**Storage**

Trong Flutter, "storage" thường được dùng để đề cập đến cách lưu trữ và quản lý dữ liệu trên các thiết bị di động, chẳng hạn như điện thoại di động hoặc máy tính bảng. Cụ thể, nó liên quan đến việc lưu trữ và truy xuất các dữ liệu như tệp văn bản, hình ảnh, video, cơ sở dữ liệu hoặc các loại dữ liệu khác mà ứng dụng cần sử dụng.

* **Shared Preferences:** Đây là một cách đơn giản để lưu trữ và truy xuất dữ liệu như các cặp key-value. Shared Preferences hỗ trợ lưu trữ các loại dữ liệu như boolean, số nguyên, số thực và chuỗi. Dữ liệu được lưu trữ trong bộ nhớ của ứng dụng và có thể truy xuất mà không cần sử dụng gói plugin bên ngoài.
* **File System:** Flutter cung cấp giao diện để làm việc với hệ thống tập tin trên thiết bị. Bạn có thể lưu trữ dữ liệu trong tệp văn bản, hình ảnh, video hoặc các loại tệp khác. Bạn cũng có thể đọc và ghi dữ liệu vào tệp theo yêu cầu.
* **SQLite:** SQLite là một hệ thống quản lý cơ sở dữ liệu quan hệ nhỏ gọn và rất phổ biến trong ứng dụng di động. Flutter cung cấp gói plugin SQFlite để tương tác với cơ sở dữ liệu SQLite. Điều này cho phép bạn tạo, đọc, cập nhật và xóa các bản ghi trong cơ sở dữ liệu.
* **Firebase:** Firebase là một nền tảng đám mây của Google cung cấp nhiều dịch vụ, bao gồm cơ sở dữ liệu realtime (Firebase Realtime Database) và lưu trữ dựa trên văn bản (Firebase Cloud Firestore). Firebase giúp bạn dễ dàng lưu trữ và đồng bộ hóa dữ liệu trên các thiết bị và nền tảng khác nhau.
* **Hive:** Hive là một cơ sở dữ liệu NoSQL cực kỳ nhanh và nhẹ cho Flutter. Nó hỗ trợ các loại dữ liệu tùy chỉnh và sử dụng native code để đạt hiệu suất tốt hơn so với một số giải pháp lưu trữ khác.
* **Sử dụng External Storage:** Nếu bạn cần lưu trữ dữ liệu ngoài ứng dụng của mình, ví dụ như lưu trữ hình ảnh, video hoặc tệp dữ liệu lớn khác, bạn có thể lưu trữ chúng trong bộ nhớ ngoài của thiết bị.

**3rd Party**

"3rd Party" (Third Party) là thuật ngữ chỉ các công cụ, thư viện, API, gói hoặc phần mềm được phát triển bởi bên thứ ba, tức là không phải do nhà phát triển gốc hoặc người dùng phát triển ứng dụng tạo ra. Trong ngữ cảnh của Flutter, "3rd Party" thường đề cập đến các gói plugin, thư viện và công cụ mà không được tích hợp sẵn trong Flutter SDK, nhưng có thể được sử dụng để mở rộng chức năng của ứng dụng.

Các gói 3rd Party trong Flutter có thể cung cấp các tính năng và chức năng mà Flutter không tích hợp sẵn, giúp chúng ta nhanh chóng và dễ dàng xây dựng các tính năng phức tạp hơn trong ứng dụng của mình. Một số ví dụ về các gói 3rd Party phổ biến trong Flutter bao gồm:

* **http:** Gói này cung cấp các phương thức để thực hiện các yêu cầu HTTP như GET, POST, PUT, DELETE, và cung cấp tích hợp với các API từ các nguồn bên ngoài.
* **provider:** Gói provider cung cấp một giải pháp quản lý trạng thái đơn giản và hiệu quả cho ứng dụng Flutter thông qua mô hình kiến trúc "Provider".
* **firebase\_core và firebase\_auth:** Đây là các gói cung cấp khả năng tích hợp ứng dụng Flutter với dịch vụ Firebase của Google, cho phép xác thực người dùng, lưu trữ dữ liệu trong thời gian thực và thực hiện nhiều tính năng khác.
* **shared\_preferences:** Gói này cung cấp một giao diện dễ sử dụng để lưu trữ và truy xuất các thông tin nhỏ dưới dạng key-value trong ứng dụng.
* **flutter\_svg:** Gói này cho phép bạn hiển thị và tương tác với hình ảnh SVG (Scalable Vector Graphics) trong ứng dụng của mình.

Để sử dụng các gói 3rd Party, chúng ta cần thêm chúng vào tệp **pubspec.yaml** của ứng dụng Flutter và sau đó chạy lệnh **flutter pub get** để cập nhật các phụ thuộc và tải về các gói cần thiết. Sau đó, chúng ta có thể nhập các thư viện và sử dụng chúng trong code của mình.

Việc sử dụng các gói 3rd Party trong Flutter giúp chúng ta tiết kiệm thời gian và công sức phát triển, cho phép chúng ta tập trung vào việc xây dựng các tính năng chính của ứng dụng mà không cần phải xây dựng từ đầu tất cả các chức năng. Tuy nhiên, khi sử dụng các gói 3rd Party, hãy chắc chắn kiểm tra tính tin cậy của chúng, đảm bảo chúng đang được cập nhật và được hỗ trợ bởi cộng đồng đáng tin cậy.

**Libraries**

"Libraries" (thư viện) là các tập hợp mã được viết sẵn để cung cấp các chức năng và tính năng cụ thể. Thư viện giúp chúng ta tái sử dụng mã, giảm thiểu thời gian và công sức cần để xây dựng các tính năng phức tạp, và thường được phân phối dưới dạng các gói có thể được thêm vào ứng dụng của chúng ta.

