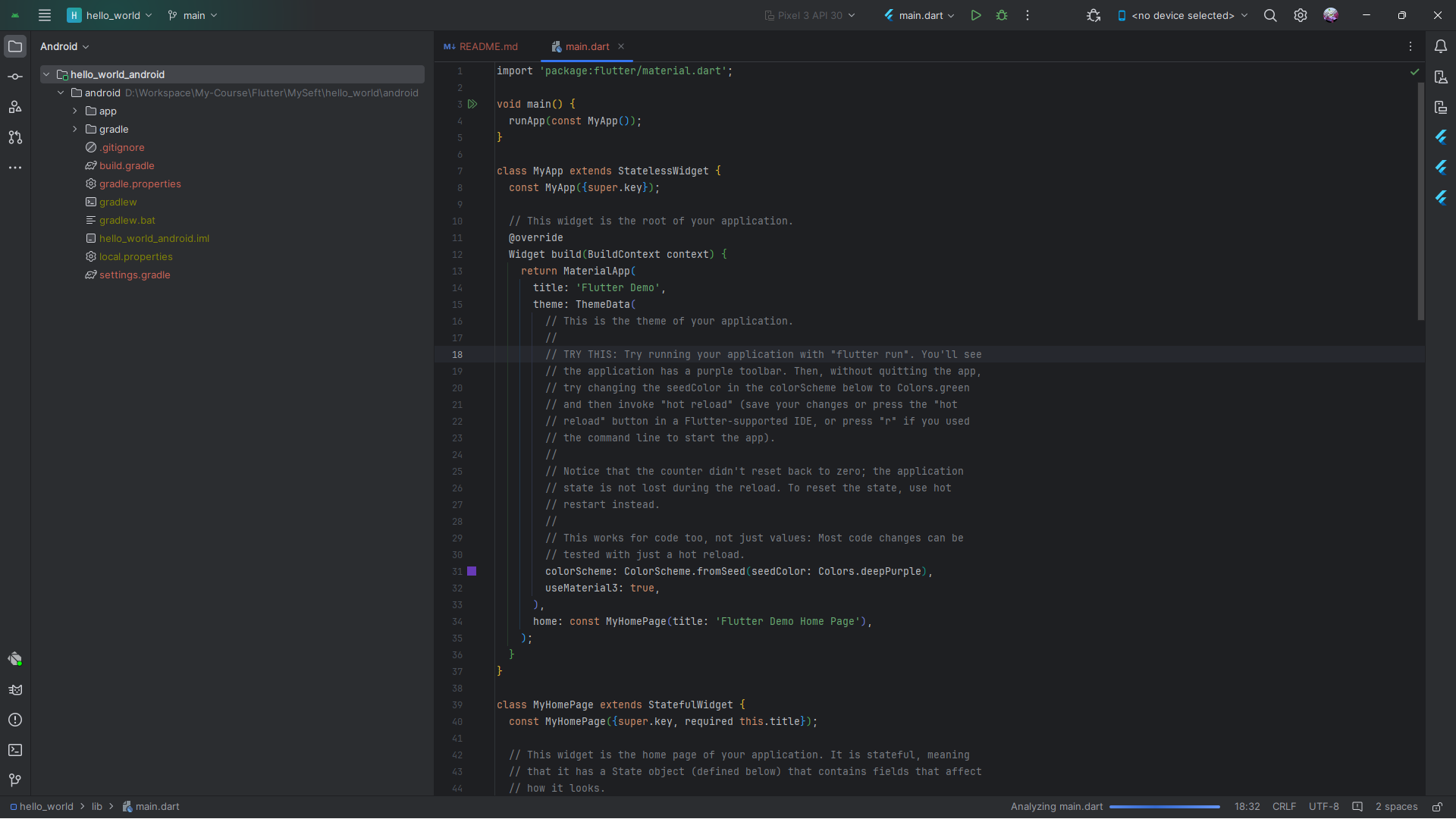
Hình 1. 27. Tạo mới một dự án Flutter

**Bước 2: Thiết lập tên dự án, đường dẫn, mô tả loại dự án và các thiết lập khác**

****

Hình 1. 28. Thiết lập tên dự án, đường dẫn, mô tả, loại dự án và các thiết lập khác cho dự án Flutter đầu tiên

**Bước 3: Sau khi khởi tạo dự án thành công, ta truy cập vào dự án và thử chạy dự án đó trên máy ảo mà chúng ta đã tạo trước đó**



Hình 1. 29. Những dòng code mặc định của một dự án Flutter

A screenshot of a computer

Description automatically generated

Hình 1. 30. Khởi tạo máy ảo cho ứng dụng Flutter đầu tiên

Để có thể chạy một ứng dụng Flutter thì điều tiên quyết phải có đó là các dòng lệnh như sau:

|  |
| --- |
| import 'package:Flutter/material.dart';  void main() {  runApp(const MyApp());  } |

Giải thích từng dòng lệnh như sau:

* Dòng lệnh **import 'package:Flutter/material.dart'** giúp import thư viện Material vào ứng dụng Flutter. Thư viện Material cung cấp các widget cơ bản để xây dựng giao diện người dùng cho ứng dụng Flutter, bao gồm:
  + Nút bấm
  + Text
  + Hộp thoại
  + Thanh cuộn
  + V.v.
* Hàm **main()** là điểm khởi đầu của mọi ứng dụng Flutter. Hàm này thực hiện hai nhiệm vụ chính:
  + Khởi tạo widget gốc của ứng dụng.
  + Gọi hàm **runApp()** để hiển thị widget gốc trên màn hình.
* Widget **MyApp()** là widget gốc của ứng dụng Flutter. Widget này chịu trách nhiệm hiển thị giao diện người dùng chính của ứng dụng.
* **Lý do cần thiết:**
* Các dòng lệnh này cung cấp cho ứng dụng Flutter các thành phần và chức năng cần thiết để hoạt động.
* Dòng lệnh import thư viện Material cung cấp các widget cơ bản để xây dựng giao diện người dùng.
* Hàm *main()* là điểm khởi đầu của ứng dụng và chịu trách nhiệm hiển thị giao diện người dùng.
* Widget *MyApp()* là widget gốc của ứng dụng và hiển thị giao diện người dùng chính.
* **Do đó, nếu không có các dòng lệnh này, ứng dụng Flutter sẽ không thể chạy**

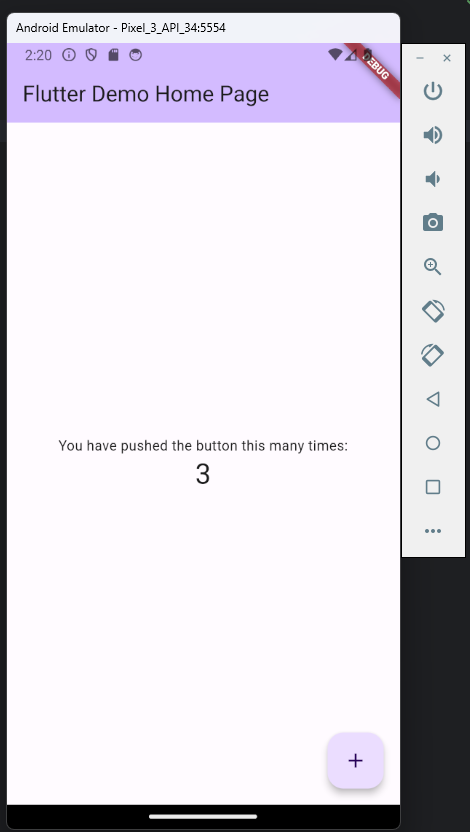
Để khởi chạy được các dòng lệnh dart trên Android Studio để chúng thực thi lên máy ảo thì ta cần ấn tổ hợp phím **Shift + F10** hoặc ấn vào nút  trên góc phải màn hình. Tiếp sau đó, Android Studio sẽ tự động thực thi biên dịch code và tiến hành cài đặt ứng dụng lên máy ảo:

A screenshot of a computer

Description automatically generated

Hình 1. 31. Android Studio biên dịch code và cài đặt ứng dụng Flutter lên máy ảo

Sau khi quá trình biên dịch hoàn tất, máy ảo sẽ khởi động ứng dụng:



Hình 1. 32. Máy ảo khởi động chương trình Flutter đầu tiên

Sau quá trình đó, Android Studio sẽ đặt máy ảo và chương trình trong trạng thái Hot Reload, mỗi khi ta thay đổi dòng code và ấn Save, thì giao diện chương trình cũng sẽ tự động load lại theo:

A close up of a button

Description automatically generated

Hình 1. 33. Android Studio đặt chương trình đang chạy vào trạng thái Hot Reload

# CHƯƠNG 2. CHI TIẾT VỀ FLUTTER

## 2.1. Những kiến thức cơ bản về Flutter

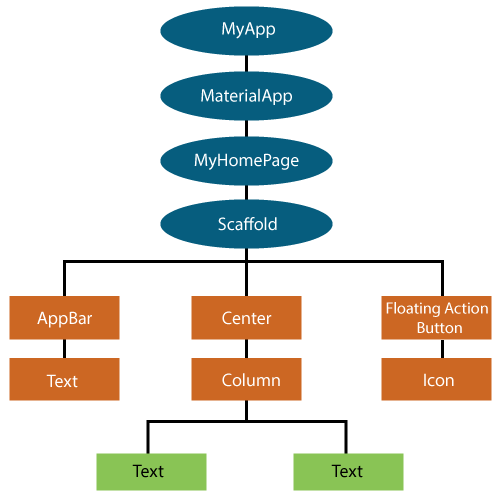
### 2.1.1. Tổng quan về Widgets và một số Widgets cần phải biết

#### 2.1.1.1. Khái niệm về Widget trong Flutter

Bất cứ khi nào ta viết code để xây dựng bất cứ thứ gì trong Flutter, thì cái mà ta xây dựng được sẽ nằm trong một *Widgets*. Mục đích chính là xây dựng ứng dụng từ các widget. Nó mô tả chế độ xem ứng dụng của ta trông như thế nào với cấu hình và trạng thái hiện tại của chúng. Khi ta thực hiện bất kỳ thay đổi nào trong code, widget con sẽ xây dựng lại mô tả của nó bằng cách tính toán sự khác biệt của widget con hiện tại và trước đó để xác định những thay đổi tối thiểu đối với việc hiển thị trong giao diện người dùng của ứng dụng.

Các widget được lồng vào nhau để xây dựng ứng dụng. Nó có nghĩa là thư mục gốc của ứng dụng tự nó là một widget, và tất cả các cách nhìn xuống cũng là một widget. Ví dụ: một widget có thể hiển thị một thứ gì đó, có thể xác định thiết kế, có thể xử lý tương tác, v.v.

