**Shortcut - Intent - Action**

# CallbackShortcuts

[CallbackShortcuts](https://api.flutter.dev/flutter/widgets/CallbackShortcuts/CallbackShortcuts.html)({[Key](https://api.flutter.dev/flutter/foundation/Key-class.html)? key, required [Map](https://api.flutter.dev/flutter/dart-core/Map-class.html)<[ShortcutActivator](https://api.flutter.dev/flutter/widgets/ShortcutActivator-class.html), [VoidCallback](https://api.flutter.dev/flutter/dart-ui/VoidCallback.html)> bindings, required [Widget](https://api.flutter.dev/flutter/widgets/Widget-class.html) child})

This is similar to but simpler than the [Shortcuts](https://api.flutter.dev/flutter/widgets/Shortcuts-class.html) widget as it doesn't require [Intent](https://api.flutter.dev/flutter/widgets/Intent-class.html)s and [Actions](https://api.flutter.dev/flutter/widgets/Actions-class.html) widgets. Instead, it accepts a map of [ShortcutActivator](https://api.flutter.dev/flutter/widgets/ShortcutActivator-class.html)s to [VoidCallback](https://api.flutter.dev/flutter/dart-ui/VoidCallback.html)s.

Unlike [Shortcuts](https://api.flutter.dev/flutter/widgets/Shortcuts-class.html), this widget does not separate key bindings and their implementations.

class \_Shortcut1State extends State<Shortcut1> {

  @override

  Widget build(BuildContext context) {

    return CallbackShortcuts(

//  required Map<ShortcutActivator, VoidCallback> bindings:

// is a Map<Key, Value> -->  Map<ShortcutActivator, void Function()>

      bindings: <ShortcutActivator, VoidCallback>{

        const SingleActivator(LogicalKeyboardKey.arrowUp): () {

          setState(() => count = count + 1);

        },

        const SingleActivator(LogicalKeyboardKey.arrowDown): () {

          setState(() => count = count - 1);

        },

      },

      //  required Widget child:

      child: Focus( // phải có Focus để nhận sự kiện

        autofocus: true,

        child: Column(

          children: <Widget>[

            const Text('Press the up arrow key to add to the counter'),

            const Text('Press the down arrow key to subtract from the counter'),

            Text('count: $count'),

          ],

        ),

      ),

    );

  }

}

# Shortcuts

[Shortcuts](https://api.flutter.dev/flutter/widgets/Shortcuts/Shortcuts.html)({[Key](https://api.flutter.dev/flutter/foundation/Key-class.html)? key, required [Map](https://api.flutter.dev/flutter/dart-core/Map-class.html)<[ShortcutActivator](https://api.flutter.dev/flutter/widgets/ShortcutActivator-class.html), [Intent](https://api.flutter.dev/flutter/widgets/Intent-class.html)> shortcuts, required [Widget](https://api.flutter.dev/flutter/widgets/Widget-class.html) child, [String](https://api.flutter.dev/flutter/dart-core/String-class.html)? debugLabel})

Creates a const [Shortcuts](https://api.flutter.dev/flutter/widgets/Shortcuts-class.html) widget that owns the map of shortcuts and creates its own manager.

*const*

[Shortcuts.manager](https://api.flutter.dev/flutter/widgets/Shortcuts/Shortcuts.manager.html)({[Key](https://api.flutter.dev/flutter/foundation/Key-class.html)? key, required [ShortcutManager](https://api.flutter.dev/flutter/widgets/ShortcutManager-class.html) manager, required [Widget](https://api.flutter.dev/flutter/widgets/Widget-class.html) child, [String](https://api.flutter.dev/flutter/dart-core/String-class.html)? debugLabel})

Creates a const [Shortcuts](https://api.flutter.dev/flutter/widgets/Shortcuts-class.html) widget that uses the [manager](https://api.flutter.dev/flutter/widgets/Shortcuts/manager.html) to manage the map of shortcuts.

*const*

This widget establishes a [ShortcutManager](https://api.flutter.dev/flutter/widgets/ShortcutManager-class.html) to be used by its descendants when invoking an [Action](https://api.flutter.dev/flutter/widgets/Action-class.html) via a keyboard key combination that maps to an [Intent](https://api.flutter.dev/flutter/widgets/Intent-class.html).