Trong Flutter, có một số thư viện chính thức và các thư viện được cộng đồng phát triển có thể giúp chúng ta nhanh chóng xây dựng ứng dụng đa dạng. Dưới đây là một số ví dụ về thư viện quan trọng và phổ biến trong Flutter:

* **flutter/material:** Đây là thư viện cung cấp các widget và công cụ để xây dựng giao diện người dùng tuân theo chương trình thiết kế tài liệu vật lý (Material Design). Thư viện này chứa các widget như Text, Button, Icon, AppBar, và nhiều widget khác để xây dựng giao diện ứng dụng đẹp mắt và tương thích với nhiều thiết bị.
* **flutter/cupertino:** Thư viện này cung cấp các widget và công cụ để xây dựng giao diện người dùng theo chương trình thiết kế iOS (Cupertino Design). Điều này cho phép chúng ta tạo ra giao diện giống với ứng dụng iOS trên các thiết bị chạy Flutter.
* **dart:async:** Thư viện này cung cấp các lớp và phương thức để làm việc với các tác vụ bất đồng bộ và quản lý luồng trong Dart.
* **http:** Thư viện này cung cấp các phương thức để thực hiện các yêu cầu HTTP như GET, POST, PUT, DELETE để tương tác với các API từ các nguồn bên ngoài.
* **provider:** Thư viện này giúp quản lý trạng thái ứng dụng một cách hiệu quả thông qua mô hình "Provider". Đây là một cách để chia sẻ trạng thái và dữ liệu giữa các widget mà không cần sử dụng quá nhiều InheritedWidget.
* **firebase\_core và firebase\_auth:** Các thư viện này cho phép chúng ta tích hợp ứng dụng Flutter với dịch vụ Firebase của Google, giúp quản lý xác thực người dùng, lưu trữ dữ liệu trong thời gian thực và thực hiện các tính năng khác.
* **flutter\_bloc và rxdart:** Đây là các thư viện hỗ trợ quản lý trạng thái ứng dụng thông qua mô hình quản lý trạng thái (state management) như BLoC (Business Logic Component) và sử dụng các khái niệm liên quan đến Reactive Programming.
* **intl:** Thư viện này giúp chúng ta quản lý và định dạng văn bản đa ngôn ngữ trong ứng dụng của mình.

**Behavior**

"Behavior" (hành vi) thường đề cập đến cách mà các thành phần giao diện người dùng phản ứng và tương tác với các sự kiện và hành động từ người dùng. Behavior quyết định cách các widget và phần tử trong ứng dụng hoạt động, phản hồi và thay đổi trạng thái dựa trên tương tác của người dùng.

Dưới đây là một số khái niệm quan trọng liên quan đến Behavior trong Flutter:

* **Interactivity:** Đây là tính năng cho phép người dùng tương tác với các widget và phần tử trong ứng dụng. Interactivity được đạt được thông qua việc gắn kết các hàm xử lý sự kiện (event handlers) như onTap, onPressed, onChanged,... để phản hồi với hành động từ người dùng như nhấp chuột, chạm vào màn hình, nhập liệu, và nhiều hành động khác.
* **Animations:** Behavior có thể được sử dụng để thực hiện các hiệu ứng và hoạt hình khi người dùng tương tác với ứng dụng. Flutter cung cấp nhiều cách để tạo ra các hiệu ứng chuyển động và hoạt hình, từ các animations cơ bản như di chuyển, thu/phóng, đến các animations phức tạp hơn như hoạt hình tùy chỉnh.
* **State Management:** Behavior cũng liên quan đến cách quản lý trạng thái của ứng dụng. Khi người dùng tương tác với ứng dụng, trạng thái của các widget và thành phần có thể thay đổi. Sử dụng các mô hình quản lý trạng thái như BLoC, Provider, Redux,… giúp chúng ta quản lý và cập nhật trạng thái một cách hiệu quả và dễ dàng.
* **Gestures:** Flutter hỗ trợ nhiều loại gestures (cử chỉ) như tap, double tap, long press, swipe, pinch, và nhiều gestures khác. Bằng cách gắn kết các hàm xử lý gestures vào các widget, chúng ta có thể định nghĩa hành vi tương ứng với các cử chỉ từ người dùng.
* **Scrolling Behavior:** Scrolling behavior cho phép chúng ta tạo các danh sách cuộn và phần giao diện khác có thể cuộn. Flutter cung cấp các widget như ListView, GridView, SingleChildScrollView và nhiều widget khác để thực hiện scrolling behavior.
* **Focus Behavior:** Đối với các ứng dụng có khả năng tương tác với bàn phím (ví dụ: nhập liệu), Focus Behavior quy định cách widget có focus (được tập trung) và phản ứng với các sự kiện liên quan đến bàn phím.

Tóm lại, Behavior trong Flutter là cách mà các thành phần của ứng dụng phản hồi và tương tác với người dùng, đồng thời cũng định nghĩa cách chúng thay đổi trạng thái và hoạt động dựa trên các hành động khác nhau.

**Components**

"Components" (các thành phần) thường đề cập đến các phần tử cơ bản và đơn giản trong giao diện người dùng của ứng dụng. Các thành phần này thường là những phần tử xây dựng nên giao diện người dùng, và mỗi thành phần thường đảm nhiệm một chức năng hoặc hiển thị một phần cụ thể của thông tin.

Dưới đây là một số ví dụ về các thành phần phổ biến trong Flutter:

* **Text:** Thành phần Text cho phép hiển thị văn bản đơn giản hoặc định dạng trong giao diện người dùng.
* **Button:** Button là thành phần cho phép người dùng tương tác bằng cách nhấn vào nút. Flutter cung cấp nhiều loại nút như RaisedButton, FlatButton, IconButton, và nhiều loại khác.
* **Icon:** Thành phần Icon hiển thị biểu tượng hình ảnh đơn giản hoặc biểu tượng từ các bộ biểu tượng tích hợp.
* **Image:** Thành phần Image cho phép hiển thị hình ảnh trong giao diện người dùng. Chúng ta có thể sử dụng hình ảnh từ mạng hoặc từ tệp được lưu trữ trong ứng dụng.
* **Input Fields:** Các thành phần này cho phép người dùng nhập dữ liệu vào ứng dụng, bao gồm các loại như TextField (dòng văn bản), TextFormField (dòng văn bản với xác nhận), Checkbox (hộp kiểm), Radio (nút radio), và nhiều loại khác.
* **Card:** Thành phần Card cho phép chúng ta tạo ra các hộp chứa thông tin, thường được sử dụng để hiển thị dữ liệu có cấu trúc.
* **AppBar:** AppBar là một thành phần chứa tiêu đề và các tùy chọn điều hướng cho một màn hình trong ứng dụng. Nó thường xuất hiện ở đỉnh của màn hình.
* **ListView:** Thành phần ListView cho phép bạn tạo danh sách cuộn, hiển thị nhiều dòng thông tin hoặc các thành phần khác.
* **GridView:** GridView cho phép bạn xây dựng lưới các thành phần, thường được sử dụng để hiển thị danh sách hình ảnh hoặc thông tin.
* **Container:** Container là một thành phần chứa khác, cho phép chúng ta cung cấp các thuộc tính như màu sắc, kích thước và viền để tạo nên giao diện người dùng phức tạp.

Các thành phần này thường được kết hợp với nhau để tạo thành giao diện người dùng hoàn chỉnh. Flutter cung cấp một loạt các thành phần và widget để bạn có thể tạo ra giao diện ứng dụng đa dạng và hấp dẫn.

**STATE MANAGEMENT**

Trong Flutter, "quản lý trạng thái" (state management) là cách bạn quản lý và cập nhật dữ liệu trong ứng dụng của bạn, đảm bảo rằng các thay đổi về dữ liệu được phản ánh trong giao diện người dùng. Flutter cung cấp các phương pháp khác nhau để xử lý quản lý trạng thái dựa trên sự phức tạp của ứng dụng và sở thích cá nhân. Dưới đây là một số kỹ thuật quản lý trạng thái phổ biến được sử dụng trong Flutter:

* **setState:** Đây là cách đơn giản nhất để quản lý trạng thái trong Flutter. Bạn có thể sử dụng setState trong một widget có trạng thái (stateful widget) để kích hoạt việc xây dựng lại cây widget khi trạng thái thay đổi. Nó hoạt động tốt cho các ứng dụng nhỏ và đơn giản trong đó trạng thái nằm cục bộ của một widget duy nhất.
* **InheritedWidget:** Đây là một widget cho phép bạn chia sẻ dữ liệu xuống cây widget mà không cần chuyển dữ liệu trực tiếp cho mỗi widget. Nó hữu ích để quản lý trạng thái cần được truy cập bởi nhiều widget, chẳng hạn như dữ liệu chủ đề hoặc trạng thái xác thực (authentication).
* **Provider:** Provider là một thư viện quản lý trạng thái phổ biến trong Flutter, sử dụng InheritedWidget bên trong để xử lý trạng thái. Nó cho phép bạn quản lý và chia sẻ trạng thái hiệu quả giữa nhiều widget mà không cần mã khởi tạo dư thừa. Provider được biết đến với sự đơn giản và dễ sử dụng.
* **Bloc (Business Logic Component)**: BLoC là một mô hình kiến trúc phân tách lớp trình chiếu (presentation layer) và logic kinh doanh (business logic). Nó sử dụng streams và sinks để xử lý trạng thái và sự kiện. Gói flutter\_bloc là một cài đặt phổ biến của mô hình BLoC.
* **Redux:** Redux là một kho chứa trạng thái có thể dự đoán được cho ứng dụng Dart/Flutter. Nó duy trì một kho chứa trạng thái toàn cục duy nhất và sử dụng luồng dữ liệu một chiều, giúp làm cho việc thay đổi trạng thái trở nên dễ dàng quản lý và dự đoán hơn.
* **MobX:** MobX là một thư viện quản lý trạng thái sử dụng các đối tượng có thể quan sát (observable) để theo dõi các thay đổi trong trạng thái. Nó nổi tiếng với tính đơn giản và thường được sử dụng trong các kịch bản lập trình phản ứng (reactive programming).
* **GetX:** GetX là một gói Flutter đầy đủ chức năng cung cấp quản lý trạng thái cùng với các tiện ích khác như điều hướng, tiêm phụ thuộc và nhiều hơn nữa. Mục tiêu của GetX là đơn giản, mạnh mẽ và hiệu quả.
* **Riverpod:** Riverpod là một sự lựa chọn thay thế cho Provider, tập trung vào cải thiện API và làm cho việc kiểm thử và quản lý trạng thái trở nên dễ dàng hơn.

Hãy nhớ rằng việc chọn kỹ thuật quản lý trạng thái phụ thuộc chủ yếu vào sự phức tạp của ứng dụng của bạn và sự quen thuộc với một phương pháp cụ thể. Các dự án khác nhau có thể có lợi từ các giải pháp quản lý trạng thái khác nhau, vì vậy hãy đánh giá các yêu cầu cụ thể của ứng dụng của bạn trước khi quyết định sử dụng phương pháp nào. Ngoài ra, cảnh quan quản lý trạng thái trong Flutter luôn luôn tiếp tục phát triển, nên nên theo dõi những cập nhật và phương pháp tốt nhất mới nhất.

**Quanlity Assurance**

Đảm bảo chất lượng (QA) trong Flutter đề cập đến quá trình đảm bảo rằng ứng dụng Flutter của bạn đáp ứng các tiêu chuẩn cao nhất về chất lượng, độ tin cậy và sự hài lòng của người dùng. Nó bao gồm một loạt các hoạt động và thực tiễn nhằm xác định và ngăn ngừa lỗi, đảm bảo hiệu suất nhất quán và mang lại trải nghiệm người dùng bóng bẩy và không có lỗi. Dưới đây là một số khía cạnh chính của đảm bảo chất lượng trong Flutter:

1. \*\*Kiểm tra đơn vị và Tự động hóa kiểm tra\*\*: Viết kiểm tra đơn vị cho các thành phần và tính năng riêng lẻ của ứng dụng Flutter của bạn để đảm bảo rằng chúng hoạt động chính xác. Triển khai tự động hóa thử nghiệm bằng cách sử dụng khung thử nghiệm của Flutter để tự động chạy thử nghiệm bất cứ khi nào thay đổi mã được thực hiện.