Hình ảnh dưới dây mô tả trực quan đơn giản của một cây Widgets:



Hình 2. 1. Một ví dụ về cây Widgets trong Flutter

Chúng ta có thể tạo một Widget trong Flutter như sau:

|  |
| --- |
| Class ImageWidget extends StatelessWidget {  // Class Body  } |

Một ví dụ khác, ví dụ về Hello World:

|  |
| --- |
| import 'package:flutter/material.dart';  class MyHomePage extends StatelessWidget {  MyHomePage({Key key, this.title}) : super(key: key);  // This widget is the home page of your application.  final String title;    @override  Widget build(BuildContext context) {  return Scaffold(  appBar: AppBar(  title: Text(this.title),  ),  body: Center(  child: Text('Hello World),  ),  );  }  } |

#### 2.1.1.2. Các loại Widget con

Ngoài ra, trong Flutter chúng ta có thể chia widget Flutter thành 2 dạng:

1. **Hiển thị** (Visiable) (Đầu ra, đầu vào)
2. **Vô hình** (Invisiable) (Bố cục và kiểm soát – Layout and Control)

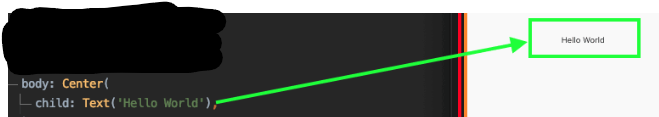
**Dạng 1: Một số Widget hiển thị cơ bản có sử dụng đầu ra, đầu vào cho dữ liệu bao gồm:**

* **Text**

Widget Text giữ một số văn bản để hiển thị trên màn hình. Chúng ta có thể căn chỉnh widget văn bản bằng cách sử dụng thuộc tính *textAlign* và thuộc tính style cho phép tùy chỉnh Text bao gồm phông chữ, độ đậm của phông chữ, kiểu phông chữ, khoảng cách giữa các chữ cái, màu sắc và nhiều hơn nữa. Chúng ta có thể sử dụng nó như các đoạn mã dưới đây.

|  |
| --- |
| // Ví dụ về widget Text  Text(  'Hello World!',  textAlign: TextAlign.center,  style: new TextStyle(fontWeight: FontWeight.bold),  ) |

Hình ảnh minh hoạ:



Hình 2. 2. Hình ảnh minh hoạ cho Widget Text

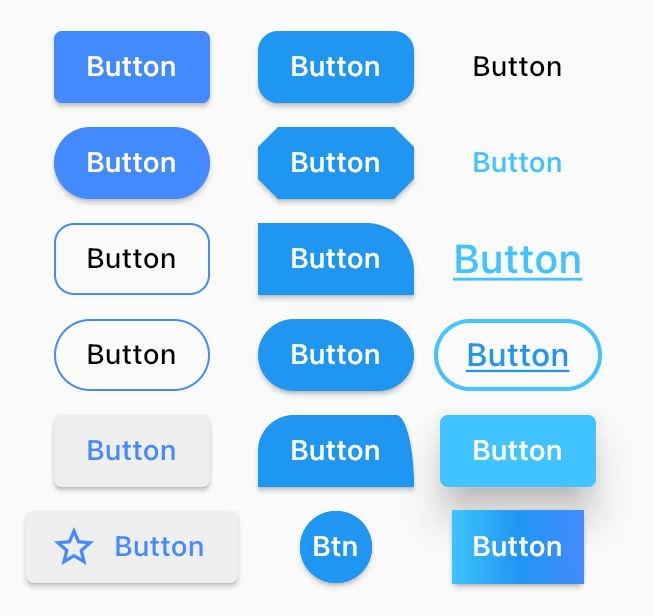
* **Button**

Widget này cho phép ta thực hiện một số hành động khi nhấp chuột. Flutter có sẵn các phiên bản Button khác nhau tuỳ thuộc vào mục đích sử dụng của chúng ta đối với giao diện mà chúng ta đang xây dựng, Ví dụ như: TextButton, ElevatedButton, OutlinedButton và IconButton. Chúng ta có thể sử dụng nó như các đoạn mã dưới đây.

|  |
| --- |
| // Một số loại widget Button thường được sử dụng  TextButton(  onPressed: () {  // Xử lý khi nút được bấm  },  child: Text('Nút TextButton'),  )  ElevatedButton(  onPressed: () {  // Xử lý khi nút được bấm  },  child: Text('Nút ElevatedButton'),  )  OutlinedButton(  onPressed: () {  // Xử lý khi nút được bấm  },  child: Text('Nút OutlinedButton'),  )  IconButton(  onPressed: () {  // Xử lý khi nút được bấm  },  icon: Icon(Icons.add),  ) |

Trong đoạn code ví dụ trên, các widget Button luôn luôn có thuộc tính *onPressed()* để thực hiện hành động sau khi ta ấn vào Button và một thuộc tính child hoặc thuộc tính tương ứng với loại Button để chứa nội dung hiển thị của Button đó.

Hình ảnh minh hoạ:



Hình 2. 3. Hình ảnh minh hoạ cho một số Widget Button

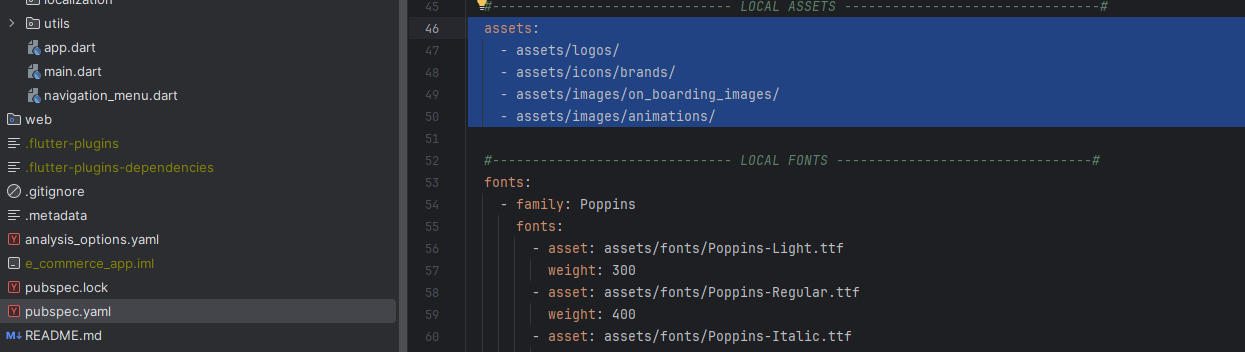
* **Image**

Widget này giữ hình ảnh có thể tìm nạp hình ảnh từ nhiều nguồn như từ thư mục nội dung hoặc trực tiếp từ URL. Nó cung cấp nhiều hàm tạo để tải hình ảnh, được đưa ra dưới đây:

* Hình ảnh(Image): Đây là một trình tải hình ảnh chung, được sử dụng bởi *ImageProvider* .
* Tài sản (asset): Nó tải hình ảnh từ thư mục tài sản dự án của ta.
* tệp(file): Nó tải hình ảnh từ thư mục hệ thống.
* bộ nhớ(memory): Nó tải hình ảnh từ bộ nhớ.
* mạng(network): Nó tải hình ảnh từ mạng.

Để thêm hình ảnh vào dự án, trước tiên ta cần tạo một thư mục nội dung nơi ta lưu giữ hình ảnh của mình và sau đó thêm dòng bên dưới vào tệp *pubspec.yaml* như sau:

|  |
| --- |
| assets:  - <Đường dẫn tới thư mục chứa ảnh> |



Hình 2. 4. Đường dẫn hình ảnh cấu hình trong pubspec.yaml trong dự án thực tế

Mã nguồn hoàn chỉnh để thêm hình ảnh được hiển thị bên dưới trong ví dụ hello world:

|  |
| --- |
| class MyHomePage extends StatelessWidget {  MyHomePage({Key key, this.title}) : super(key: key);  // This widget is the home page of your application.  final String title;    @override  Widget build(BuildContext context) {  return Scaffold(  appBar: AppBar(  title: Text(this.title),  ),  body: Center(  child: Image.asset('assets/computer.png'),  ),  );  }  } |

Khi ta chạy ứng dụng, ta sẽ nhận được kết quả như sau:

A computer with a blue screen

Description automatically generated

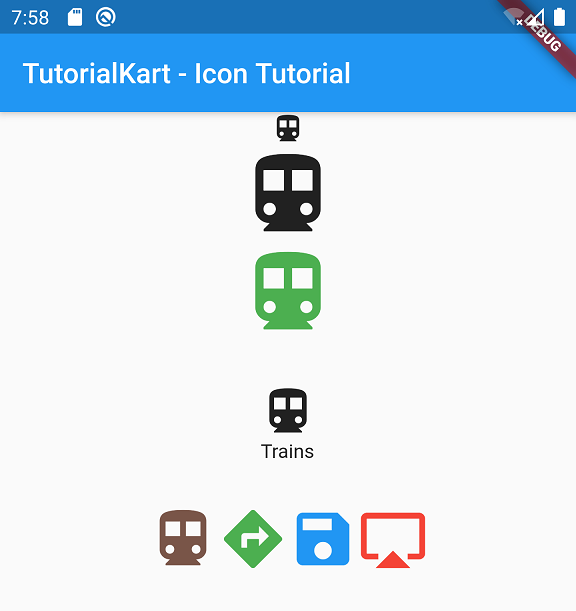
Hình 2. 5. Hình ảnh minh hoạ cho Widget Image

* **Image**

Widget này hoạt động như một thùng chứa để lưu trữ Biểu tượng trong Flutter. Đoạn mã sau đây giải thích rõ ràng hơn.

|  |
| --- |
| Icon(  Icons.add,  size: 34.0,  ) |

Hình ảnh minh hoạ:



Hình 2. 6. Hình ảnh minh hoạ cho Widget Icon

**Dạng 2: Một số Widget dạng ẩn, liên quan đến việc bố trí và hiển thị các dạng bố cục lên màn hình:**

* **Column**

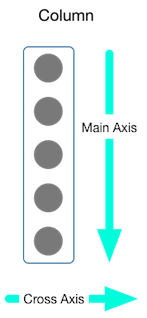
Widget Column là một widget layout cơ bản trong Flutter, dùng để sắp xếp các widget con theo chiều dọc, từ trên xuống dưới. Nó đóng vai trò quan trọng trong việc xây dựng giao diện người dùng (UI) cho các ứng dụng Flutter. Ta có đoạn code ví dụ như sau đây:

|  |
| --- |
| Column(  mainAxisAlignment: MainAxisAlignment.center,  crosssAxisAlignment: CrossAxisAlignment.center,  mainAxisSize: MainAxisSize.minContent,  children: [  Text("Item1"),   Text("Item2"),  Text("Item3"),  ],  ), |