This is similar to but more powerful than the [CallbackShortcuts](https://api.flutter.dev/flutter/widgets/CallbackShortcuts-class.html) widget. Unlike [CallbackShortcuts](https://api.flutter.dev/flutter/widgets/CallbackShortcuts-class.html), this widget separates key bindings and their implementations. This separation allows [Shortcuts](https://api.flutter.dev/flutter/widgets/Shortcuts-class.html) to have key bindings that adapt to the focused context. For example, the desired action for a deletion intent may be to delete a character in a text input, or to delete a file in a file menu.

/// Intent là một lớp trừu tượng được sử dụng để map tới một hành động cụ thể. Intent được sử dụng để kích hoạt hành động.

class IncrementIntent extends Intent {

  const IncrementIntent();

}

class DecrementIntent extends Intent {

  const DecrementIntent();

}

class Shortcut2 extends StatefulWidget {

  const Shortcut2({super.key});

  @override

  State<Shortcut2> createState() => \_Shortcut2State();

}

class \_Shortcut2State extends State<Shortcut2> {

  int count = 0;

  @override

  Widget build(BuildContext context) {

    return Shortcuts(

      //  required Map<ShortcutActivator, Intent> shortcuts:

      //  Map<ShortcutActivator, Intent> --> Map<SingleActivator, IncrementIntent>

      //Creates a const [Shortcuts] widget that owns the map of shortcuts and creates its own manager.

      shortcuts: const <ShortcutActivator, Intent>{

        SingleActivator(LogicalKeyboardKey.arrowUp): IncrementIntent(),

        SingleActivator(LogicalKeyboardKey.arrowDown): DecrementIntent(),

      },

      //required Widget child: - con của Shortcuts

      child: Actions(

        // required Map<Type, Action<Intent>> actions:

        //  ánh xạ từ loại hành động (Type) đến hành động (Action) cụ thể.

        // <Type, Action<Intent>> --> IncrementIntent: CallbackAction<IncrementIntent>

        // A map of [Intent] keys to [Action] objects that defines which actions this widget knows about.

        actions: <Type, Action<Intent>>{

          // CallbackAction<T extends Intent>

          IncrementIntent: CallbackAction<IncrementIntent>(

            // required void Function(T intent) onInvoke:

            onInvoke: (IncrementIntent intent) => setState(

              () {

                count = count + 1;

              },

            ),

          ),

          DecrementIntent: CallbackAction<DecrementIntent>(

            onInvoke: (DecrementIntent intent) => setState(() {

              count = count - 1;

            }),

          ),

        },

        // required Widget child: - con của Actions

        child: Focus( // phải có Focus để nhận sự kiện

          autofocus: true,

          child: Column(

            children: <Widget>[

              const Text('Shortcuts Example 2'),

              const Text('Add to the counter by pressing the up arrow key'),

              const Text(

                  'Subtract from the counter by pressing the down arrow key'),

              Text('count: $count'),

            ],

          ),

        ),

      ),

    );

  }

}

Example 2 – Truyền tham số qua Intent và Action

/// Model là một lớp được kế thừa từ ChangeNotifier. ChangeNotifier là một phần của Flutter được sử dụng để quản lý trạng thái và thông báo sự thay đổi trong trạng thái đó cho các thành phần giao diện người dùng.

class Model with ChangeNotifier {

  int count = 0;

  void incrementBy(int amount) {

    count += amount;

    notifyListeners();

  }

  void decrementBy(int amount) {

    count -= amount;

    notifyListeners();

  }

}

/// Intent là một lớp trừu tượng được sử dụng để map tới một hành động cụ thể. Intent được sử dụng để kích hoạt hành động.

class IncrementIntent1 extends Intent {

  const IncrementIntent1(this.amount);

  final int amount;

}

class DecrementIntent1 extends Intent {

  const DecrementIntent1(this.amount);

  final int amount;

}

/// Action là một lớp trừu tượng được sử dụng để thực thi một Intent cụ thể. Action được sử dụng để thực thi một Intent.

class IncrementAction extends Action<IncrementIntent1> {

  IncrementAction(this.model);

  final Model model;

  /// Thực thi một Intent cụ thể.