2. \*\*Kiểm tra tích hợp\*\*: Tiến hành kiểm tra tích hợp để đảm bảo rằng các phần khác nhau trong ứng dụng của bạn hoạt động liền mạch với nhau. Điều này giúp xác định bất kỳ vấn đề nào có thể phát sinh khi các thành phần khác nhau tương tác với nhau.

3. \*\*Kiểm tra giao diện người dùng\*\*: Thực hiện kiểm tra giao diện người dùng để xác minh rằng giao diện người dùng hoạt động như mong đợi. Các công cụ như `flutter\_driver` của Flutter có thể được sử dụng để tự động kiểm tra giao diện người dùng và đảm bảo rằng các yếu tố trực quan của ứng dụng và tương tác người dùng đang hoạt động bình thường.

4. \*\*Kiểm tra hiệu suất\*\*: Kiểm tra hiệu suất của ứng dụng trong các điều kiện khác nhau, chẳng hạn như các cấu hình thiết bị và điều kiện mạng khác nhau. Xác định và giải quyết bất kỳ tắc nghẽn hiệu suất nào có thể ảnh hưởng đến trải nghiệm người dùng.

5. \*\*Kiểm tra khả năng truy cập\*\*: Đảm bảo rằng ứng dụng của bạn có thể truy cập được đối với người dùng khuyết tật. Flutter cung cấp các công cụ và thư viện để giúp bạn kiểm tra và cải thiện khả năng truy cập ứng dụng của mình.

6. \*\*Kiểm tra bảo mật\*\*: Tiến hành đánh giá bảo mật để xác định các lỗ hổng và rủi ro bảo mật tiềm ẩn trong ứng dụng của bạn. Đảm bảo rằng dữ liệu người dùng được xử lý an toàn và ứng dụng của bạn tuân theo các phương pháp hay nhất để bảo vệ dữ liệu.

7. \*\*Tích hợp liên tục và Triển khai liên tục (CI/CD)\*\*: Thiết lập quy trình CI/CD để tự động hóa quy trình xây dựng, thử nghiệm và triển khai ứng dụng của bạn. Điều này giúp phát hiện sớm các sự cố và đảm bảo rằng chỉ mã chất lượng cao và được kiểm tra kỹ càng mới được đưa vào sản xuất.

8. \*\*Đánh giá mã\*\*: Triển khai quy trình đánh giá mã trong đó các thành viên trong nhóm xem xét mã của nhau để xác định các vấn đề, chia sẻ kiến thức và đảm bảo tuân thủ các tiêu chuẩn viết mã cũng như các phương pháp hay nhất.

9. \*\*Giám sát và ghi nhật ký lỗi\*\*: Triển khai cơ chế giám sát và ghi nhật ký lỗi để theo dõi và phân tích các sự cố và lỗi của ứng dụng trong thời gian thực. Điều này giúp bạn xác định và giải quyết các vấn đề một cách nhanh chóng.

10. \*\*Phản hồi của người dùng và Thử nghiệm beta\*\*: Thu thập phản hồi từ người dùng thực thông qua thử nghiệm beta. Điều này có thể cung cấp thông tin chi tiết có giá trị về các vấn đề về khả năng sử dụng, lỗi và các lĩnh vực cần cải thiện.

11. \*\*Tài liệu\*\*: Duy trì tài liệu rõ ràng và toàn diện về mã, kiến trúc và tính năng của ứng dụng. Mã có tài liệu tốt giúp các nhà phát triển hiểu và duy trì ứng dụng dễ dàng hơn.

12. \*\*Quản lý phát hành và kiểm soát phiên bản\*\*: Sử dụng các hệ thống kiểm soát phiên bản như Git để theo dõi các thay đổi và quản lý các phiên bản khác nhau của ứng dụng của bạn. Lập kế hoạch và quản lý các bản phát hành một cách cẩn thận để đảm bảo quá trình triển khai suôn sẻ.

Bằng cách triển khai các thực hành QA này trong quy trình phát triển Flutter của mình, bạn có thể cung cấp ứng dụng chất lượng cao mang lại trải nghiệm người dùng tích cực và đáp ứng kỳ vọng của người dùng.

**Version Control**

Quản lý phiên bản Flutter cho phép các loại phiên bản Flutter khác nhau có sẵn trên cơ sở dự án. Điều này có nghĩa là chúng ta có thể xác định các loại phiên bản Flutter cụ thể cho các loại dự án khác nhau, nó cho phép chúng ta phát hành nhiều kênh, lưu vào bộ nhớ cache cục bộ. Vì vậy, khi chuyển đổi các phiên bản thì chúng ta không phải đợi thiết lập.

**Flutter Version Management (FVM):**

Khi làm việc trên dự án Flutter, chúng tôi cần phát hành phiên bản cập nhật và ứng dụng, xác minh nó và chuyển đổi các loại SDK khác nhau để kiểm tra nó, điều này khiến chúng tôi mất thời gian phát triển. Để tránh điều này, chúng tôi sử dụng Quản lý phiên bản Flutter, cung cấp cho chúng tôi các loại phiên bản Flutter khác nhau cho máy của chúng tôi. Vì vậy, mỗi lần Flutter có thể kiểm tra ứng dụng dựa trên bản phát hành Flutter cập nhật mà không cần chờ cài đặt và sẽ có thể chuyển sang phiên bản Flutter cho phù hợp.

**Cài đặt FVM:**

Trước hết, chúng ta cần phải đảm bảo rằng flutter đã được cài đặt và flutter có phải là kênh ổn định hay không. nếu không thì Nhập mã dưới đây vào dòng lệnh của bạn.

// set flutter to stable channel

flutter channel stable

// check flutter channel

flutter channel

// output

Flutter channels:

master

dev

beta

\* stable

Sau đó, chúng ta phải xác định xem flutter của chúng ta đã được cài đặt hay chưa, nếu chưa thì trước tiên chúng ta sẽ cài đặt FVM.