Các thuộc tính quan trọng của Column như sau:

* **children**: Danh sách các widget con được sắp xếp theo chiều dọc.
* **mainAxisAlignment**: Xác định cách sắp xếp các widget con theo chiều dọc. Các giá trị phổ biến bao gồm:
  + **start**: Sắp xếp các widget con ở đầu (top) của Column.
  + **center**: Sắp xếp các widget con ở giữa (center) của Column.
  + **end**: Sắp xếp các widget con ở cuối (bottom) của Column.
  + **spaceBetween**: Tạo khoảng cách đều giữa các widget con.
  + **spaceAround**: Tạo khoảng cách bằng nhau giữa các widget con và hai đầu của Column
* **crossAxisAlignment**: Xác định cách sắp xếp các widget con theo chiều ngang. Các giá trị phổ biến bao gồm:
  + **start**: Sắp xếp các widget con ở cạnh trái (start) của Column.
  + **center**: Sắp xếp các widget con ở giữa (center) của Column.
  + **end**: Sắp xếp các widget con ở cạnh phải (end) của Column.
  + **stretch**: Mở rộng các widget con để lấp đầy chiều ngang của Column.
* **mainAxisSize**: Xác định kích thước tối đa của Column theo chiều dọc. Các giá trị phổ biến bao gồm:
  + **MainAxisSize.minContent**: Kích thước tối thiểu cần thiết để chứa các widget con.
  + **MainAxisSize.maxContent**: Mở rộng để chứa tất cả các widget con.
  + **MainAxisSize.fill**: Mở rộng để lấp đầy chiều cao có sẵn.

Hình ảnh minh hoạ:



Hình 2. 7. Hình ảnh minh hoạ cho Widget Column

* **Row**

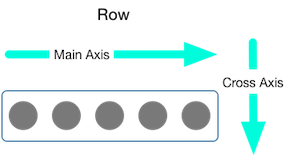
Widget Row là một widget layout cơ bản trong Flutter, dùng để sắp xếp các widget con theo chiều ngang, từ trái sang phải. Nó đóng vai trò quan trọng trong việc xây dựng giao diện người dùng (UI) cho các ứng dụng Flutter. Ta có đoạn code ví dụ như sau đây:

|  |
| --- |
| Row (  mainAxisAlignment: MainAxisAlignment.center,  crosssAxisAlignment: CrossAxisAlignment.center,  mainAxisSize: MainAxisSize.minContent,  children: [  Text("Item1"),   Text("Item2"),  Text("Item3"),  ],  ), |

Các thuộc tính quan trọng của Row như sau:

* **children**: Danh sách các widget con được sắp xếp theo chiều ngang.
* **mainAxisAlignment**: Xác định cách sắp xếp các widget con theo chiều ngang. Các giá trị phổ biến bao gồm:
  + **start**: Sắp xếp các widget con ở đầu (top) của Row.
  + **center**: Sắp xếp các widget con ở giữa (center) của Row.
  + **end**: Sắp xếp các widget con ở cuối (bottom) của Row.
  + **spaceBetween**: Tạo khoảng cách đều giữa các widget con.
  + **spaceAround**: Tạo khoảng cách bằng nhau giữa các widget con và hai đầu của Row
* **crossAxisAlignment**: Xác định cách sắp xếp các widget con theo chiều dọc. Các giá trị phổ biến bao gồm:
  + **start**: Sắp xếp các widget con ở cạnh trái (start) của Row.
  + **center**: Sắp xếp các widget con ở giữa (center) của Row.
  + **end**: Sắp xếp các widget con ở cạnh phải (end) của Row.
  + **stretch**: Mở rộng các widget con để lấp đầy chiều ngang của Row.
* **mainAxisSize**: Xác định kích thước tối đa của Row theo chiều dọc. Các giá trị phổ biến bao gồm:
  + **MainAxisSize.minContent**: Kích thước tối thiểu cần thiết để chứa các widget con.
  + **MainAxisSize.maxContent**: Mở rộng để chứa tất cả các widget con.
  + **MainAxisSize.fill**: Mở rộng để lấp đầy chiều cao có sẵn.

Hình ảnh minh hoạ:



Hình 2. 8. Hình ảnh minh hoạ cho Widget Row

Ngoài ra, ta còn có các Widget thường được sử dụng cho việc căn chỉnh, đệm, căn lề cho các Widget con như: **Padding, Margin, Center…** Ta có các dòng code ví dụ như sau:

|  |
| --- |
| // Một số cách để sử dụng Widget dạng căn chỉnh  // Cách 1: Sử dụng bên trong 1 Widget cha  Center(  child: Container(  padding: EdgeInsets.all(10),  margin: EdgeInsets.all(10),  child: Text('Hello World'),  ),  );  // Cách 2: Sử dụng như một Widget thông thường  // Có thể áp dụng cho cả Margin và Center  Padding(  padding: EdgeInsets.all(10),  child: Text('Hello World'),  ) |

Bên cạnh các Widget vừa được giới thiệu trên thì Flutter còn có rất nhiều dạng Widget khác liên quan tới việc căn chỉnh các Widget con như **Align, SizedBox, Stack, CustomPaint….** Do bài báo cáo này có giới hạn nên không thể trình bày chi tiết hết tất cả, có thể tìm hiểu thêm tại <https://docs.flutter.dev/>.

#### 2.1.1.3. Phân loại Widget State

State là một phần cốt lõi của Flutter, đóng vai trò quan trọng trong việc tạo ra các giao diện người dùng (UI) động và phản hồi. State đại diện cho dữ liệu mà widget hiển thị và thay đổi theo thời gian. Việc quản lý state hiệu quả là chìa khóa để xây dựng các ứng dụng Flutter linh hoạt và mượt mà.

Hai phương pháp chính để quản lý state trong Flutter:

1. StatelessWidget
2. StatefulWidget

* **StatefulWidget**

StatefulWidget có thông tin state. Nó chủ yếu chứa hai lớp: **state object** và **widget**. Nó là động vì nó có thể thay đổi dữ liệu bên trong trong suốt thời gian tồn tại của widget. widget con này không có phương thức **build()** . Nó có phương thức **createState()** , trả về một lớp mở rộng Lớp **state** của Flutters. Các ví dụ của StatefulWidget là *Checkbox*, *Radio*, *Slider*, *InkWell*, *Form* và *TextField*.

StatefulWidget thường được sử dụng cho các widget có state thay đổi theo thời gian, cần cập nhật giao diện khi state thay đổi. Nó đồng thời cung cấp phương thức **setState()** để cập nhật state của widget, dẫn đến việc xây dựng lại widget và hiển thị thay đổi trên giao diện

|  |
| --- |
| // Một ví dụ cho StatefulWidget  class CounterWidget extends StatefulWidget {  @override  \_CounterWidgetState createState() => \_CounterWidgetState();  }  class \_CounterWidgetState extends State<CounterWidget> {  int \_counter = 0;  void incrementCounter() {  setState(() {  \_counter++;  });  }  @override  Widget build(BuildContext context) {  return Column(  children: [  Text('Số lượng: $\_counter'),  ElevatedButton(  onPressed: incrementCounter,  child: Text('Tăng'),  ),  ],  );  }  } |

Khối code trên sẽ tạo ra một nút bấm và một doạn chữ với giá trị ­**\_counter** tương ứng với số lần được ấn vào trong nút bấm, mỗi lần ta bấm vào nút bấm thì sẽ gọi lại hàm **incrementCounter()** và **setState()** sẽ kích hoạt việc xây dựng lại Widget. Do đó phương thức **build** sẽ được gọi lại và giá trị **­\_counter** cập nhật lại trong widget text (mỗi lần bấm sẽ tăng biến counter lên 1).

* **StatelessWidget**

Stateless Widget là một widget trong Flutter không lưu trữ trạng thái và không thay đổi giao diện theo thời gian. Nó chỉ hiển thị dữ liệu dựa trên thông tin được truyền vào. Stateless widget được sử dụng cho các trường hợp hiển thị nội dung tĩnh, không cần cập nhật theo tương tác của người dùng hoặc thay đổi dữ liệu bên ngoài.

Đặc điểm chính của Stateless Widget:

* Không lưu trữ trạng thái: Stateless widget không có biến lưu trữ trạng thái thay đổi theo thời gian.
* Giao diện tĩnh: Giao diện của stateless widget chỉ được xác định dựa trên dữ liệu được truyền vào và không thay đổi khi widget được xây dựng lại.
* Hiệu quả: Stateless widget thường hiệu quả hơn stateful widget vì nó không cần cập nhật giao diện thường xuyên.
* Dễ sử dụng: Stateless widget dễ sử dụng và hiểu hơn stateful widget vì nó không cần quản lý trạng thái.

Cách sử dụng Stateless Widget:

1. **Khai báo lớp**: Tạo một lớp kế thừa từ *StatelessWidget*.
2. **Constructor**: Định nghĩa constructor để nhận dữ liệu đầu vào.
3. **Phương thức build**: Phân phối widget con và xác định giao diện widget.
4. **Sử dụng**: Tạo instance của stateless widget và truyền dữ liệu vào constructor.

|  |
| --- |
| // Một ví dụ cho StatelessWidget  class MyStatelessWidget extends StatelessWidget {  final String title;  final String content;  const MyStatelessWidget({  Key? key,  required this.title,  required this.content,  }) : super(key: key);  @override  Widget build(BuildContext context) {  return Column(  children: [  Text(  title,  style: const TextStyle(fontSize: 24, fontWeight: FontWeight.bold),  ),  const SizedBox(height: 16),  Text(  content,  style: const TextStyle(fontSize: 16),  ),  ],  );  }  } |

Khối code trên sẽ tạo cho ta một giao diện tĩnh gồm một cột **Column** và 2 đoạn **Text.** So với StatefulWidget thì ta có một bảng so sánh như sau:

|  |  |  |
| --- | --- | --- |
| Đặc điểm | StatelessWidget | StatefulWidget |
| Trạng thái | Không lưu trữ | Lưu trữ và quản lý |
| Giao diện | Tĩnh | Thay đổi theo thời gian |
| Hiệu quả | Hiệu quả hơn | Kém hiệu quả hơn |
| Độ khó | Dễ sử dụng | Khó sử dụng hơn |

Vậy sau khi trình bày được cả 2 cách quản lý state, ta đúc kết được như sau:

* **Kết luận:**
  + Sử dụng **StatelessWidget** cho các trường hợp hiển thị nội dung tĩnh, không cần cập nhật theo thời gian.
  + Sử dụng **StatefulWidget** cho các trường hợp cần hiển thị nội dung thay đổi theo thời gian, ví dụ như bộ đếm số, danh sách thay đổi …

### 2.1.2. Bố cục giao điện (Layout)

#### 2.1.2.1. Khái niệm bố cục và bố trí trong Flutter

Khái niệm chính của cơ chế bố trí là widget. Chúng ta biết rằng sự flutter giả định mọi thứ như một widget. Vì vậy, hình ảnh, biểu tượng, văn bản và thậm chí cả bố cục( layout) của ứng dụng của ta đều là widget. Ở đây, một số thứ ta không thấy trên giao diện người dùng ứng dụng của mình, chẳng hạn như các hàng (row), cột (column) và lưới sắp xếp (grid layout), ràng buộc (constant) và căn chỉnh các widget hiển thị cũng là các widget.