  //   void invoke(covariant T intent);

  @override

  // 'covariant' cho phép bạn sử dụng một kiểu con của IncrementIntent1 làm tham số cho hàm invoke, giúp linh hoạt hơn trong việc sử dụng các loại tham số.

  void invoke(covariant IncrementIntent1 intent) {

    model.incrementBy(intent.amount);

  }

}

class DecrementAction extends Action<DecrementIntent1> {

  DecrementAction(this.model);

  final Model model;

  @override

  /// Thực thi một Intent cụ thể.

  //   void invoke(covariant T intent);

  void invoke(covariant DecrementIntent1 intent) {

    model.decrementBy(intent.amount);

  }

}

class Shortcut3 extends StatefulWidget {

  const Shortcut3({super.key});

  @override

  State<Shortcut3> createState() => \_Shortcut3State();

}

class \_Shortcut3State extends State<Shortcut3> {

  Model model = Model();

  @override

  Widget build(BuildContext context) {

    return Shortcuts(

      shortcuts: const <ShortcutActivator, Intent>{

        // có thể sử dụng LogicalKeySet hoặc SingleActivator, khuyến khích dùng SingleActivator

        SingleActivator(LogicalKeyboardKey.arrowUp):

            IncrementIntent1(2), // truyền tham số qua Intent

        SingleActivator(LogicalKeyboardKey.arrowDown): DecrementIntent1(2),

      },

      child: Actions(

        actions: <Type, Action<Intent>>{

          IncrementIntent1: IncrementAction(model), // truyền tham số qua Action

          DecrementIntent1: DecrementAction(model),

        },

        child: Focus( // phải có Focus để nhận sự kiện

          autofocus: true,

          child: Column(

            children: <Widget>[

              const Text('Shortcuts Example 3'),

              const Text('Add to the counter by pressing the up arrow key'),

              const Text(

                  'Subtract from the counter by pressing the down arrow key'),

              // AnimatedBuilder là một widget được sử dụng để xây dựng các widget khác dựa trên các giá trị của một hoặc nhiều đối tượng Animation.

              // để cập nhật giao diện người dùng khi giá trị count thay đổi.

              AnimatedBuilder(

                animation: model,

                builder: (BuildContext context, Widget? child) {

                  return Text('count: ${model.count}');

                },

              ),

            ],

          ),

        ),

      ),

    );

  }

}

# CallbackAction

CallbackAction và các hành động (Action) khác trong hệ thống hành động của Flutter đều được sử dụng để xử lý sự kiện và hành động từ người dùng hoặc từ các lối tắt. Sự khác biệt chính giữa CallbackAction và các hành động khác là trong cách bạn định nghĩa xử lý cho hành động đó.

CallbackAction cho phép bạn định nghĩa một hàm callback (một hàm được gọi khi hành động được kích hoạt) trực tiếp bên trong nó.

Nó thường được sử dụng khi bạn muốn xử lý sự kiện hoặc hành động một cách đơn giản và trực tiếp, không cần tạo một lớp hành động riêng biệt.

Ví dụ sử dụng CallbackAction:

CallbackAction<MyIntent>(

onInvoke: (MyIntent intent) {

// Xử lý hành động ở đây

},

)

Hành động không sử dụng CallbackAction:

Khi bạn không sử dụng CallbackAction, bạn sẽ phải định nghĩa một lớp hành động riêng biệt để xử lý sự kiện. Lớp này sẽ kế thừa từ lớp Action và phải triển khai phương thức invoke để xử lý hành động.

Thường thì khi bạn cần xử lý sự kiện phức tạp hơn, hoặc muốn tái sử dụng mã xử lý cho nhiều hành động, bạn sẽ sử dụng các hành động riêng biệt thay vì CallbackAction.

Ví dụ không sử dụng CallbackAction:

class MyAction extends Action<MyIntent> {

@override

void invoke(covariant MyIntent intent) {

// Xử lý hành động ở đây

}

}

Tóm lại, sự khác biệt chính giữa việc sử dụng CallbackAction và không sử dụng nó nằm ở mức độ tổ chức mã xử lý hành động và cách bạn định nghĩa xử lý cho hành động đó. CallbackAction tiện lợi cho các xử lý đơn giản và ngắn gọn, trong khi các hành động riêng biệt thích hợp cho các tình huống phức tạp hơn hoặc cần sử dụng lại mã xử lý cho nhiều hành động.