$ pub global activate fvm

Bây giờ chúng ta sẽ thấy rằng một số cảnh báo được đưa ra ở cuối quá trình cài đặt, vì vậy chúng ta cần thêm đường dẫn fvm vào tệp cấu hình shell (.bashrc, bash\_profile, v.v.) trước khi thực hiện bước tiếp theo.

export PATH=”$PATH:`pwd`/flutter/bin”$ fvm install

export PATH=”$PATH:`pwd`/bin/cache/dart-sdk/bin”

export PATH=”$PATH:`pwd`/.pub-cache/bin”

#### **Cài đặt SDK**

FVM cho phép chúng ta cài đặt nhiều loại bản phát hành hoặc nhiều kênh. Để cài đặt kênh sử dụng ổn định và để cài đặt phiên bản phát hành flutter, chúng ta sẽ sử dụng v2.0.5 hoặc 1.17.0-dev.3.1 và ngay sau khi chúng ta chạy - bỏ qua thiết lập sau đó nó sẽ bỏ qua quá trình thiết lập.

$ fvm install stable or fvm install 2.0.5

#### **Phiên bản SDK cấu hình dự án:**

Sau đó, chúng ta sẽ thấy rằng Dự án có được cấu hình để sử dụng một phiên bản cụ thể hay không, nếu không, chúng ta sẽ cài đặt nó trên phiên bản thích hợp mà không cần đối số.

$ fvm install

#### **Danh sách các phiên bản đã cài đặt của Flutter:**

Bây giờ bằng cách gõ vào lệnh sau, chúng ta có thể liệt kê phiên bản đã cài đặt trên máy của mình bằng cách sử dụng lệnh dưới đây FVM sẽ lưu trữ phiên bản SDK.

$ fvm list

#### **Nâng cấp phiên bản SDK:**

Sử dụng lệnh nâng cấp phiên bản SDK khi chúng tôi cần nâng cấp phiên bản SDK hiện tại của mình, do đó bạn phải gọi lệnh của SDK Flutter của mình như trong cài đặt Flutter thông thường.

$ fvm flutter upgrade

### Cài đặt IDE:

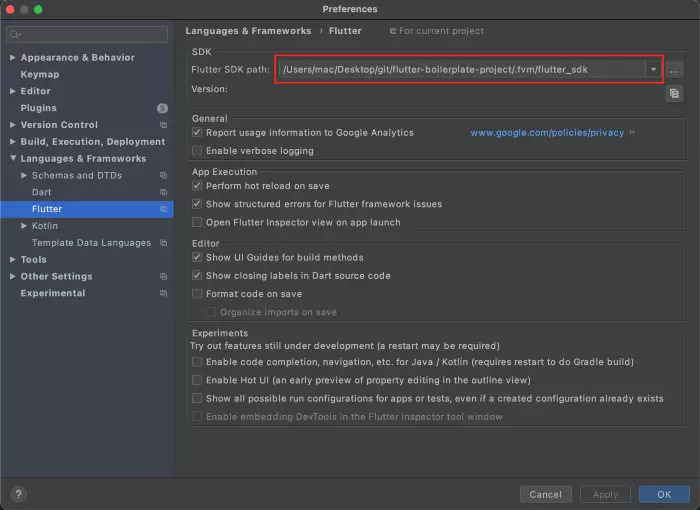
Bây giờ chúng ta sẽ xem cách định cấu hình IDE, dưới đây chúng tôi đã chỉ ra cách định cấu hình trong Android Studio chúng ta hãy xem nó ngay bây giờ.

#### **Android Studio**

Trong thư mục dự án gốc của bạn, hãy sao chép liên kết tượng trưng sau tuyệt đối.

Example: /absolute/path-to-your-project/.fvm/flutter\_sdk

Sau đó, chúng tôi sẽ mở Languages and Frameworks trong menu của Android Studio -> Bây giờ hãy tìm kiếm Flutter và thay đổi đường dẫn đến Flutter SDK. và thực hiện thay đổi. Bây giờ bạn có thể chạy nó với các phiên bản đã chọn và gỡ lỗi nó. Nếu bạn muốn xem cài đặt mới được thực hiện thì chúng ta có thể sử dụng Android Studio sẽ khởi động lại.



**FIREBASE**

**Tìm hiểu về Firebase**

Nếu bạn mới sử dụng Firebase trong Flutter, chúng tôi khuyên bạn nên bắt đầu với video và [phòng thí nghiệm mã hóa Tìm hiểu Firebase cho Flutter :](https://firebase.google.com/codelabs/firebase-get-to-know-flutter)

**Điều kiện tiên quyết**

Trước khi bắt đầu, tài liệu giả định rằng bạn có thể tạo (hoặc có sẵn) dự án Flutter và cũng có tài khoản Firebase đang hoạt động.

**Cài đặt**

Trước khi sử dụng bất kỳ dịch vụ nào của Firebase, bạn phải trước tiên cài đặt plugin firebase\_core, nhiệm vụ của nó là kết nối ứng dụng của bạn với Firebase.

Cài đặt plugin bằng cách chạy lệnh sau từ thư mục gốc của dự án:



Khởi tạo FlutterFire

FlutterFire hiện đã hỗ trợ khởi tạo trực tiếp từ Dart!

Trước khi có thể sử dụng bất kỳ dịch vụ Firebase nào, FlutterFire cần được khởi chạy (bạn có thể coi quá trình này giống như chính FlutterFire "bootstrapping"). Bước khởi tạo là không đồng bộ, nghĩa là bạn sẽ cần ngăn mọi hoạt động sử dụng liên quan đến FlutterFire cho đến khi quá trình khởi tạo hoàn tất.