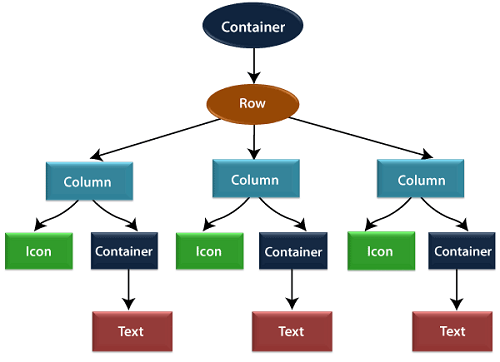
Flutter cho phép chúng ta tạo bố cục bằng cách soạn nhiều widget để xây dựng các widget phức tạp hơn. Ví dụ , chúng ta có thể thấy hình ảnh dưới đây hiển thị ba biểu tượng với nhãn bên dưới mỗi biểu tượng:

A green arrow pointing to a route

Description automatically generated

Hình 2. 9. Sắp xếp bố cục trong Flutter

Trong hình ảnh thứ hai, chúng ta có thể thấy bố cục trực quan của hình ảnh trên. Hình ảnh này hiển thị một hàng gồm ba cột và các cột này chứa một biểu tượng và nhãn:



Hình 2. 10. Minh hoạ về cây bố cục trong Flutter

Trong hình trên, vùng chứa là một lớp widget cho phép chúng ta tùy chỉnh widget con. Nó chủ yếu được sử dụng để thêm đường viền, đệm, lề, màu nền và nhiều thứ khác. Tại đây, widget văn bản nằm dưới vùng chứa để thêm lề. Toàn bộ hàng cũng được đặt trong một vùng chứa để thêm lề và phần đệm xung quanh hàng. Ngoài ra, phần còn lại của giao diện người dùng được kiểm soát bởi các thuộc tính như màu sắc, kiểu văn bản, v.v.

Các bước sau đây cho biết cách bố trí Widget trong Flutter:

* **Bước 1:** Đầu tiên, ta cần chọn một Bố cục widget.
  + Bước đầu tiên là lựa chọn loại bố cục widget phù hợp cho giao diện ta muốn tạo. Flutter cung cấp nhiều widget bố cục khác nhau để sắp xếp và định vị các widget con, ví dụ như:
    - **Row:** Sắp xếp widget con theo chiều ngang (từ trái sang phải).
    - **Column:** Sắp xếp widget con theo chiều dọc (từ trên xuống dưới).
    - **Stack:** Xếp chồng các widget con lên nhau.
    - **Wrap:** Bọc các widget con trong một dòng hoặc cột, tự điều chỉnh kích thước theo không gian có sẵn.
    - **Center:** Đặt widget con ở trung tâm của widget cha.
  + Lựa chọn bố cục phù hợp phụ thuộc vào nhu cầu cụ thể và cấu trúc giao diện ta muốn xây dựng.
* **Bước 2:** Tiếp theo, tạo một widget hiển thị.
  + Sau khi chọn bố cục, ta cần tạo một hoặc nhiều widget để hiển thị nội dung mong muốn. Flutter cung cấp nhiều loại widget khác nhau để hiển thị văn bản, hình ảnh, nút bấm, v.v.
    - Ví dụ: sử dụng widget **Text** để hiển thị văn bản, **Image** để hiển thị hình ảnh, **ElevatedButton** để tạo nút bấm.
  + Mỗi widget có các thuộc tính riêng để định cấu hình giao diện và hành vi của nó.
  + Ta cần thiết lập các thuộc tính phù hợp để widget hiển thị nội dung và chức năng như mong muốn.
* **Bước 3:** Sau đó, thêm widget hiển thị vào widget layout.
  + Sau khi tạo widget hiển thị, ta cần thêm chúng vào widget bố cục đã chọn ở Bước 1.
  + Mỗi widget bố cục có phương thức hoặc thuộc tính để thêm widget con
    - Ví dụ: *children* cho **Row**, **Column**, **Stack**, **Wrap**, v.v.
* **Bước 4:** Cuối cùng, thêm widget bố cục vào trang mà ta muốn hiển thị.
  + Bước cuối cùng là thêm widget bố cục chứa các widget hiển thị vào trang mà ta muốn hiển thị.
  + Flutter cung cấp nhiều loại widget trang khác nhau, ví dụ như **Scaffold**, **MaterialApp**, v.v.
    - Widget trang thường là widget gốc của giao diện ứng dụng.
  + Thêm widget bố cục vào widget trang sẽ hiển thị nội dung của widget con trên màn hình

Ta có một đoạn code minh hoạ như sau:

|  |
| --- |
| // Widget hiển thị văn bản "Hello, Flutter!"  Text(  'Hello, Flutter!',  style: TextStyle(fontSize: 24),  )  // Widget bố cục Column  Column(  children: [  // Widget hiển thị văn bản "Hello, Flutter!" được thêm vào bố cục  Text(  'Hello, Flutter!',  style: TextStyle(fontSize: 24),  ),  ],  )  // Widget trang Scaffold hiển thị widget bố cục Column  Scaffold(  appBar: AppBar(  title: Text('Ví dụ Flutter'),  ),  body: Column(  children: [  // Widget bố cục Column được thêm vào widget trang Scaffold  Column(  children: [  Text(  'Hello, Flutter!',  style: TextStyle(fontSize: 24),  ),  ],  ),  ],  ),  ) |

Tóm lại, việc tạo và hiển thị widget đơn giản trong Flutter bao gồm các bước cơ bản: chọn bố cục widget, tạo widget hiển thị, thêm widget hiển thị vào bố cục, và thêm bố cục widget vào trang. Hiểu rõ các bước này và sử dụng các widget phù hợp sẽ giúp ta xây dựng giao diện người dùng Flutter hiệu quả và đáp ứng nhu cầu thiết kế.

A screenshot of a phone

Description automatically generated

Hình 2. 11. Một ví dụ về xây dựng bố cục của một ứng dụng trong thực tế

#### 2.1.2.2. Một số Widget bố cục

Trong Flutter, ta phân loại Widget bố cục thành 2 loại:

1. widget đơn
2. widget đa

* **Widget đơn**

Widget bố cục con duy nhất là một loại widget, có thể chỉ có **một widget** bên trong widget bố cục mẹ. Các widget này cũng có thể chứa chức năng bố cục đặc biệt. Flutter cung cấp cho chúng ta nhiều widget con để làm cho giao diện người dùng của ứng dụng trở nên hấp dẫn. Nếu chúng ta sử dụng các widget này một cách thích hợp, nó có thể tiết kiệm thời gian của chúng ta và làm cho code ứng dụng dễ đọc hơn. Danh sách các loại widget đơn lẻ khác nhau là:

1. **Container:** Đây là widget bố cục phổ biến nhất cung cấp các tùy chọn có thể tùy chỉnh để đặt màu, định vị và định cỡ các widget.

|  |
| --- |
| Center(  child: Container(  margin: const EdgeInsets.all(15.0),  color: Colors.blue,  width: 42.0,  height: 42.0,  ),  ) |

1. **Padding:** Nó là một widget được sử dụng để sắp xếp widget con của nó theo khoảng đệm đã cho. Nó chứa *EdgeInsets* là một lớp trong Flutter đại diện cho các khoảng cách (padding và margin) xung quanh một widget, cho phía mong muốn mà ta muốn cung cấp đệm.

|  |
| --- |
| const Greetings(  child: Padding(  padding: EdgeInsets.all(14.0),  child: Text('Hello Cafedev!'),  ),  ) |

1. **Center:** widget này cho phép ta căn giữa widget trong chính nó.
2. **Align:** Đây là một widget, căn chỉnh widget của nó trong chính nó và định kích thước nó dựa trên kích thước của đối tượng con. Nó cung cấp nhiều quyền kiểm soát hơn để đặt widget ở vị trí chính xác mà ta cần.

|  |
| --- |
| Center(  child: Container(  height: 110.0,  width: 110.0,  color: Colors.blue,  child: Align(  alignment: Alignment.topLeft,  child: FlutterLogo(  size: 50,  ),  ),  ),  ) |

1. **SizedBox:** widget này cho phép ta cung cấp kích thước được chỉ định cho widget thông qua tất cả các màn hình.

|  |
| --- |
| SizedBox(  width: 300.0,  height: 450.0,  child: const Card(child: Text('Hello Cafedev!')),  ) |

1. **AspectRatio:** widget này cho phép ta giữ kích thước của widget theo một tỷ lệ khung hình được chỉ định.

|  |
| --- |
| AspectRatio(  aspectRatio: 5/3,  child: Container(  color: Colors.bluel,  ),  ), |

1. **Baseline:** widget này thay đổi widget theo đường cơ sở của widget con bên trong.