# SingleActivator và LogicalKeySet

So sánh dưa trên cách triển khai  [Map](https://api.flutter.dev/flutter/dart-core/Map-class.html)<[ShortcutActivator](https://api.flutter.dev/flutter/widgets/ShortcutActivator-class.html), [Intent](https://api.flutter.dev/flutter/widgets/Intent-class.html)>

  shortcuts: <ShortcutActivator, Intent>{

        // map shortcut với LogicalKeySet

        LogicalKeySet(LogicalKeyboardKey.keyC, LogicalKeyboardKey.controlLeft):

            const IncrementIntent4(),

        // map shortcut với SingleActivator

        const SingleActivator(LogicalKeyboardKey.keyJ, control: true):

            const IncrementIntent4(),

      },

**SingleActivator** và **LogicalKeySet** là hai cách để định nghĩa các lối tắt phím trong Flutter. Dưới đây là sự khác biệt giữa chúng:

1. **SingleActivator**:
   * **SingleActivator** là một lớp đại diện cho một lối tắt phím đơn giản sử dụng một phím duy nhất.
   * Ví dụ: **const SingleActivator(LogicalKeyboardKey.arrowUp)** đại diện cho lối tắt sử dụng phím mũi tên lên.
   * Đây là lựa chọn thích hợp khi bạn muốn liên kết một lối tắt với một phím duy nhất.
2. **LogicalKeySet**:
   * **LogicalKeySet** là một lớp cho phép bạn định nghĩa một tập hợp các phím để tạo ra một lối tắt phức tạp.
   * Ví dụ: **LogicalKeySet(LogicalKeyboardKey.control, LogicalKeyboardKey.keyA)** đại diện cho lối tắt tổ hợp phím "Ctrl + A".
   * Đây là lựa chọn thích hợp khi bạn muốn xác định các lối tắt phức tạp sử dụng nhiều phím hoặc phím kết hợp.

Tóm lại, sự khác biệt chính giữa **SingleActivator** và **LogicalKeySet** nằm ở mức độ phức tạp của lối tắt. **SingleActivator** được sử dụng cho các lối tắt đơn giản với một phím duy nhất, trong khi **LogicalKeySet** cho phép bạn định nghĩa các lối tắt phức tạp hơn bằng cách sử dụng nhiều phím hoặc phím kết hợp.

It is not recommended to use [LogicalKeySet](https://api.flutter.dev/flutter/widgets/LogicalKeySet-class.html) for a common shortcut such as Delete or Ctrl+C, prefer [SingleActivator](https://api.flutter.dev/flutter/widgets/SingleActivator-class.html) when possible, whose behavior more closely resembles that of typical platforms.

# [ShortcutActivator](https://api.flutter.dev/flutter/widgets/ShortcutActivator-class.html) và [SingleActivator](https://api.flutter.dev/flutter/widgets/SingleActivator-class.html)

So sánh dựa trên khai báo [Map](https://api.flutter.dev/flutter/dart-core/Map-class.html)<[ShortcutActivator](https://api.flutter.dev/flutter/widgets/ShortcutActivator-class.html), [Intent](https://api.flutter.dev/flutter/widgets/Intent-class.html)>

// Khai báo ShortcutActivator - triển khai được cả LogicalKeySet và SingleActivator

shortcuts: <ShortcutActivator, Intent>{

        // map shortcut với LogicalKeySet

        LogicalKeySet(LogicalKeyboardKey.keyC, LogicalKeyboardKey.controlLeft):

            const IncrementIntent4(),

        // map shortcut với SingleActivator

        const SingleActivator(LogicalKeyboardKey.keyJ, control: true):

            const IncrementIntent4(),

      },

// Khai báo map shortcut với SingleActivator - chỉ triển khai được với SingleActivator

      bindings: <SingleActivator, VoidCallback>{

        const SingleActivator(LogicalKeyboardKey.arrowUp): () {

          setState(() => count = count + 1);

        },

        const SingleActivator(LogicalKeyboardKey.arrowDown): () {

          setState(() => count = count - 1);