**Sử dụng FlutterFire CLI**

Để khởi tạo FlutterFire, hãy gọi [initializeApp](https://pub.dev/documentation/firebase_core/latest/firebase_core/Firebase/initializeApp.html)phương thức trên [Firebase](https://pub.dev/documentation/firebase_core/latest/firebase_core/Firebase-class.html)lớp. Phương thức chấp nhận cấu hình ứng dụng dự án Firebase của bạn, cấu hình này có thể được lấy cho tất cả các nền tảng được hỗ trợ bằng cách sử dụng FlutterFire CLI:

****

Sau khi được định cấu hình, một firebase\_options.darttệp sẽ được tạo cho bạn có chứa tất cả các tùy chọn cần thiết để khởi tạo. Ngoài ra, nếu ứng dụng Flutter của bạn hỗ trợ Android thì [plugin Android Google Services Gradle](https://firebase.google.com/docs/android/setup#add-config-file) sẽ tự động được áp dụng cho bạn.

**Tìm hiểu thêm về FlutterFire CLI**

Nếu bạn thêm hỗ trợ cho một nền tảng mới trong ứng dụng Flutter của mình (ví dụ: thêm Android khi ứng dụng của bạn trước đây không hỗ trợ Android) hoặc nếu bạn đưa các dịch vụ Firebase mới vào ứng dụng của mình (ví dụ: thêm ) thì bạn nên định cấu hình lại Firebase cho ứng dụng của mình thông firebase\_databasequa CLI ( flutterfire configure).

**Khởi tạo**

Tiếp theo, các tùy chọn đã tạo cần được cung cấp cho [initializeApp](https://pub.dev/documentation/firebase_core/latest/firebase_core/Firebase/initializeApp.html)phương thức. Vì đây là hoạt động không đồng bộ nên mainchức năng có thể được sửa đổi để đảm bảo quá trình khởi tạo hoàn tất trước khi chạy ứng dụng.

Đầu tiên nhập firebase\_coreplugin và firebase\_options.darttệp được tạo:

A blue rectangular object with a white line

Description automatically generated

Tiếp theo, trong mainhàm, ensure WidgetsFlutterBindingđược khởi tạo và sau đó khởi tạo Firebase:

A computer screen shot of a computer screen

Description automatically generated

Được DefaultFirebaseOptions.currentPlatformnhập từ firebase\_options.darttệp được tạo của chúng tôi.

Sau khi khởi chạy, bạn đã sẵn sàng bắt đầu sử dụng FlutterFire!

**Cài đặt thủ công**

Với tính năng khởi tạo chỉ dành cho Dart hiện được hỗ trợ, việc định cấu hình và cài đặt nền tảng theo cách thủ công không còn cần thiết nữa. Nếu bạn muốn xem tài liệu về khởi tạo thủ công, hãy xem tài liệu [Cài đặt thủ công](https://firebase.flutter.dev/docs/manual-installation) .

**Các bước tiếp theo**

Về bản thân, [firebase\_core](https://pub.dev/documentation/firebase_core/latest/firebase_core/firebase_core-library.html)plugin cung cấp chức năng cơ bản để sử dụng với Firebase. FlutterFire được chia thành các plugin riêng lẻ, có thể cài đặt cho phép bạn tích hợp với một dịch vụ Firebase cụ thể.

Xem tài liệu cho từng plugin để tìm hiểu thêm về chúng.

**NATIVE INTEGRATION**

Hướng dẫn này mô tả cách viết mã dành riêng cho nền tảng tùy chỉnh. Một số chức năng dành riêng cho nền tảng có sẵn thông qua các gói hiện có; xem [bằng cách sử dụng các gói](https://docs.flutter.dev/packages-and-plugins/using-packages) .

Flutter sử dụng một hệ thống linh hoạt cho phép bạn gọi các API dành riêng cho nền tảng bằng ngôn ngữ hoạt động trực tiếp với các API đó:

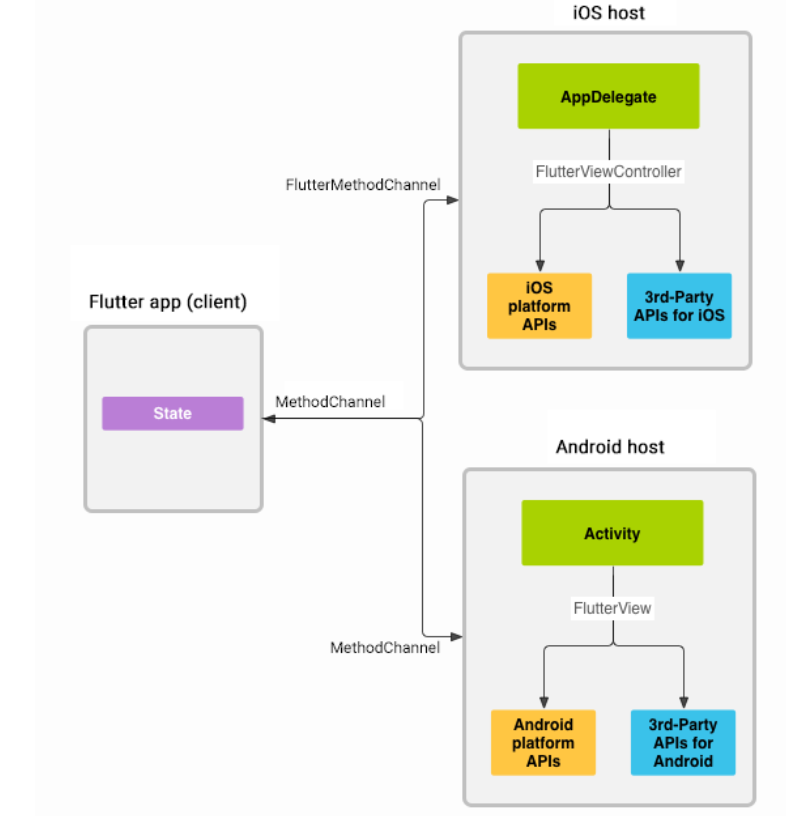
* Kotlin hoặc Java trên Android
* Swift hoặc Objective-C trên iOS
* C++ trên Windows
* Mục tiêu-C trên macOS
* C trên Linux