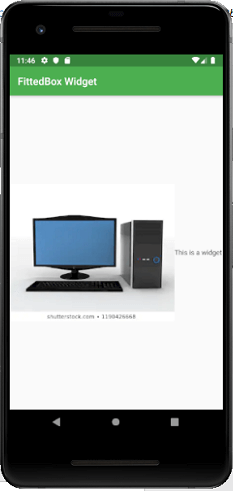
|  |
| --- |
| child: Baseline(  baseline: 30.0,  baselineType: TextBaseline.alphabetic,  child: Container(  height: 60,  width: 50,  color: Colors.blue,  ),  ) |

1. **ConstrainedBox:** Đây là một widget cho phép ta buộc các ràng buộc bổ sung lên widget con của nó. Nó có nghĩa là ta có thể buộc widget con có một ràng buộc cụ thể mà không làm thay đổi các thuộc tính của widget con.

|  |
| --- |
| ConstrainedBox(  constraints: new BoxConstraints(  minHeight: 150.0,  minWidth: 150.0,  maxHeight: 300.0,  maxWidth: 300.0,  ),  child: new DecoratedBox(  decoration: new BoxDecoration(color: Colors.red),  ),  ), |

1. **CustomSingleChildLayout:** Nó là một widget, chuyển từ bố cục của con đơn thành một delegate. Người được ủy quyền (delegate) quyết định vị trí của widget và cũng được sử dụng để xác định kích thước của widget.
2. **FittedBox:** Nó chia tỷ lệ và định vị widget theo sự phù hợp được chỉ định.

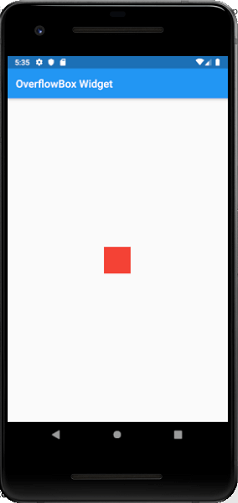
|  |
| --- |
| import 'package:flutter/material.dart';    void main() => runApp(MyApp());    class MyApp extends StatelessWidget {  // It is the root widget of your application.  @override  Widget build(BuildContext context) {  return MaterialApp(  title: 'Multiple Layout Widget',  debugShowCheckedModeBanner: false,  theme: ThemeData(  // This is the theme of your application.  primarySwatch: Colors.green,  ),  home: MyHomePage(),  );  }  }  class MyHomePage extends StatelessWidget {    @override  Widget build(BuildContext context) {  return Scaffold(  appBar: AppBar(title: Text("FittedBox Widget")),  body: Center(  child: FittedBox(child: Row(  children: <Widget>[  Container(  child: Image.asset('assets/computer.png'),  ),  Container(  child: Text("This is a widget"),  )  ],  ),  fit: BoxFit.contain,  )  ),  );  }  } |



Hình 2. 12. Ví dụ về cách sử dụng Widget FittedBox

1. **FractionallySizedBox:** Nó là một widget cho phép kích thước của widget con của nó theo phần nhỏ của không gian có sẵn.
2. **LimitedBox:** widget này cho phép chúng ta giới hạn kích thước của nó chỉ khi nó không bị giới hạn.
3. **Offstage:** Nó được sử dụng để đo kích thước của một widget mà không cần đưa nó lên màn hình.
4. **OverflowBox:** Nó là một widget, cho phép áp đặt các ràng buộc khác nhau đối với widget con của nó so với nó nhận được từ cha mẹ. Nói cách khác, nó cho phép con làm tràn widget cha.

|  |
| --- |
| import 'package:flutter/material.dart';    void main() => runApp(MyApp());    class MyApp extends StatelessWidget {  // It is the root widget of your application.  @override  Widget build(BuildContext context) {  return MaterialApp(  title: 'Single Layout Widget',  debugShowCheckedModeBanner: false,  theme: ThemeData(  // This is the theme of your application.  primarySwatch: Colors.blue,  ),  home: MyHomePage(),  );  }  }  class MyHomePage extends StatelessWidget {    @override  Widget build(BuildContext context) {  return Scaffold(  appBar: AppBar(  title: Text("OverflowBox Widget"),  ),  body: Center(  child: Container(  height: 50.0,  width: 50.0,  color: Colors.red,  child: OverflowBox(  minHeight: 70.0,  minWidth: 70.0,  child: Container(  height: 50.0,  width: 50.0,  color: Colors.blue,  ),  ),  ),  ),  );  }  } |



Hình 2. 13. Ví dụ về cách sử dụng Widget OverflowBox

* **Widget đa**

Đa widget là một loại widget chứa **nhiều hơn một widget con bên trong** và cách bố trí của các widget này là **duy nhấ**t. Ví dụ: widget Row bố trí widget theo hướng ngang và widget Column bố trí widget theo hướng dọc. Nếu chúng ta kết hợp widget Row và Column, thì nó có thể xây dựng bất kỳ cấp độ nào của widget phức tạp.

Ta có đoạn code minh hoạ như sau:

|  |
| --- |
| import 'package:flutter/material.dart';    void main() => runApp(MyApp());    class MyApp extends StatelessWidget {  // It is the root widget of your application.  @override  Widget build(BuildContext context) {  return MaterialApp(  title: 'Multiple Layout Widget',  debugShowCheckedModeBanner: false,  theme: ThemeData(  // This is the theme of your application.  primarySwatch: Colors.blue,  ),  home: MyHomePage(),  );  }  }  class MyHomePage extends StatelessWidget {  @override  Widget build(BuildContext context) {  return Center(  child: Container(  alignment: Alignment.center,  color: Colors.white,  child: Row(  children: <Widget>[  Expanded(  child: Text('Peter', textAlign: TextAlign.center),  ),  Expanded(  child: Text('John', textAlign: TextAlign.center ),    ),  Expanded(  child: FittedBox(  fit: BoxFit.contain, // otherwise the logo will be tiny  child: const FlutterLogo(),  ),  ),  ],  ),  ),  );  }  } |



Hình 2. 14. Kết hợp giữa Widget Row và Column

Danh sách các Widget đa gồm:

* **Row:** Nó cho phép sắp xếp các widget con của nó theo hướng nằm ngang.
* **Column:** Nó cho phép sắp xếp các widget con của nó theo hướng dọc.
* **ListView:** Đây là widget cuộn phổ biến nhất cho phép chúng ta sắp xếp các widget con của nó lần lượt theo hướng cuộn.
* **GridView:** Nó cho phép chúng ta sắp xếp các widget con của nó dưới dạng một mảng widget 2D, có thể cuộn được. Nó bao gồm một mô hình lặp lại của các ô được sắp xếp theo bố cục ngang và dọc.
* **Expanded:** Nó cho phép tạo các con của widget Hàng và Cột chiếm diện tích tối đa có thể.Table: Nó là một widget cho phép chúng ta sắp xếp các con của nó trong một widget dựa trên bảng.
* **Flow:** Nó cho phép chúng ta triển khai widget dựa trên luồng.
* **Stack:** Đây là một widget thiết yếu, chủ yếu được sử dụng để chồng lên một số widget. Nó cho phép ta đặt nhiều lớp lên màn hình. Ví dụ sau đây giúp ta hiểu điều đó.

|  |
| --- |
| import 'package:flutter/material.dart';    void main() => runApp(MyApp());    class MyApp extends StatelessWidget {  // It is the root widget of your application.  @override  Widget build(BuildContext context) {  return MaterialApp(  title: 'Multiple Layout Widget',  debugShowCheckedModeBanner: false,  theme: ThemeData(  // This is the theme of your application.  primarySwatch: Colors.blue,  ),  home: MyHomePage(),  );  }  }  class MyHomePage extends StatelessWidget {  @override  Widget build(BuildContext context) {  return Center(  child: Container(  alignment: Alignment.center,  color: Colors.white,  child: Stack(  children: <Widget>[  // Max Size  Container(  color: Colors.blue,  ),  Container(  color: Colors.pink,  height: 400.0,  width: 300.0,  ),  Container(  color: Colors.yellow,  height: 220.0,  width: 200.0,  )  ],  ),  ),  );  }  } |

A cell phone with a colorful screen

Description automatically generated

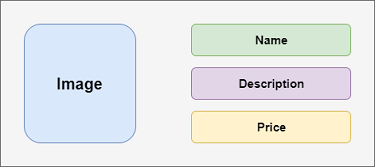
Hình 2. 15. Một ví dụ minh hoạ của Widget Stack được sử dụng như thế nào

#### 2.1.2.3. Xây dựng thử một bố cục phức tạp

Trong phần này, chúng ta sẽ tìm hiểu cách có thể tạo giao diện người dùng phức tạp bằng cách sử dụng cả widget bố cục đơn và nhiều widget. Khung bố cục cho phép ta tạo bố cục giao diện người dùng phức tạp bằng cách lồng các hàng và cột vào bên trong các hàng và cột. Bây giờ, trong bài báo cáo này sẽ tạo một ví dụ về giao diện người dùng phức tạp bằng cách **tạo danh sách sản phẩm**. Ta có đoạn code mẫu như sau:

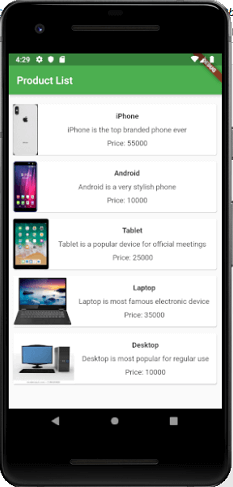
|  |
| --- |
| import 'package:flutter/material.dart';    void main() => runApp(MyApp());    class MyApp extends StatelessWidget {  // It is the root widget of your application.  @override  Widget build(BuildContext context) {  return MaterialApp(  title: 'Flutter Demo Application', theme: ThemeData(  primarySwatch: Colors.green,),  home: MyHomePage(title: 'Complex layout example'),  );  }  }  class MyHomePage extends StatelessWidget {  MyHomePage({Key key, this.title}) : super(key: key);  final String title;    @override  Widget build(BuildContext context) {  return Scaffold(  appBar: AppBar(title: Text("Product List")),  body: ListView(  padding: const EdgeInsets.fromLTRB(3.0, 12.0, 3.0, 12.0),  children: <Widget>[  ProductBox(  name: "iPhone",  description: "iPhone is the top branded phone ever",  price: 55000,  image: "iphone.png"  ),  ProductBox(  name: "Android",  description: "Android is a very stylish phone",  price: 10000,  image: "android.png"  ),  ProductBox(  name: "Tablet",  description: "Tablet is a popular device for official meetings",  price: 25000,  image: "tablet.png"  ),  ProductBox(  name: "Laptop",  description: "Laptop is most famous electronic device",  price: 35000,  image: "laptop.png"  ),  ProductBox(  name: "Desktop",  description: "Desktop is most popular for regular use",  price: 10000,  image: "computer.png"  ),  ],  )  );  }  }  class ProductBox extends StatelessWidget {  ProductBox({Key key, this.name, this.description, this.price, this.image}) :  super(key: key);  final String name;  final String description;  final int price;  final String image;    Widget build(BuildContext context) {  return Container(  padding: EdgeInsets.all(2),  height: 110,  child: Card(  child: Row(  mainAxisAlignment: MainAxisAlignment.spaceEvenly,  children: <Widget>[  Image.asset("assets/" + image),  Expanded(  child: Container(  padding: EdgeInsets.all(5),  child: Column(  mainAxisAlignment: MainAxisAlignment.spaceEvenly,  children: <Widget>[  Text(  this.name, style: TextStyle(  fontWeight: FontWeight.bold  )  ),  Text(this.description), Text(  "Price: " + this.price.toString()  ),  ],  )  )  )  ]  )  )  );  }  } |