        },

      },

// Khai báo map shortcut với LogicalKeySet - chỉ triển khai được với LogicalKeySet

   shortcuts: <LogicalKeySet, Intent>{

        LogicalKeySet(LogicalKeyboardKey.arrowUp): const IncrementIntent(),

        // const SingleActivator(LogicalKeyboardKey.arrowDown):

        //     const DecrementIntent()

      },

**ShortcutActivator** và **SingleActivator** là hai cách để kích hoạt (activate) lối tắt (shortcut) trong hệ thống lối tắt của Flutter. Dưới đây là sự khác biệt giữa chúng:

1. **SingleActivator**:
   * **SingleActivator** là một lớp đại diện cho một kích hoạt đơn giản sử dụng một phím duy nhất.
   * Ví dụ: **const SingleActivator(LogicalKeyboardKey.arrowUp)** đại diện cho việc kích hoạt lối tắt khi người dùng nhấn phím mũi tên lên.
   * **SingleActivator** thường được sử dụng cho các lối tắt đơn giản liên kết với một phím cụ thể.
2. **ShortcutActivator**:
   * **ShortcutActivator** là một lớp chung hơn và linh hoạt hơn. Nó cho phép bạn xác định một tập hợp các phím hoặc phím kết hợp để kích hoạt lối tắt.
   * Ví dụ: **const ShortcutActivator(<LogicalKeyboardKey>[LogicalKeyboardKey.control, LogicalKeyboardKey.keyA])** đại diện cho lối tắt tổ hợp phím "Ctrl + A".
   * **ShortcutActivator** thường được sử dụng khi bạn cần định nghĩa các lối tắt phức tạp hơn sử dụng nhiều phím hoặc phím kết hợp.

Tóm lại, sự khác biệt chính giữa **SingleActivator** và **ShortcutActivator** nằm ở mức độ phức tạp của kích hoạt lối tắt. **SingleActivator** được sử dụng cho các lối tắt đơn giản với một phím duy nhất, trong khi **ShortcutActivator** cho phép bạn định nghĩa các lối tắt phức tạp hơn bằng cách sử dụng nhiều phím hoặc phím kết hợp.

**Note:** ShortcutActivator is a replacement (as of Flutter 2.3.0) for LogicalKeySet. It allows for more flexible and correct activation of shortcuts. LogicalKeySet is a ShortcutActivator, of course, but there is also SingleActivator, which takes a single key and the optional modifiers to be pressed before the key, and CharacterActivator, which activates a shortcut based on the character produced by a key sequence, instead of the logical keys themselves. ShortcutActivator is also meant to be subclassed to allow for custom ways of activating shortcuts from key events.

Khi được sử dụng làm Phím tắt, LogicalKeySet sẽ kích hoạt ý định khi tất cả các phím được nhấn và không có phím nào khác được nhấn, ngoại trừ các phím bổ trợ được xem xét mà không xem xét các bên (ví dụ: Ctrl trái và Ctrl phải được xem là như nhau).

# Focus

[Focus](https://api.flutter.dev/flutter/widgets/Focus/Focus.html)({[Key](https://api.flutter.dev/flutter/foundation/Key-class.html)? key, required [Widget](https://api.flutter.dev/flutter/widgets/Widget-class.html) child, [FocusNode](https://api.flutter.dev/flutter/widgets/FocusNode-class.html)? focusNode, [FocusNode](https://api.flutter.dev/flutter/widgets/FocusNode-class.html)? parentNode, [bool](https://api.flutter.dev/flutter/dart-core/bool-class.html) autofocus = false, [ValueChanged](https://api.flutter.dev/flutter/foundation/ValueChanged.html)<[bool](https://api.flutter.dev/flutter/dart-core/bool-class.html)>? onFocusChange, [FocusOnKeyEventCallback](https://api.flutter.dev/flutter/widgets/FocusOnKeyEventCallback.html)? onKeyEvent, [FocusOnKeyCallback](https://api.flutter.dev/flutter/widgets/FocusOnKeyCallback.html)? onKey, [bool](https://api.flutter.dev/flutter/dart-core/bool-class.html)? canRequestFocus, [bool](https://api.flutter.dev/flutter/dart-core/bool-class.html)? skipTraversal, [bool](https://api.flutter.dev/flutter/dart-core/bool-class.html)? descendantsAreFocusable, [bool](https://api.flutter.dev/flutter/dart-core/bool-class.html)? descendantsAreTraversable, [bool](https://api.flutter.dev/flutter/dart-core/bool-class.html) includeSemantics = true, [String](https://api.flutter.dev/flutter/dart-core/String-class.html)? debugLabel})

Creates a widget that manages a [FocusNode](https://api.flutter.dev/flutter/widgets/FocusNode-class.html).