Hỗ trợ API dành riêng cho nền tảng dựng sẵn của Flutter không dựa vào việc tạo mã, mà dựa vào kiểu truyền thông báo linh hoạt. Ngoài ra, bạn có thể sử dụng gói [Pigeon](https://pub.dev/packages/pigeon) để [gửi các tin nhắn an toàn có cấu trúc](https://docs.flutter.dev/platform-integration/platform-channels#pigeon) bằng cách tạo mã:

* Phần Flutter của ứng dụng gửi tin nhắn đến *máy chủ* của nó , phần không phải Dart của ứng dụng, qua kênh nền tảng.
* Máy *chủ* lắng nghe trên kênh nền tảng và nhận tin nhắn. Sau đó, nó gọi vào bất kỳ số lượng API dành riêng cho nền tảng nào—sử dụng ngôn ngữ lập trình gốc—và gửi phản hồi lại cho ứng *dụng khách* , phần Flutter của ứng dụng.

**Tổng quan về kiến ​​trúc: các kênh nền tảng**

Tin nhắn được truyền giữa máy khách (UI) và máy chủ (nền tảng) bằng cách sử dụng các kênh nền tảng như được minh họa trong sơ đồ này:



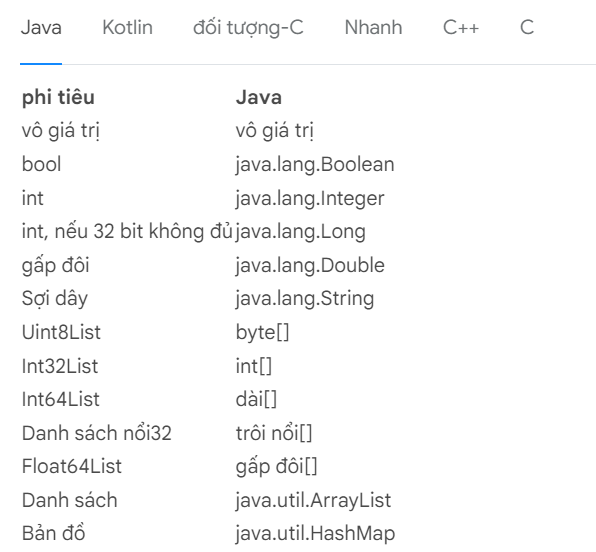
Tin nhắn và phản hồi được truyền không đồng bộ để đảm bảo giao diện người dùng vẫn phản hồi nhanh.

Về phía máy khách, [MethodChannel](https://api.flutter.dev/flutter/services/MethodChannel-class.html)cho phép gửi tin nhắn tương ứng với các cuộc gọi phương thức. Về phía nền tảng, MethodChanneltrên Android ( [MethodChannelAndroid](https://api.flutter.dev/javadoc/io/flutter/plugin/common/MethodChannel.html)) và FlutterMethodChanneltrên iOS ( [MethodChanneliOS](https://api.flutter.dev/objcdoc/Classes/FlutterMethodChannel.html)) cho phép nhận cuộc gọi phương thức và gửi lại kết quả. Các lớp này cho phép bạn phát triển plugin nền tảng với rất ít mã 'bản soạn sẵn'.

**Hỗ trợ các loại dữ liệu kênh nền tảng và codec**

Các kênh nền tảng tiêu chuẩn sử dụng một codec thông báo tiêu chuẩn hỗ trợ tuần tự hóa nhị phân hiệu quả của các giá trị giống như JSON đơn giản, chẳng hạn như booleans, số, Chuỗi, bộ đệm byte cũng như Danh sách và Bản đồ của những giá trị này (xem để biết chi tiết) [StandardMessageCodec](https://api.flutter.dev/flutter/services/StandardMessageCodec-class.html). Quá trình tuần tự hóa và giải tuần tự hóa các giá trị này đến và từ các thư diễn ra tự động khi bạn gửi và nhận các giá trị.

Bảng sau đây cho biết cách nhận các giá trị Dart ở phía nền tảng và ngược lại:

****

**Ví dụ: Gọi mã dành riêng cho nền tảng bằng các kênh nền tảng**

Đoạn mã sau trình bày cách gọi API dành riêng cho nền tảng để truy xuất và hiển thị mức pin hiện tại. Nó sử dụng API Android BatteryManager, API iOS device.batteryLevel, API Windows GetSystemPowerStatusvà UPowerAPI Linux với một thông báo nền tảng duy nhất, getBatteryLevel().

Ví dụ thêm mã dành riêng cho nền tảng bên trong chính ứng dụng chính. Nếu bạn muốn sử dụng lại mã dành riêng cho nền tảng cho nhiều ứng dụng, thì bước tạo dự án sẽ hơi khác một chút (xem [phần phát triển gói](https://docs.flutter.dev/packages-and-plugins/developing-packages#plugin) ), nhưng mã kênh của nền tảng vẫn được viết theo cùng một cách.

**Bước 1: Tạo một dự án ứng dụng mới**

Bắt đầu bằng cách tạo một ứng dụng mới:

* Trong một thiết bị đầu cuối chạy:flutter create batterylevel

Theo mặc định, mẫu của chúng tôi hỗ trợ viết mã Android bằng Kotlin hoặc mã iOS bằng Swift. Để sử dụng Java hoặc Objective-C, hãy sử dụng các cờ -ivà/hoặc -a:

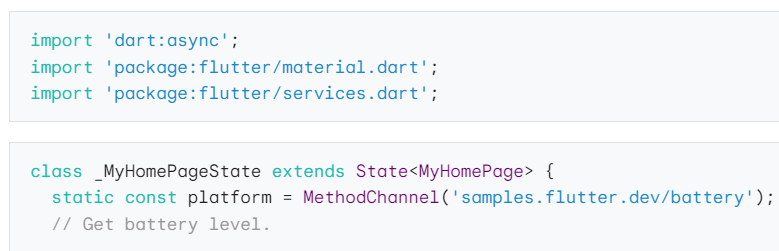
* Trong một thiết bị đầu cuối chạy:flutter create -i objc -a java batterylevel

**Bước 2: Tạo ứng dụng khách nền tảng Flutter**

Lớp của ứng dụng Stategiữ trạng thái ứng dụng hiện tại. Mở rộng điều đó để giữ trạng thái pin hiện tại.