Trong đoạn code trên, chúng ta tạo **ProductBox** widget chứa các chi tiết của sản phẩm, chẳng hạn như hình ảnh, tên, giá và mô tả. Trong widget ProductBox, sử dụng các widget sau: Container, Row, Column, Expanded, Card, Text, Image, v.v. widget này có bố cục sau:



Hình 2. 16. Bố cục của ví dụ ProductBox Widget

Sau khi chạy trên giả lập android, ta có kết quả như sau:



Hình 2. 17. Kết quả chạy thử giả lập trong ví dụ bố cục

### 2.1.3. Cử chỉ giao diện (Gestures)

Cử chỉ (Gestures) là một tính năng thú vị trong Flutter cho phép chúng ta tương tác với ứng dụng di động (hoặc bất kỳ thiết bị dựa trên cảm ứng). Nói chung, cử chỉ xác định bất kỳ hành động hoặc chuyển động vật lý nào của người dùng nhằm mục đích kiểm soát thiết bị di động. Một số ví dụ về cử chỉ là:

* Khi màn hình di động bị khóa, ta trượt ngón tay trên màn hình để mở khóa.
* Nhấn vào một nút trên màn hình điện thoại di động của ta và nhấn và giữ biểu tượng ứng dụng trên thiết bị dựa trên cảm ứng để kéo biểu tượng đó qua các màn hình.

Flutter chia hệ thống cử chỉ thành hai lớp khác nhau, được đưa ra dưới đây:

1. Con trỏ (Pointers)
2. Cử chỉ (Gestures)

#### 2.1.3.1. Con trỏ

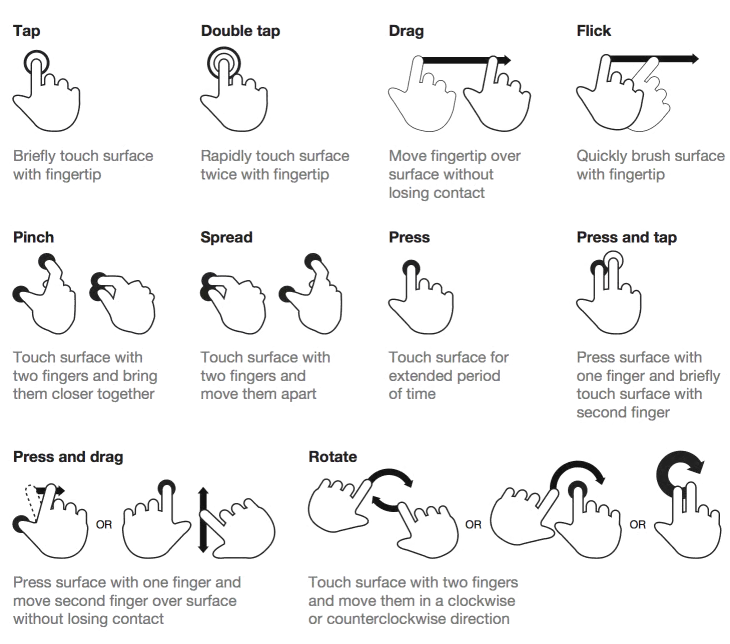
Con trỏ (Pointers) là lớp đầu tiên đại diện cho dữ liệu thô về tương tác của người dùng. Nó có các sự kiện, mô tả **vị trí** và **chuyển động** của các con trỏ như chạm, chuột và kiểu trên màn hình. Flutter không cung cấp bất kỳ cơ chế nào để hủy hoặc dừng các sự kiện con trỏ được gửi đi thêm. Flutter cung cấp một widget **Listener** để lắng nghe các sự kiện con trỏ trực tiếp từ lớp widget. Con trỏ sự kiện được phân loại thành bốn loại chủ yếu:

* **PointerDownEvents:** Nó cho phép con trỏ tiếp xúc với màn hình tại một vị trí cụ thể.
* **PointerMoveEvents:** Nó cho phép con trỏ di chuyển từ vị trí này đến vị trí khác trên màn hình.
* **PointerUpEvents:** Nó cho phép con trỏ dừng tiếp xúc với màn hình.
* **PointerCancelEvents:** Sự kiện này được gửi khi tương tác với con trỏ bị hủy.
* **Ví dụ sử dụng:** Có thể sử dụng để theo dõi vị trí ngón tay của người dùng khi di chuyển trên màn hình.

#### 2.1.3.2. Cử chỉ

Đây là lớp thứ hai đại diện cho các hành động ngữ nghĩa như chạm, vuốt, kéo, phóng to, thu nhỏ, v.v. được nhận dạng từ nhiều sự kiện con trỏ riêng lẻ. Nó cũng có thể gửi nhiều sự kiện tương ứng với vòng đời cử chỉ như kéo bắt đầu, kéo cập nhật và kéo kết thúc. Một số cử chỉ được sử dụng phổ biến được liệt kê dưới đây:

1. **Tap (Chạm):** Có nghĩa là chạm vào bề mặt màn hình từ đầu ngón tay trong một thời gian ngắn rồi thả chúng ra. Cử chỉ này chứa các sự kiện sau:
   1. onTapDown
   2. onTapUp
   3. onTap
   4. onTapCancel
2. **Double Tap (Nhấn đúp):** Nó tương tự như cử chỉ Nhấn, nhưng ta cần nhấn hai lần trong thời gian ngắn. Cử chỉ này chứa các sự kiện sau:
   1. onDoubleTap
3. **Drag (Kéo):** Nó cho phép chúng ta chạm vào bề mặt của màn hình bằng đầu ngón tay và di chuyển nó từ vị trí này sang vị trí khác rồi thả chúng ra. Flutter phân loại kéo thành hai loại:
   1. **Kéo ngang:** Cử chỉ này cho phép con trỏ di chuyển theo hướng ngang. Nó chứa các sự kiện sau:
      1. onHorizontalDragStart
      2. onHorizontalDragUpdate
      3. onHorizontalDragEnd
   2. **Kéo dọc:** Cử chỉ này cho phép con trỏ di chuyển theo hướng thẳng đứng. Nó chứa các sự kiện sau:
      1. onVerticalDragStart
      2. onVerticalDragStart
      3. onVerticalDragStart
4. **Long Press (Nhấn lâu):** Có nghĩa là chạm vào bề mặt của màn hình tại một vị trí cụ thể trong một thời gian dài. Cử chỉ này chứa các sự kiện sau:\
   1. onLongPress
5. **Pan (Di chuyển):** Có nghĩa là chạm vào bề mặt của màn hình bằng đầu ngón tay, có thể di chuyển theo bất kỳ hướng nào mà không cần nhả đầu ngón tay. Cử chỉ này chứa các sự kiện sau:
   1. onPanStart
   2. onPanUpdate
   3. onPanEnd
6. **Pinch (Chụm):** Có nghĩa là chụm (di chuyển ngón tay và ngón cái của một người hoặc đưa chúng lại gần nhau trên màn hình cảm ứng) bề mặt của màn hình bằng cách sử dụng hai ngón tay để phóng to hoặc thu nhỏ màn hình.



Hình 2. 18. Hình ảnh minh hoạ cử chỉ trong Flutter

#### 2.1.3.3. Bắt sự kiện cử chỉ

Flutter cung cấp một tiện ích hỗ trợ tuyệt vời cho tất cả các loại cử chỉ bằng cách sử dụng tiện ích **GestureDetector**. GestureWidget là các widget không trực quan, chủ yếu được sử dụng để phát hiện cử chỉ của người dùng. Ý tưởng cơ bản của bộ phát hiện cử chỉ là một tiện ích **không trạng thái** có chứa các tham số trong hàm tạo của nó cho các sự kiện chạm khác nhau.

Trong một số tình huống, có thể có nhiều bộ phát hiện cử chỉ tại một vị trí cụ thể trên màn hình và sau đó, khung sẽ xác định cử chỉ nào sẽ được gọi. Widget GestureDetector quyết định cử chỉ nào sẽ nhận ra dựa trên lệnh gọi lại nào của nó là không rỗng.

Hãy để chúng ta tìm hiểu cách chúng ta có thể sử dụng các cử chỉ này trong ứng dụng của mình với sự kiện **onTap()** đơn giản và xác định cách GestureDetector xử lý điều này. Ở đây, chúng ta sẽ tạo **một widget hộp**, thiết kế nó theo đặc điểm kỹ thuật mong muốn của chúng ta và sau đó thêm hàm onTap() vào nó. Ta có code như sau:

|  |
| --- |
| import 'package:flutter/material.dart';  void main() => runApp(MyApp());  class MyApp extends StatelessWidget {  const MyApp({super.key});  // This widget is the root of your application.  @override  Widget build(BuildContext context) {  return MaterialApp(  title: 'Flutter Demo Application',  theme: ThemeData(  primarySwatch: Colors.green,  ),  home: MyHomePage(),  );  }  }  class MyHomePage extends StatefulWidget {  const MyHomePage({super.key});  @override  MyHomePageState createState() => MyHomePageState();  }  class MyHomePageState extends State<MyHomePage> {  @override  Widget build(BuildContext context) {  return Scaffold(  appBar: AppBar(  title: const Text('Gestures Example'),  centerTitle: true,  ),  body: Center(  child: GestureDetector(  onTap: () {  print('Box Clicked');  },  child: Container(  height: 60.0,  width: 120.0,  padding: const EdgeInsets.all(10.0),  decoration: BoxDecoration(  color: Colors.blueGrey,  borderRadius: BorderRadius.circular(15.0),  ),  child: const Center(  child: Text('Click Me'),  ),  ),  ),  ),  );  }  } |

Sau đó, ta khởi động máy ảo và tiến hành build chương trình, ta có kết quả như sau:

A screen shot of a cell phone

Description automatically generated

Hình 2. 19. Giao diện bắt sự kiện cử chỉ

Khi ta ấn vào Container xám có dòng chữ **Click me** sẽ có dòng console hiển thị lên kết quả rằng “'Box Clicked” cho thấy khu vực Container đó đã được ấn vào. Và khi ta ấn vào khu vực trắng thì sẽ không có chuyện gì sảy ra trên console.

### 2.1.4. Quản lý trạng thái (State)

Tiếp đến trong phần này, ta sẽ tìm hiểu về quản lý trạng thái (State) và cách chúng ta sử lý nó trong Flutter. Chúng ta biết rằng trong Flutter, mọi thứ đều là một widget. Có thể phân loại widget này thành hai loại, một là **widget Không trạng (Stateless Widget)** thái và một là **widget có Trạng thái(Stateful Widget)**. Widget không trạng thái không có bất kỳ trạng thái bên trong nào. Nó có nghĩa là một khi nó được xây dựng, chúng ta không thể thay đổi hoặc sửa đổi nó cho đến khi chúng được khởi tạo lại. Mặt khác, Widget Stateful là động và có trạng thái. Nó có nghĩa là chúng ta có thể sửa đổi nó một cách dễ dàng trong suốt vòng đời của nó mà không cần khởi động lại nó một lần nữa.

#### 2.1.4.1. Khái niệm

Trạng thái là thông tin có thể **đọc** được khi widget được tạo và có thể **thay đổi hoặc sửa đổi** trong suốt thời gian tồn tại của ứng dụng. Nếu ta muốn thay đổi widget của mình, ta cần cập nhật đối tượng trạng thái, có thể được thực hiện bằng cách sử dụng hàm **setState()** có sẵn cho các **Widget Stateful**. Hàm setState() cho phép chúng ta thiết lập các thuộc tính của đối tượng trạng thái kích hoạt vẽ lại giao diện người dùng.

Quản lý trạng thái (State) là một trong những quy trình phổ biến và cần thiết nhất trong vòng đời của một ứng dụng. Theo tài liệu chính thức, Flutter mang tính chất khai báo. Điều đó có nghĩa là Flutter xây dựng giao diện người dùng của mình bằng cách phản ánh trạng thái hiện tại của ứng dụng của ta. Hình sau giải thích rõ hơn về nơi ta có thể xây dựng giao diện người dùng từ trạng thái ứng dụng.

A close-up of a logo

Description automatically generated

Hình 2. 20. Một giải nghĩa cho Quản lý trạng thái trên Flutter

Chúng ta hãy lấy một ví dụ đơn giản để hiểu khái niệm quản lý trạng thái (State). Giả sử ta đã tạo danh sách khách hàng hoặc sản phẩm trong ứng dụng của mình. Bây giờ, giả sử ta đã thêm động một khách hàng hoặc sản phẩm mới vào danh sách đó. Sau đó, cần phải làm mới danh sách để xem mục mới được thêm vào bản ghi. Vì vậy, bất cứ khi nào ta thêm một mục mới, ta cần phải làm mới danh sách. Kiểu lập trình này yêu cầu quản lý trạng thái (State) xử lý tình huống như vậy để cải thiện hiệu suất. Đó là bởi vì mỗi khi ta thực hiện một thay đổi hoặc cập nhật giống nhau, trạng thái sẽ được làm mới.

Trong Flutter , quản lý trạng thái (State) phân thành hai loại khái niệm, được đưa ra dưới đây:

1. Trạng thái tức thời (Ephemeral State)
2. Trạng thái ứng dụng (App State)

#### 2.1.4.2. Trạng thái tức thời (Ephemeral State)

Trạng thái này còn được gọi là Trạng thái giao diện người dùng hoặc trạng thái địa phương. Nó là một loại trạng thái có liên quan đến **widget cụ thể**, hoặc ta có thể nói rằng nó là một trạng thái chứa trong một widget duy nhất. Trong loại trạng thái này, ta không cần sử dụng các kỹ thuật quản lý state(trạng thái):

* Liên quan đến dữ liệu thay đổi theo thời gian nhanh chóng, không liên quan đến trạng thái tổng thể của ứng dụng.
* Ví dụ: vị trí con trỏ chuột, giá trị trong một trường nhập liệu, kết quả tìm kiếm tức thời.
* Không cần lưu trữ hoặc khôi phục khi ứng dụng thay đổi trạng thái.

Ví dụ phổ biến của trạng thái này là Text Field:

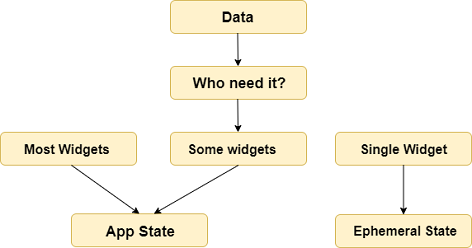
|  |
| --- |
| class MyHomepage extends StatefulWidget {  @override  MyHomepageState createState() => MyHomepageState();  }    class MyHomepageState extends State<MyHomepage> {  String \_name = "Peter";    @override  Widget build(BuildContext context) {  return RaisedButton(  child: Text(\_name),  onPressed: () {  setState(() {  \_name = \_name == "Peter" ? "John" : "Peter";  });  },  );  }  } |

Trong ví dụ trên, biến \_**name** là một trạng thái tạm thời. Ở đây, chỉ có hàm **setState()** bên trong lớp của **StatefulWidget** mới có thể truy cập vào **\_name**. Phương thức xây dựng gọi một hàm **setState()**, hàm này thực hiện sửa đổi các biến trạng thái. Khi phương thức này được thực thi, đối tượng widget sẽ được thay thế bằng đối tượng mới, mang lại giá trị biến được sửa đổi.

#### 2.1.4.3. Trạng thái ứng dụng (App State)

Nó khác với trạng thái tức thời. Đó là một loại trạng thái mà chúng ta muốn **chia sẻ** trên các phần khác nhau của ứng dụng và muốn giữ lại giữa các phiên của người dùng. Do đó, loại trạng thái này có thể được sử dụng trên toàn cầu. Đôi khi nó còn được gọi là trạng thái ứng dụng hoặc trạng thái chia sẻ. Một số ví dụ về trạng thái này là Tùy chọn 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 đã đọc / chưa đọc của các bài báo trong ứng dụng tin tức, v.v.

Sơ đồ sau giải thích sự khác biệt giữa trạng thái tạm thời và trạng thái ứng dụng một cách phù hợp hơn.



Hình 2. 21. Ví dụ minh hoạ cho trạng thái ứng dụng

#### 2.1.4.4. Quản lí trạng thái (State) bằng thư viện Provider

Ví dụ đơn giản nhất về quản lý trạng thái ứng dụng có thể được học bằng cách sử dụng **thư viện Provider.** Việc quản lý trạng thái (State) với thư việc Provider rất dễ hiểu và ít phải viết mã. Provider là thư viện của **bên thứ ba** . Ở đây, chúng ta cần hiểu ba khái niệm chính để sử dụng thư viện này:

1. ChangeNotifier
2. ChangeNotifierProvider
3. Consumer
   * **ChangeNotifier**

**ChangeNotifier** là một lớp đơn giản, cung cấp thông báo thay đổi cho người nghe của nó. Nó dễ hiểu, dễ thực hiện và được tối ưu hóa cho một số lượng nhỏ người nghe. Nó được sử dụng để người nghe quan sát một mô hình để thay đổi. Trong điều này, chúng ta chỉ sử dụng phương thức **notifyListener(**) để thông báo cho người nghe.

Ví dụ: chúng ta hãy xác định một mô hình dựa trên ChangeNotifier. Trong mô hình này, **Lớp Counter** được mở rộng (extend) với lớp **ChangeNotifier**, được sử dụng để thông báo cho người nghe của nó khi chúng ta gọi InformListists(). Đây là phương thức duy nhất cần triển khai trong mô hình ChangeNotifier. Trong ví dụ này, chúng ta đã khai báo hai hàm là tăng và giảm, được sử dụng để tăng và giảm giá trị. Chúng ta có thể gọi phương thức notifyListeners() bất kỳ lúc nào mô hình thay đổi theo cách có thể thay đổi giao diện người dùng của ứng dụng.

|  |
| --- |
| import 'package:flutter/material.dart';  class Counter with ChangeNotifier {  int \_counter;  Counter(this.\_counter);  getCounter() => \_counter; // getter  setCounter(int counter) => \_counter = counter; // setter  void increment() {  \_counter++;  notifyListeners();  }  void decrement() {  \_counter--;  notifyListeners();  }  } |

* + **ChangeNotifierProvider**

ChangeNotifierProvider là widget cung cấp một phiên bản của ChangeNotifier cho con của nó. Nó đến từ thư viện Provider. Các đoạn code sau đây giúp hiểu khái niệm về ChangeNotifierProvider.

Ở đây, chúng ta đã định nghĩa một người xây dựng (builder) những người sẽ tạo một đối tượng mới của **lớp Counter**. ChangeNotifierProvider không xây dựng lại lớp Counter trừ khi có có sự kiện yêu cầu cho việc này. Nó cũng sẽ tự động gọi phương thức **dispose()** trên mô hình Counter khi thể hiện không còn cần thiết nữa.

|  |
| --- |
| class MyApp extends StatelessWidget {  @override  Widget build(BuildContext context) {  return MaterialApp(  theme: ThemeData(  primarySwatch: Colors.indigo,  ),  home: ChangeNotifierProvider<CounterModel>(  builder: (\_) => CounterModel(),  child: CounterView(),  ),  );  }  } |

Nếu có nhu cầu cung cấp nhiều hơn một lớp, ta có thể sử dụng **MultiProvider**. MultiProvider là một widget trong Flutter cho phép ta cung cấp nhiều Provider khác nhau cho các widget con của nó. Nó hoạt động như một danh sách tất cả các Provider đang được sử dụng trong phạm vi của nó. Chúng ta có thể hình dung thông qua đoạn code như sau:

|  |
| --- |
| void main() {  runApp(  MultiProvider(  providers: [  ChangeNotifierProvider(builder: (context) => Counter()),  Provider(builder: (context) => SomeOtherClass()),  ],  child: MyApp(),  ),  );  } |

* + **Consumer**

Consumer là một widget trong Flutter được sử dụng để truy cập dữ liệu từ một Provider. Nó cho phép ta hiển thị dữ liệu được cung cấp bởi Provider trong giao diện widget. Chúng ta có thể hình dung thông qua đoạn code như sau:

|  |
| --- |
| return Consumer<Counter>(  builder: (context, count, child) {  return Text("Total price: ${count.total}");  },  ); |

**Tóm lại: Từ các khối code trên, ta có một khối code ví dụ hoàn chỉnh cho việc sử dụng Provider để quản lý trạng thái (State) như sau:**

|  |
| --- |
| import 'package:flutter/material.dart';  import 'package:provider/provider.dart';  void main() => runApp(const MyApp());  class MyApp extends StatelessWidget {  const MyApp({super.key});  // This widget is the root of your application.  @override  Widget build(BuildContext context) {  return MaterialApp(  title: 'Flutter Demo Application',  home: MyWidget(),  );  }  }  class CounterProvider extends ChangeNotifier {  int \_counter = 0;  int get counter => \_counter;  void increment() {  \_counter++;  notifyListeners();  }  }  class MyWidget extends StatelessWidget {  @override  Widget build(BuildContext context) {  return ChangeNotifierProvider(  create: (context) => CounterProvider(),  child: Consumer<CounterProvider>(builder: (context, provider, child) {  return Column(  children: [  Text('Counter: ${provider.counter}'),  ElevatedButton(  onPressed: () => provider.increment(),  child: const Icon(Icons.add),  )  ],  );  }),  );  }  } |

**A screenshot of a computer

Description automatically generated**

Hình 2. 22. Demo minh hoạ cho quản lý trạng thái (State) bằng thư viện Provider

#### 2.1.4.5. So sánh Quản lý trạng thái bằng cách thông thường và Quản lý trạng thái bằng thư viện Provider

* Quản lý state bằng cách thông thường:
  + Sử dụng **setState** trong widget để cập nhật state của widget đó.
  + Phù hợp cho việc quản lý state đơn giản trong một widget.
  + Có thể dẫn đến mã lộn xộn và khó bảo trì khi ứng dụng phức tạp hơn.
* Quản lý state bằng Provider:
  + Sử dụng Provider để chia sẻ state giữa các widget.
  + Dễ sử dụng và tái sử dụng cho các state phức tạp.
  + Giúp tách biệt logic quản lý state khỏi giao diện widget.
  + Cung cấp giải pháp quản lý state hiệu quả và linh hoạt cho các ứng dụng Flutter.

Ta có một bảng so sánh như sau:

|  |  |  |
| --- | --- | --- |
| **Tính năng** | **Cách thông thường** | **Provider** |
| Phù hợp cho | State đơn giản trong một widget | State phức tạp chia sẻ giữa nhiều widget |
| Dễ sử dụng | Khó khăn khi state phức tạp | Dễ dàng |
| Tái sử dụng | Khó khăn | Dễ dàng |
| Bảo trì | Khó khăn khi ứng dụng phức tạp | Dễ dàng |
| Tách biệt | Logic quản lý state không tách biệt khỏi giao diện | Logic quản lý state tách biệt khỏi giao diện |
| Giải pháp | Giải pháp đơn giản cho state đơn giản | Giải pháp hiệu quả và linh hoạt cho state phức tạp |

**Kết luận:**

1. Nên sử dụng quản lý state bằng cách thông thường cho state đơn giản trong một widget.
2. Nên sử dụng Provider cho state phức tạp chia sẻ giữa nhiều widget.
3. Provider cung cấp giải pháp quản lý state hiệu quả và linh hoạt hơn cho các ứng dụng Flutter.

### 2.1.5. Điều hướng màn hình

Điều hướng và định tuyến (Navigation and Routing) là một số khái niệm cốt lõi của tất cả các ứng dụng di động, cho phép người dùng di chuyển giữa các trang khác nhau. Chúng ta biết rằng mọi ứng dụng di động đều chứa một số màn hình để hiển thị các loại thông tin khác nhau. Ví dụ: một ứng dụng có thể có màn hình chứa nhiều sản phẩm khác nhau. Khi người dùng chạm vào sản phẩm đó, ngay lập tức nó sẽ hiển thị thông tin chi tiết về sản phẩm đó.

Trong Flutter, các màn hình và trang được gọi là **các tuyến** và các tuyến này chỉ là một widget. Trong Android, một tuyến tương tự như **Activity**, trong khi trong iOS, nó tương đương với **ViewController**.

#### 2.1.5.1. Điều hướng bằng Routes

Trong bất kỳ ứng dụng di động nào, điều hướng đến các trang khác nhau xác định quy trình làm việc của ứng dụng và cách xử lý điều hướng được gọi là định tuyến. Flutter cung cấp một lớp định tuyến cơ bản **MaterialPageRoute** và hai phương thức **Navigator.push()** và **Navigator.pop()** cho biết cách điều hướng giữa hai tuyến đường. Các bước sau là bắt buộc để bắt đầu điều hướng trong ứng dụng:

1. Đầu tiên, ta cần tạo hai tuyến đường. Ví dụ: Màn hình A, Màn hình B.
2. Sau đó, điều hướng đến một tuyến đường từ một tuyến đường khác bằng cách sử dụng phương thức **Navigator.push()**. Ví dụ: Di chuyển từ màn hình A sang màn hình B.
3. Cuối cùng, điều hướng đến tuyến đường đầu tiên bằng cách sử dụng phương thức **Navigator.pop()**. Ví dụ: Quay lại từ màn hình B về màn hình A

Ta có một ví dụ như sau:

|  |
| --- |
| class FirstRoute extends StatelessWidget {  @override  Widget build(BuildContext context) {  return Scaffold(  appBar: AppBar(  title: Text('First Route'),  ),  body: Center(  child: RaisedButton(  child: Text('Open route'),  onPressed: () {  // Navigate to second route when tapped.  onPressed: () {  Navigator.push(  context,  MaterialPageRoute(builder: (context) => SecondRoute()),  );  }  },  ),  ),  );  }  }    class SecondRoute extends StatelessWidget {  @override  Widget build(BuildContext context) {  return Scaffold(  appBar: AppBar(  title: Text("Second Route"),  ),  body: Center(  child: RaisedButton(  onPressed: () {  // Navigate back to first route when tapped.  onPressed: () {  Navigator.pop(context);  }  },  child: Text('Go back!'),  ),  ),  );  }  } |

Giải thích khối code trên: 2 class trên tương ứng với 2 màn hình được hiển thị lên máy ảo. Trong Màn hình thứ 1 có nút bấm để di chuyển qua màn hình thức 2 bằng phương thức **Navigator.push()** và màn hình thứ 2 có nút bấm để quay ngược lại màn hình thứ 1 là **Navigator.pop()**.

Phương thức **Navigator.push()** được sử dụng để điều hướng / chuyển sang một tuyến đường / trang / màn hình mới. Ở đây, phương thức **push()** thêm một trang / tuyến đường trên ngăn xếp và sau đó quản lý nó bằng cách sử dụng **Navigator**. Một lần nữa, chúng ta sử dụng lớp **MaterialPageRoute** cho phép chuyển đổi giữa các tuyến bằng cách sử dụng hoạt ảnh dành riêng cho nền tảng.

Và khi chúng ta cần quay ngược trở lại, chúng ta sử dụng phương thức **Navigator.pop()** để đóng tuyến thứ hai và quay lại tuyến đầu tiên. Phương thức **pop()** cho phép chúng ta loại bỏ tuyến đường hiện tại khỏi ngăn xếp, được quản lý bởi Bộ điều hướng.

Chương trình Demo như sau:

A screen shot of a cell phone

Description automatically generated

Hình 2. 23. Màn hình thứ 1 có nút bấm để di chuyển tới màn hình thứ 2

Sau khi chúng ta ấn vào nút “Click Here”, hệ thống sẽ bắt sự kiện và di chuyển chúng ta đến màn hình thứ 2 có nút “Go Back”:

A screen shot of a cell phone

Description automatically generated

Hình 2. 24. Màn hình thứ 2 có nút bấm để quay ngượic về màn hình thứ 1

#### 2.1.5.2. Điều hướng bằng RouteName

Cũng giống như điều hướng bằng Route, RouteName cho phép chúng ta cấu hình các tuyến đường mặc định trước bằng các “KeyName” mà chúng ta đặt ra. Nó sử dụng lớp **Navigator** cùng với việc đặt tên cho các màn hình (route) để thực hiện việc điều hướng. Về cơ bản nó giống với Route nhưng lại có cách thức hoạt động khác hơn nhiều.

Trong hàm **main()** của ứng dụng Flutter, ta cần định nghĩa các **named route**. Mỗi named route bao gồm tên **route** và **widget** tương ứng sẽ được hiển thị khi điều hướng đến route đó. Ta có đoạn code ví dụ như sau:

|  |
| --- |
| MaterialApp(  initialRoute: '/', // Route mặc định khi khởi động ứng dụng  routes: {  '/': (context) => FirstScreen(), // Route '/' hiển thị widget FirstScreen  '/second': (context) => SecondScreen(), // Route '/second' hiển thị widget SecondScreen  },  ); |

Điều hướng bằng route name: Sử dụng phương thức **Navigator.pushNamed()** để điều hướng đến route đã được định nghĩa. Ta chỉ cần truyền tên route vào phương thức này:

|  |
| --- |
| Navigator.pushNamed(context, '/second'); // Điều hướng đến route '/second' |

Truyền tham số (tùy chọn): Ta có thể truyền tham số cho route bằng cách sử dụng thuộc tính **arguments** của phương thức **Navigator.pushNamed()**. Tham số sẽ được truyền vào widget tương ứng của route dưới dạng **Map<String, dynamic>:**

|  |
| --- |
| Navigator.pushNamed(  context,  '/second',  arguments: {'data': 'Hello from FirstScreen'},  ); |

#### 2.1.5.3. So sánh giữa Route và RouteName

1. **Route:**
   1. Ưu điểm:
      1. Dễ dàng sử dụng và hiểu.
      2. Linh hoạt, cho phép ta truyền dữ liệu và tùy chỉnh route.
   2. Nhược điểm:
      1. Có thể trở nên lộn xộn khi ứng dụng phức tạp với nhiều route.
      2. Khó quản lý route name.
2. **RouteName:**
   1. Ưu điểm:
      1. Giữ cho mã gọn gàng và dễ đọc.
      2. Dễ dàng quản lý route name.
      3. Thích hợp cho ứng dụng lớn với nhiều route.
   2. Nhược điểm:
      1. Cần thêm bước định nghĩa named routes.
      2. Ít linh hoạt hơn so với điều hướng bằng route.

Bảng so sánh tổng quan:

|  |  |  |
| --- | --- | --- |
| **Tính năng** | **Routes** | **Route Name** |
| Cách thức hoạt động | Sử dụng widget | Sử dụng tên route |
| Ưu điểm | Dễ sử dụng, linh hoạt | Gọn gàng, dễ quản lý |
| Nhược điểm | Lộn xộn khi phức tạp | Cần định nghĩa named routes |
| Thích hợp cho | Ứng dụng đơn giản | Ứng dụng lớn |

## 2.2. Những Widgets cần biết và phải hiểu

# CHƯƠNG 3. CÁC CHƯƠNG TRÌNH DEMO

# DANH MỤC THAM KHẢO