Một widget quản lý FocusNode để cho phép keyboard focus vào widget này và các widget con của nó.

Example 1

//Focus, dùng widget FocusScope và Focus

class Focus2 extends StatefulWidget {

  const Focus2({super.key});

  @override

  State<Focus2> createState() => \_Focus2State();

}

class \_Focus2State extends State<Focus2> {

  Color \_color = Colors.white;

  KeyEventResult \_handleKeyPress(FocusNode node, RawKeyEvent event) {

    if (event is RawKeyDownEvent) {

      debugPrint(

          'Focus node ${node.debugLabel} got key event: ${event.logicalKey}');

      if (event.logicalKey == LogicalKeyboardKey.keyR) {

        debugPrint('Changing color to red.');

        setState(() {

          \_color = Colors.red;

        });

        return KeyEventResult.handled;

      } else if (event.logicalKey == LogicalKeyboardKey.keyG) {

        debugPrint('Changing color to green.');

        setState(() {

          \_color = Colors.green;

        });

        return KeyEventResult.handled;

      } else if (event.logicalKey == LogicalKeyboardKey.keyB) {

        debugPrint('Changing color to blue.');

        setState(() {

          \_color = Colors.blue;

        });

        return KeyEventResult.handled;

      }

    }

    return KeyEventResult.ignored;

  }

  @override

  Widget build(BuildContext context) {

    final TextTheme textTheme = Theme.of(context).textTheme;

    // FocusScope: FocusScope là một widget cha cho Focus. Nó sẽ tạo ra một ngữ cảnh tập trung mới và sử dụng nó để tập trung vào FocusNode con đầu tiên của nó.

    // Widget gốc của ứng dụng là FocusScope, và nó định nghĩa phạm vi tập trung (focus scope) cho widget con bên trong. autofocus: true cho phép widget này tự động tập trung khi ứng dụng được khởi chạy.

    return FocusScope(

      debugLabel: 'Scope', // debugLabel: nhãn để gỡ lỗi

      autofocus: true,

      child: DefaultTextStyle(

        style: textTheme.headlineMedium!,

        child: Focus(

          //

          onKey:

              \_handleKeyPress, // onKey: sự kiện phím - Đây là callback được gọi khi người dùng nhấn các phím trên bàn phím

          debugLabel: 'Button',

          child: Builder(

            // Widget này sử dụng Builder để tạo ra một BuildContext riêng cho widget con bên trong nó. Điều này là cần thiết vì FocusScope không phải là một widget con của MaterialApp, vì vậy nó không có một BuildContext để sử dụng.

            builder: (BuildContext context) {

              final FocusNode focusNode = Focus.of(

                  context); // Focus.of(context) - trả về focus node của widget cha gần nhất

              final bool hasFocus =

                  focusNode.hasFocus; // lấy giá trị hasFocus của focus node

              return GestureDetector(

                // GestureDetector, cho phép xử lý các sự kiện như nhấn (tap) trên nút.

                onTap: () { // click vào: switch focus true/false

                  if (hasFocus) {

                    focusNode.unfocus();

                  } else {

                    focusNode.requestFocus();

                  }

                },

                child: Center(

                  child: Container(

                    width: 400,

                    height: 100,

                    alignment: Alignment.center,

                    color: hasFocus ? \_color : Colors.white,

                    child: Text(hasFocus

                        ? "I'm in color! Press R,G,B! Click to unfocus."

                        : 'Click to focus'),

                  ),

                ),

              );

            },

          ),

        ),

      ),

    );

  }

}

Example 2

// Chỉ dùng widget Focus

class Focus3 extends StatelessWidget {

  const Focus3({super.key});

  @override

  Widget build(BuildContext context) {

    return ListView.builder(

      itemBuilder: (BuildContext context, int index) => FocusableText(

        'Item $index',

        autofocus: index == 10,

      ),

      itemCount: 50,

    );

  }

}

class FocusableText extends StatelessWidget {

  const FocusableText(

    this.data, {

    super.key,

    required this.autofocus,

  });

  /// The string to display as the text for this widget.

  final String data;

  /// Whether or not to focus this widget initially if nothing else is focused.

  final bool autofocus;

  @override

  Widget build(BuildContext context) {

    return Focus(

      autofocus: autofocus, // Must not be null. Defaults to false.

      child: Builder(builder: (BuildContext context) {

        // The contents of this Builder are being made focusable. It is inside

        // of a Builder because the builder provides the correct context

        // variable for Focus.of() to be able to find the Focus widget that is

        // the Builder's parent. Without the builder, the context variable used

        // would be the one given the FocusableText build function, and that

        // would start looking for a Focus widget ancestor of the FocusableText

        // instead of finding the one inside of its build function.

        return Container(

          padding: const EdgeInsets.all(8.0),

          // Change the color based on whether or not this Container has focus.

          // hasPrimaryFocus là thuộc tính của đối tượng Focus, được sử dụng để kiểm tra xem một phần tử (widget) có được focus hay không.

          color: Focus.of(context).hasPrimaryFocus

              ? const Color.fromARGB(31, 245, 98, 6)

              : null,

          child: Text(data),

        );

      }),

    );

  }

}

Example 3

 // Dùng FocusNode, không dùng widget Focus

class Focus1 extends StatelessWidget {

  const Focus1({super.key});

  @override

  Widget build(BuildContext context) {

    final TextTheme textTheme = Theme.of(context).textTheme;

    return DefaultTextStyle(

      style: textTheme.headlineMedium!,

      child: const ColorfulButton(),

    );

  }

}

class ColorfulButton extends StatefulWidget {

  const ColorfulButton({super.key});

  @override

  State<ColorfulButton> createState() => \_ColorfulButtonState();

}

class \_ColorfulButtonState extends State<ColorfulButton> {

  late FocusNode \_node; // khai báo focus node

  bool \_focused = false; // khai báo biến kiểm tra trạng thái focus

  late FocusAttachment

      \_nodeAttachment; // khai báo focus attachment - sử dụng để gắn \_node với BuildContext.

  Color \_color = Colors.white;

  @override

  // khoi tao gia tri ban dau, run truoc build function

  void initState() {

    //

    super.initState();

    \_node = FocusNode(debugLabel: 'Button'); // khởi tạo focus node

    \_node.addListener(\_handleFocusChange); // add listener cho focus node

    \_nodeAttachment = \_node.attach(context,

        onKey: \_handleKeyPress); // attach focus node với context

  }

// khi focus thay đổi thì cập nhật trạng thái của \_focused

  void \_handleFocusChange() {

    if (\_node.hasFocus != \_focused) {

      // kiểm tra trạng thái focus của node

      setState(() {

        \_focused = \_node.hasFocus;

      });

    }

  }

//  Hàm này xử lý sự kiện phím. Nếu người dùng nhấn các phím R, G, hoặc B, màu sắc của nút sẽ thay đổi tương ứng và sự kiện phím được xử lý và trả về KeyEventResult.handled.

  KeyEventResult \_handleKeyPress(FocusNode node, RawKeyEvent event) {

    // RawKeyEvent: sự kiện phím

    if (event is RawKeyDownEvent) {

      // RawKeyDownEvent: sự kiện khi nhấn phím

      debugPrint(

          'Focus node ${node.debugLabel} got key event: ${event.logicalKey}'); // ${event.logicalKey} - LogicalKeyboardKey#1b3a4(keyId: "0x00000064", keyLabel: "D", debugName: "Key D")

      if (event.logicalKey == LogicalKeyboardKey.keyR) {

        debugPrint('Changing color to red.');

        setState(() {

          \_color = Colors.red;

        });

        return KeyEventResult

            .handled; // KeyEventResult.handled: sự kiện phím đã được xử lý

      } else if (event.logicalKey == LogicalKeyboardKey.keyG) {

        debugPrint('Changing color to green.');

        setState(() {

          \_color = Colors.green;

        });

        return KeyEventResult.handled;

      } else if (event.logicalKey == LogicalKeyboardKey.keyB) {

        debugPrint('Changing color to blue.');

        setState(() {

          \_color = Colors.blue;

        });

        return KeyEventResult.handled;

      }

    }

    return KeyEventResult

        .ignored; // KeyEventResult.ignored: sự kiện phím không được xử lý

  }

  @override

  // khi widget bị xóa thì dispose focus node

  void dispose() {

    \_node.removeListener(\_handleFocusChange); // remove listener

    // The attachment will automatically be detached in dispose().

    \_node.dispose(); // dispose focus node

    super.dispose(); // dispose widget

  }

  @override

  Widget build(BuildContext context) {

    // \_nodeAttachment. reparent() được gọi để đảm bảo rằng \_node được gắn lại vào ngữ cảnh (context) hiện tại. Điều này quan trọng để quản lý tập trung và sự kiện phím.

    \_nodeAttachment.reparent();

    return GestureDetector(

      // GestureDetector, cho phép xử lý các sự kiện như nhấn (tap) trên nút.

      onTap: () {

        // khi nhấn vào nút

        if (\_focused) {

          // switch focus true/false

          \_node.unfocus(); // unfocus node

        } else {

          \_node.requestFocus(); // focus node

        }

      },

      child: Center(

        child: Container(

          width: 400,

          height: 100,

          color: \_focused ? \_color : Colors.white,

          alignment: Alignment.center,

          child:

              Text(\_focused ? "I'm in color! Press R,G,B!" : 'Press to focus'),

        ),

      ),

    );

  }

}

Example 4

// dùng List<FocusNode> để quản lý focus node

class Focus4 extends StatefulWidget {

  const Focus4({super.key});

  @override

  State<Focus4> createState() => \_Focus4State();

}

class \_Focus4State extends State<Focus4> {

  int focusedChild = 0; // index của child đang được focus

  List<Widget> children = <Widget>[]; // list các child

  List<FocusNode> childFocusNodes = <FocusNode>[]; // list các focus node

  @override

  void initState() {

    super.initState();

    // Add the first child.

    \_addChild();

  }

  void \_addChild() {

    // Calling requestFocus here creates a deferred request for focus, since the

    // node is not yet part of the focus tree.

    childFocusNodes.add(FocusNode(

        debugLabel:

            'Child ${children.length}')); // add focus node mới vào list. lần đầu tiên = 0 do list widget rỗng

// Set vị trí focus trong childFocusNodes

    // ignore: prefer\_is\_empty

    if (children.length > 1) {

      // Calling requestFocus here creates a deferred request for focus, since

      // the node is not yet part of the focus tree.

      childFocusNodes

          .elementAt(1)

          .requestFocus(); // phải set focusNode: childFocusNodes.last (ở dưới) thì mới có tác dụng

    }

    children.add(Padding(

      // add widget vào list

      padding: const EdgeInsets.all(2.0),

      child: ActionChip(

        autofocus: true,

        focusNode: childFocusNodes

            .last, // để set vị trí note focus được thì phải set focusNode: childFocusNodes.last (childFocusNodes.first thì ko tác dụng)

        label: Text('CHILD ${children.length}'),

        onPressed: () {},

      ),

    ));

  }

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      body: Center(

        child: Wrap(

          children: children,

        ),

      ),

      floatingActionButton: FloatingActionButton(

        onPressed: () {

          setState(() {

            focusedChild = children.length;

            debugPrint(childFocusNodes.last.toString());

            debugPrint('Adding child $focusedChild');

            \_addChild();

          });

        },

        child: const Icon(Icons.add),

      ),

    );

  }

}

# MenuAcceleratorLabel

A widget that draws the label text for a menu item (typically a [MenuItemButton](https://api.flutter.dev/flutter/material/MenuItemButton-class.html) or [SubmenuButton](https://api.flutter.dev/flutter/material/SubmenuButton-class.html)) and renders its child with information about the currently active keyboard accelerator.

(Đánh dấu gạch chân ký tự shortcut cho menu item khi bấm phím Alt)

To use the accelerators, press the Alt key to see which letters are underlined in the menu bar, and then press the appropriate letter. Accelerators are not supported on macOS or iOS since those platforms don't support them natively, so this demo will only show a regular Material menu bar on those platforms.