Đầu tiên, xây dựng kênh. Sử dụng một MethodChannelphương thức nền tảng duy nhất trả về mức pin.

Các phía máy khách và máy chủ của kênh được kết nối thông qua tên kênh được truyền trong hàm tạo kênh. Tất cả các tên kênh được sử dụng trong một ứng dụng phải là duy nhất; thêm vào đầu tên kênh một 'tiền tố tên miền' duy nhất, ví dụ: samples.flutter.dev/battery.



Tiếp theo, gọi một phương thức trên kênh phương thức, chỉ định phương thức cụ thể để gọi bằng Stringmã định danh getBatteryLevel. Cuộc gọi có thể không thành công—ví dụ: nếu nền tảng không hỗ trợ API nền tảng (chẳng hạn như khi chạy trong trình giả lập), vì vậy hãy kết thúc cuộc invokeMethodgọi trong câu lệnh thử bắt.

Sử dụng kết quả trả về để cập nhật trạng thái giao diện người dùng bên \_batteryLevel trong setState.

A screen shot of a computer program

Description automatically generated

Cuối cùng, thay thế buildphương thức từ mẫu để chứa một giao diện người dùng nhỏ hiển thị trạng thái pin trong một chuỗi và một nút để làm mới giá trị.

A screen shot of a computer program

Description automatically generated

**Bước 3: Thêm triển khai dành riêng cho nền tảng Android**

**Kotlin**

Bắt đầu bằng cách mở phần máy chủ Android của ứng dụng Flutter của bạn trong Android Studio:

1. Bắt đầu Android Studio
2. Chọn mục menu **Tệp > Mở…**
3. Điều hướng đến thư mục chứa ứng dụng Flutter của bạn và chọn thư mục **android** bên trong nó. Bấm **OK** .
4. Mở tệp MainActivity.ktnằm trong thư mục **kotlin** trong chế độ xem Dự án.

Bên trong configureFlutterEngine()phương thức, tạo một MethodChannelvà gọi setMethodCallHandler(). Đảm bảo sử dụng cùng một tên kênh như đã được sử dụng ở phía máy khách Flutter.

A screenshot of a computer program

Description automatically generated

Thêm mã Android Kotlin sử dụng API pin Android để truy xuất mức pin. Mã này hoàn toàn giống với mã bạn viết trong ứng dụng Android gốc.

Đầu tiên, thêm các mục nhập cần thiết ở đầu tệp:

A screenshot of a computer program

Description automatically generated

Tiếp theo, thêm phương thức sau vào MainActivitylớp, bên dưới configureFlutterEngine()phương thức:

A screenshot of a computer program

Description automatically generated

Bây giờ bạn có thể chạy ứng dụng trên Android. Nếu sử dụng Trình giả lập Android, hãy đặt mức pin trong bảng Điều khiển mở rộng có thể truy cập từ nút **…** trên thanh công cụ.

**Bước 4: Thêm triển khai dành riêng cho nền tảng iOS**

Bắt đầu bằng cách mở phần máy chủ iOS của ứng dụng Flutter của bạn trong Xcode:

1. Bắt đầu Xcode.
2. Chọn mục menu **Tệp > Mở…** .
3. Điều hướng đến thư mục chứa ứng dụng Flutter của bạn và chọn thư mục **ios** bên trong nó. Bấm **OK** .

Thêm hỗ trợ cho Swift trong thiết lập mẫu tiêu chuẩn sử dụng Objective-C:

1. **Mở rộng Runner > Runner** trong Project navigator.
2. Mở tệp AppDelegate.swiftnằm trong **Runner > Runner** trong Project navigator.

Ghi đè application:didFinishLaunchingWithOptions:chức năng và tạo một FlutterMethodChannelliên kết với tên kênh samples.flutter.dev/battery:

A screenshot of a computer program

Description automatically generated

Tiếp theo, thêm mã iOS Swift sử dụng API pin iOS để truy xuất mức pin. Mã này hoàn toàn giống với mã bạn viết trong ứng dụng iOS gốc.

Thêm phần sau đây dưới dạng một phương thức mới ở cuối AppDelegate.swift:

A screen shot of a computer code

Description automatically generated

Cuối cùng, hoàn thành setMethodCallHandler()phương pháp đã thêm trước đó. Bạn cần xử lý một phương thức nền tảng duy nhất, getBatteryLevel()vì vậy hãy kiểm tra điều đó trong callđối số. Việc triển khai phương thức nền tảng này gọi mã iOS được viết ở bước trước. Nếu một phương thức không xác định được gọi, thay vào đó hãy báo cáo phương thức đó.

**Bước 5: Thêm triển khai dành riêng cho nền tảng Windows**

Bắt đầu bằng cách mở phần máy chủ Windows của ứng dụng Flutter của bạn trong Visual Studio:

1. Chạy flutter build windowstrong thư mục dự án của bạn một lần để tạo tệp giải pháp Visual Studio.
2. Bắt đầu Visual Studio.
3. Chọn **Mở một dự án hoặc giải pháp** .
4. Điều hướng đến thư mục chứa ứng dụng Flutter của bạn, sau đó vào thư mục bản **dựng** , sau đó đến thư mục **windows** , sau đó chọn batterylevel.slntệp. Nhấp **vào Mở** .

Thêm triển khai C++ của phương thức kênh nền tảng:

1. Mở rộng **batterylevel > Source Files** trong Solution Explorer.
2. Mở tệp flutter\_window.cpp.

Đầu tiên, thêm các mục cần thiết vào đầu tệp, ngay sau #include "flutter\_window.h":

A screen shot of a computer code

Description automatically generated

Chỉnh sửa FlutterWindow::OnCreatephương thức và tạo một flutter::MethodChannelliên kết với tên kênh samples.flutter.dev/battery: