# Flutter App Implementation Guidelines for E2E Testing

Version: July 31, 2023

MagicPod

# Table of Contents

# 1. Introduction

This guide is an application implementation guide for enabling E2E testing of iOS and Android apps created using Flutter.

## 1-1. Scope of this Guide

The methods introduced in this guide are applicable to Appium and all E2E automated testing tools that internally utilize Appium.

## 1-2. Versions Validated by this Guide

The following versions of libraries and tools were used to validate the content of this guide.

- Flutter version 3.3.0
- Appium version 2.0.0-beta.71

For E2E automated testing tools that internally utilize Appium, MagicPod (https://magicpod.com/en/) is used for behavior verification.

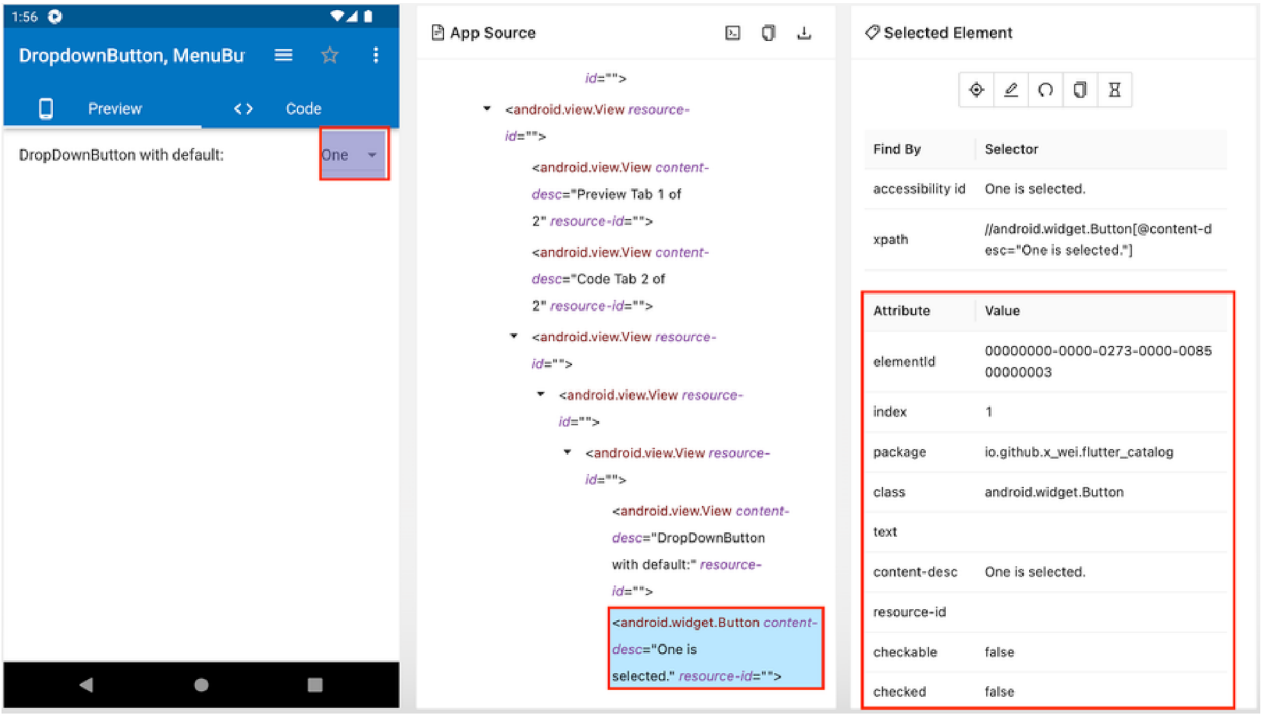
# 2. Issues with E2E Test Automation for Flutter Apps

A frequent issue when automating E2E testing for Flutter apps using Appium is the incorrect recognition of UI elements within the Flutter app. This chapter provides a detailed explanation of this problem.

First, we explain how Appium recognizes UI elements.

Similar to how Flutter has a SemanticsTree, Appium has a page source. The page source is a data structure that represents the UI elements on the screen in a tree structure. When creating an E2E test in Appium, it is necessary to locate the desired UI element in this page source and verify UI element information. One method for verifying is with the Appium Inspector (https://inspector.appiumpro.com/). Using Appium Inspector allows you to identify where each UI element is located on an app screen, as well as its attribute values, and more.

In the Appium Inspector's screen (shown below), the left side shows the location of the 'One' button on the app screen, the middle indicates the location of the 'One' button in the page source, and the right side displays the attribute values associated with the 'One' button.

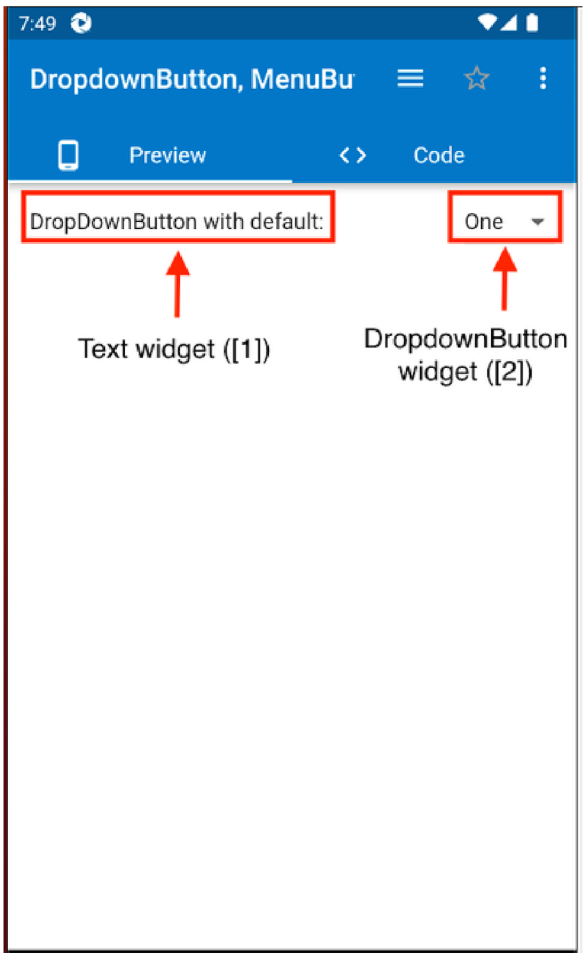


There are two main patterns of cases where Appium fails to recognize UI elements in a Flutter app.

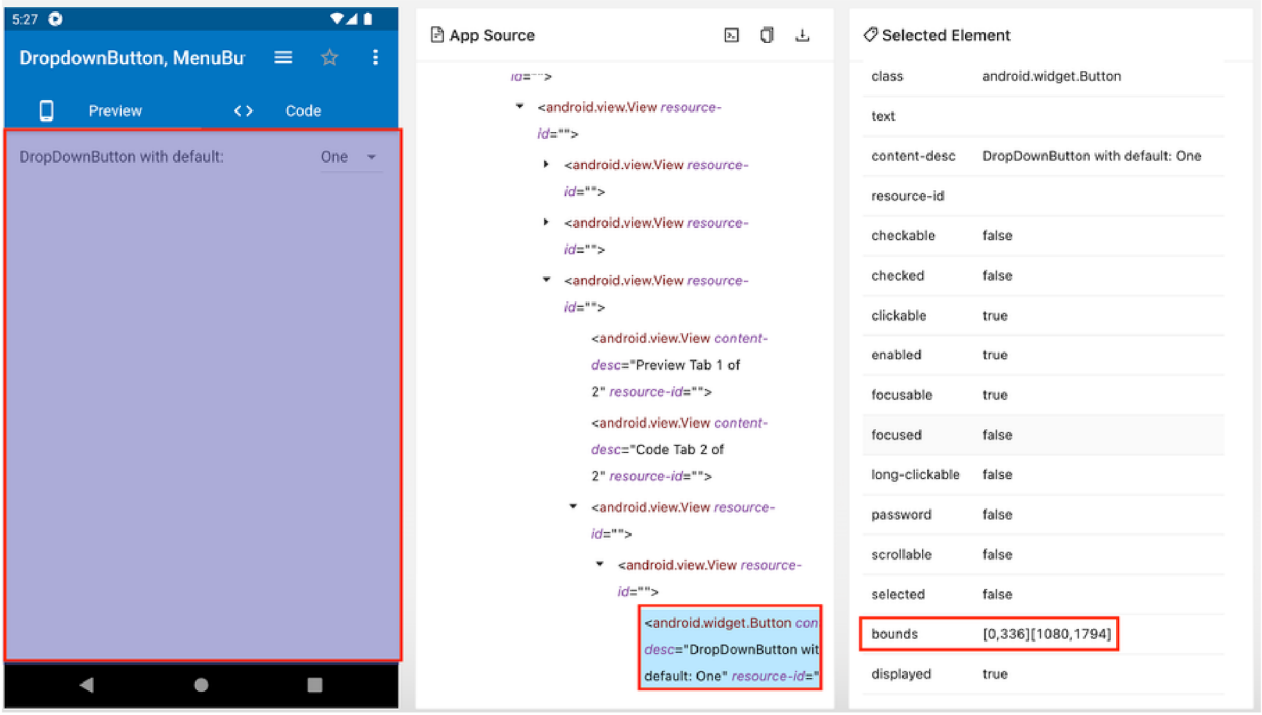
## 2-1. Multiple Widgets are Recognized as a Single Block of UI Element

The first issue is "multiple widgets are recognized as a single block of elements, rather than as separate elements". Let's explain this in detail using the following Flutter app screen.

On this screen, we are using a ListTile, which has the Text widget ([1]) and the DropdownButton widget ([2]) as its 'title' and 'trailing' properties, respectively. Moreover, the Text widget ([1]) has "DropDownButton with default:" as its text, and the DropdownButton widget ([2]) has the options "One", "Two", "Three", and "Four".



Now, using the Appium Inspector, let's check how each widget is recognized as a UI element. The results are illustrated in the diagram below.




Looking at the results, the area of the Text widget and the DropdownButton widget seems to be recognized as one single block, and furthermore, the size is recognized as a much larger area than it actually is. This causes the following problems during test automation.

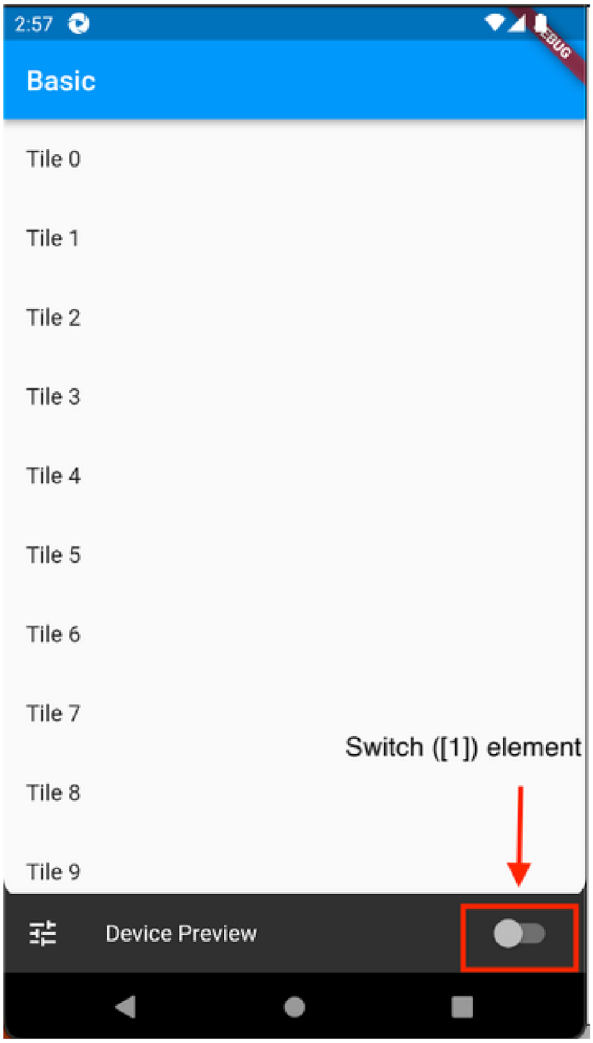
1. It is not possible to individually retrieve the display text values of the Text widget and the DropdownButton widget.
2. The position of the UI element differs significantly from the actual position of the DropdownButton, making it impossible to tap the DropdownButton using a click command. Appium's click command clicks the center position of the target UI element, so if the position of the UI element differs from the actual position, the tap will fail. (From the value of the bounds attribute, the top-left coordinate of this UI element is recognized as (x, y) = (0, 336), and its size as 1080*1458. However, since the screen size is 1080*1794, it is clear that the recognized position of this UI element is significantly different from its actual position.)

As far as we've checked, similar issues have occurred in at least the following situations:

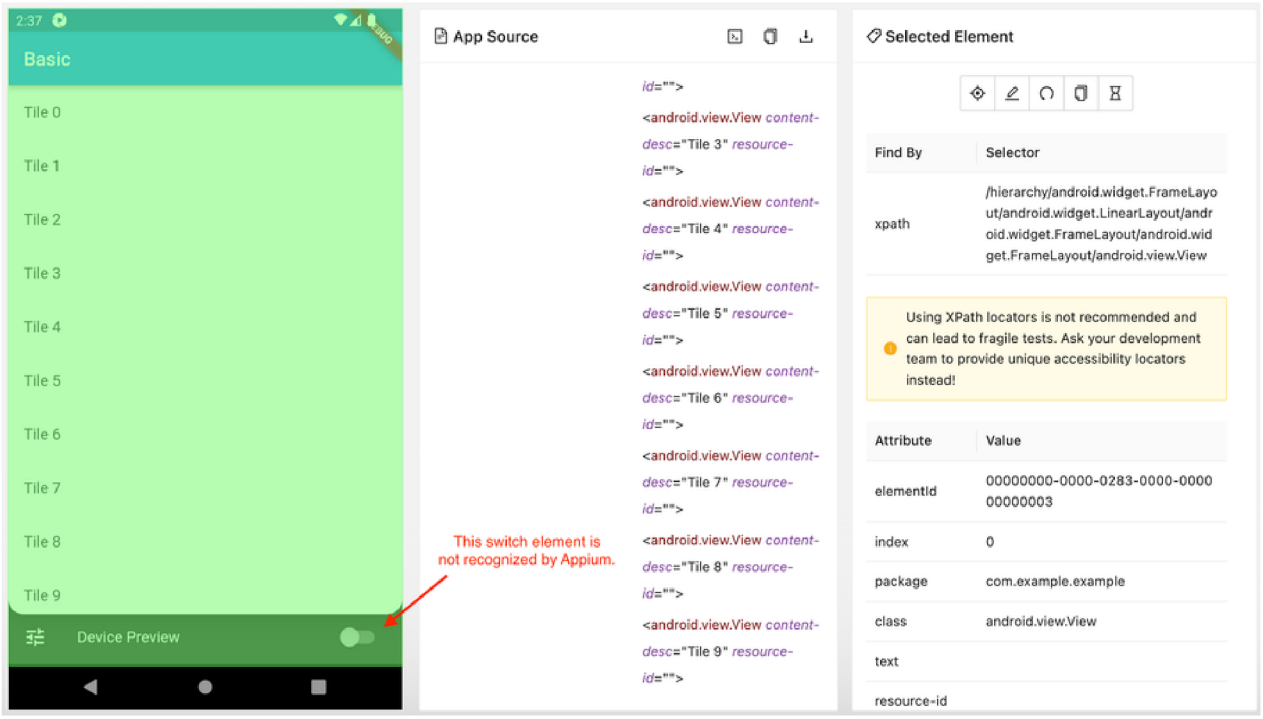
- When using a Container widget in the children property of a Stack widget.
- When using TextField, Text, HighlightView, Center widgets, etc., in the children property of a Column widget.

## 2-2. UI Elements Not Recognized at All

The second issue is "UI elements corresponding to screen items are not recognized at all". This time, we will explain in detail using the following Flutter App screen. Using the Appium Inspector, let's examine how the switch element ([1]) located at the bottom right of the screen is recognized.



The recognition results are as follows.




It's a bit hard to see, but the switch at the bottom right of the screen is not recognized as a UI element at all. If this happens, it becomes impossible to tap on the element using a click command.

# 3. Solutions

In the previous chapter, we described two patterns where UI elements in Flutter apps are not properly recognized. In this chapter, we will introduce solutions to these issues.

## 3-1. Upgrade Flutter to version 3.3.0 or later

The first step is to upgrade Flutter to version 3.3.0 or later, as newer versions of Flutter have greatly improved issues with UI element recognition. According to https://github.com/flutter/flutter/issues/18060#issuecomment-1251740879, the enhancements we discuss in the following sections appear to be effective for iOS devices running Flutter 3.0.0 or later, and even for other devices using versions prior to Flutter 3.0.0. However, based on this guide, which has been thoroughly validated, we recommend Flutter 3.3.0 or above.

## 3-2. Review the Implementation of UI Elements in the App

Next, detailed solutions regarding the two issues identified in the previous chapter are introduced. As a solution, it's essential to review the implementation of UI elements that aren't properly recognized.

### 3-2-1. Isolate Widgets as SemanticsNode for Desired Testing Units

First, as a solution to the issue in 2-1 where "multiple widgets are recognized as a single block of UI element", we explain how to isolate widgets as SemanticsNode for the desired testing units. The screen from earlier is presented again, where the Text widget and DropdownButton widget areas were recognized as a single block of UI element.

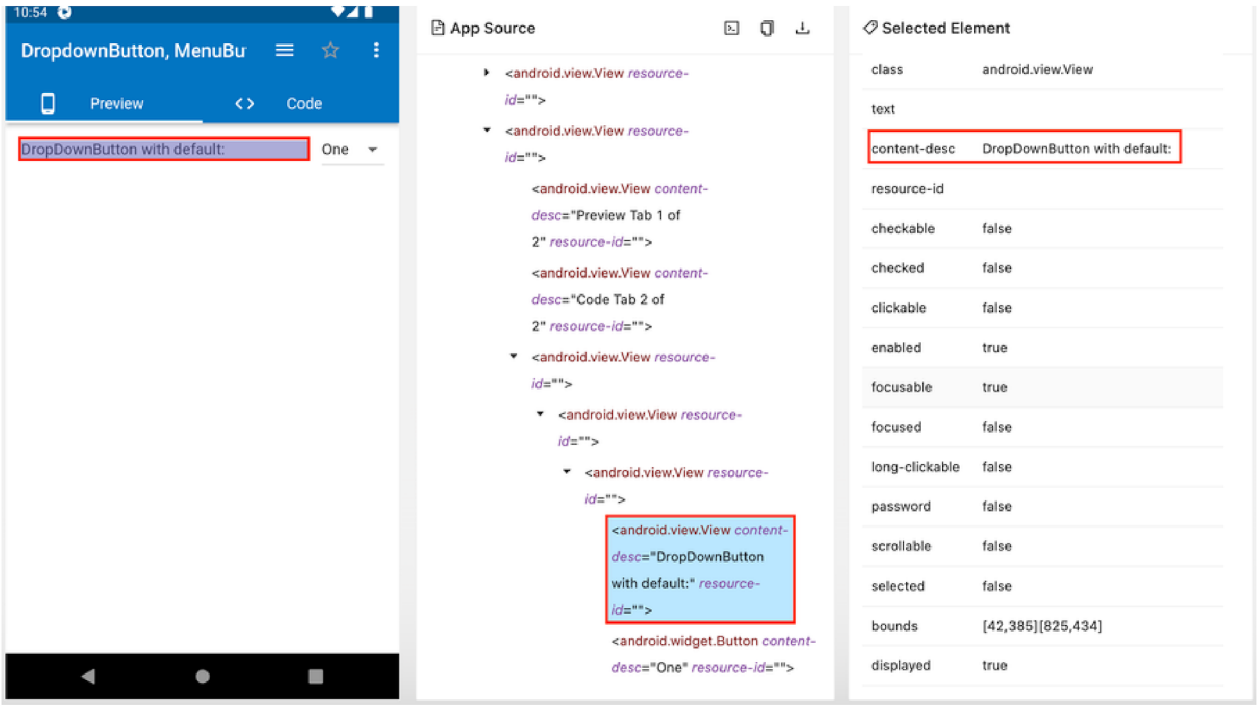

This screen is implemented using a combination of a ListTile widget, a Text widget, and a DropdownButton widget.

```dart
@override
Widget build(BuildContext context) {
  return Column(
    children: <Widget>[
      ListTile(
        title: const Text('DropDownButton with default:'),
        trailing: DropdownButton<String>(
          // Must be one of items.value.
          value: _btn1SelectedVal,
          onChanged: (String? newValue) {
            if (newValue != null) {
              setState(() => _btn1SelectedVal = newValue);
            }
          },
          items: this._dropDownMenuItems,
        ), // DropdownButton
      ), // ListTile
    ], // <Widget>[]
  ); // Column
}
```

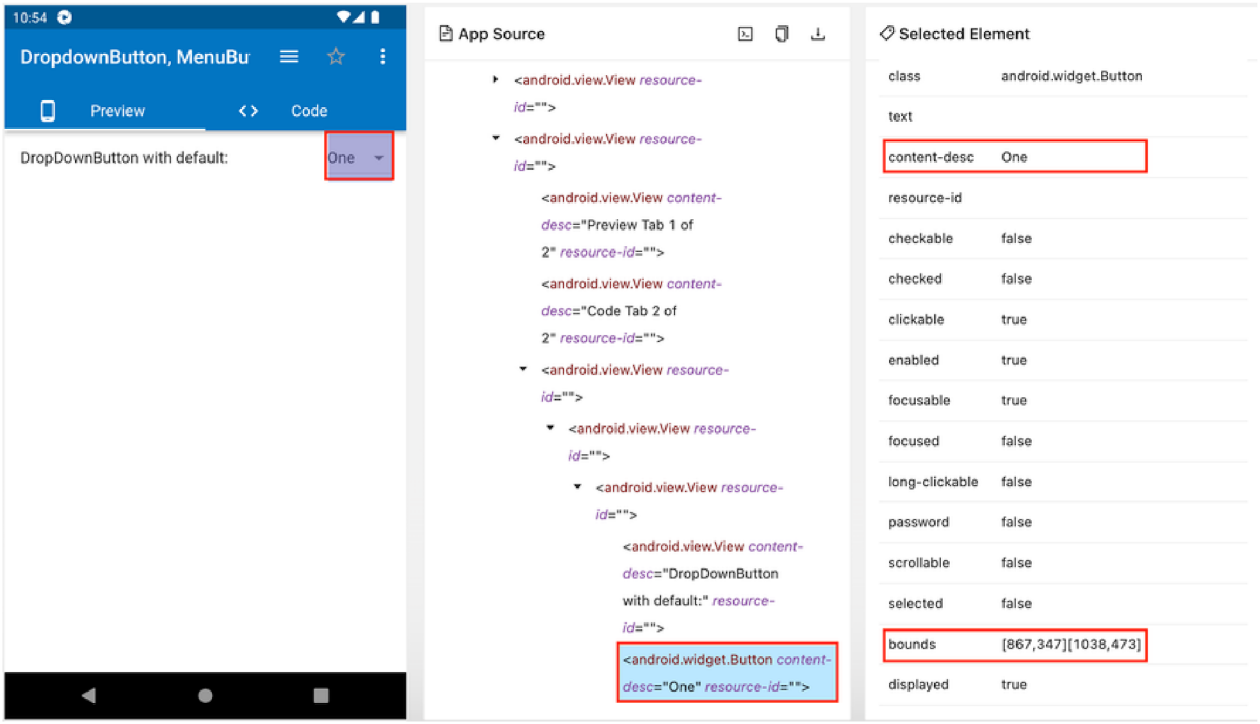
Isolate the Text widget and DropdownButton widget as SemanticsNode. To do this, wrap each widget with a Semantics widget with the container property set to true.

```dart
@override
Widget build(BuildContext context) {
  return Column(
    children: <Widget>[
      ListTile(
        title: Semantics(
          container: true,
          child: const Text('DropDownButton with default:')
        ), // Semantics
        trailing: Semantics(
          container: true,
          child: DropdownButton<String>(
            // Must be one of items.value.
            value: _btn1SelectedVal,
            onChanged: (String? newValue) {
              if (newValue != null) {
                setState(() => _btn1SelectedVal = newValue);
              }
            },
            items: this._dropDownMenuItems,
          ), // DropdownButton
        ), // Semantics
      ), // ListTile
    ], // <Widget>[]
  ); // Column
}
```

The Text widget is then recognized as an independent single UI element.



Similarly, the Drop-down Button widget has also been recognized as an independent UI element.

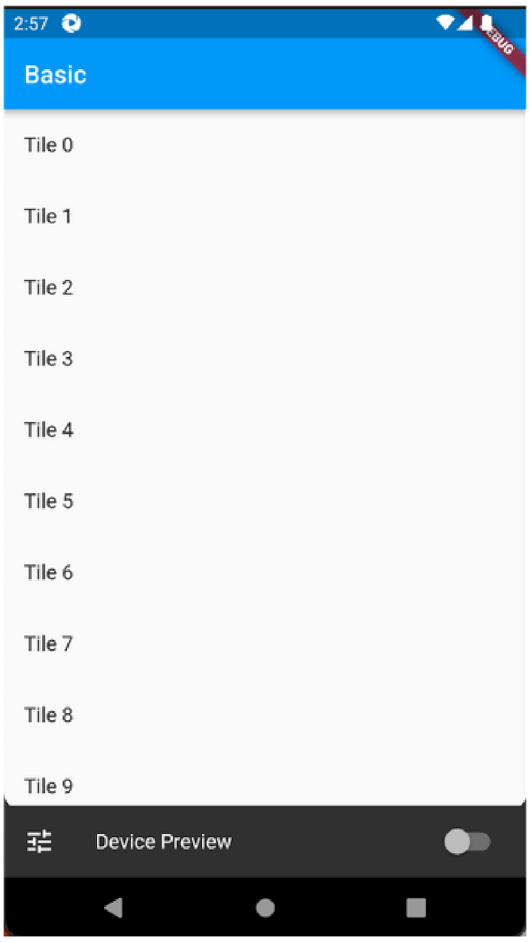



Since each widget was recognized as an independent UI element, the position of each UI element has been aligned with reality. Therefore, the DropdownButton widget can now be operated by Appium's click command. Moreover, from an accessibility standpoint, this enhancement ensures more precise button location detection in Android's 'TalkBack' and iOS's 'VoiceOver'.

In this instance, we isolated the Text widget and the DropdownButton widget as individual UI elements. However, there's no issue with isolating the ListTile widget as individual UI element as well.
In such cases, it is better to use widgets like GestureDetector (https://api.flutter.dev/flutter/widgets/GestureDetector-class.html) to modify the ListTile widget to detect tap events.

### 3-2-2. Properly Setting the Z-order of a Stack Widget

Next, as a solution to the issue in 2-2 where "UI Elements Not Recognized at All", we explain how to properly set the Z-order of a Stack widget. The screen from earlier is presented again, where the switch element located at the bottom right corner was not recognized.



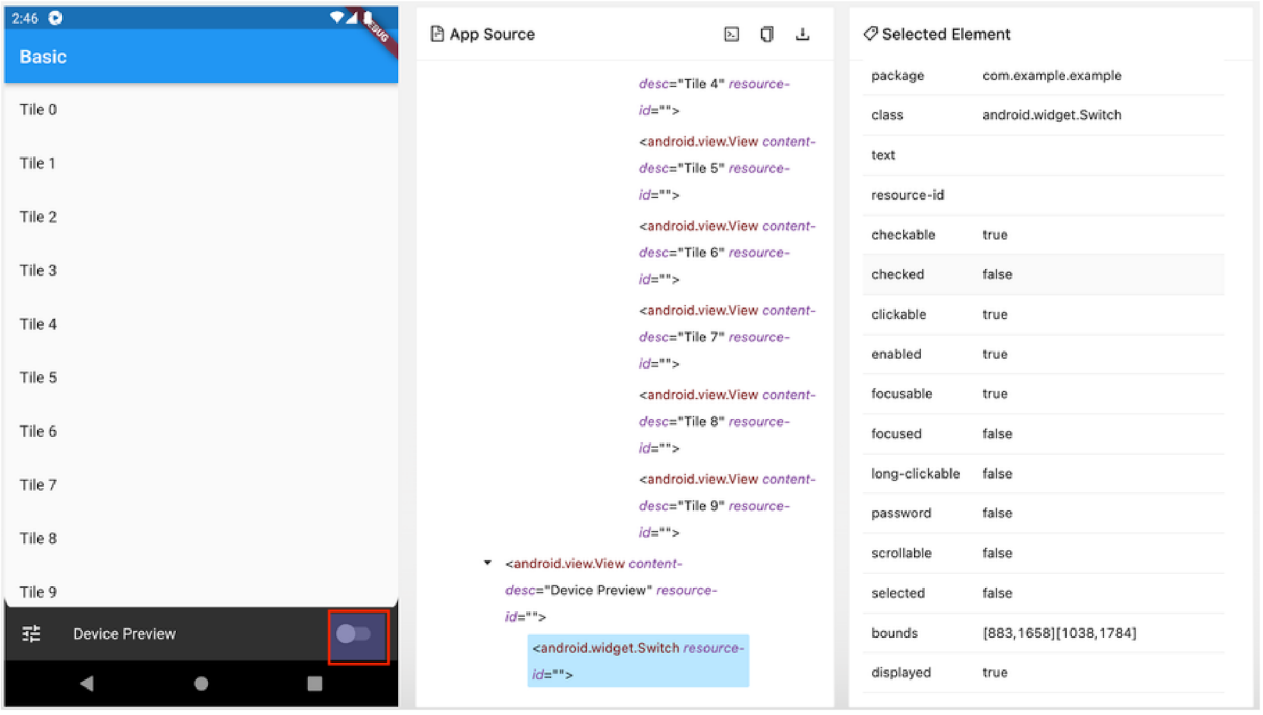

This screen is implemented using a Stack widget. It may look complicated, but essentially, it's just using a Stack widget and stacking the Positioned widget and AnimatedPositioned widget in sequence.

```
554    return Stack(
555      children: <Widget>[
556        Positioned(
557          key: const Key('Small'),
558          bottom: 0,
559          right: 0,
560          left: 0,
561          child: DevicePreviewSmallLayout(
562            slivers: widget.tools,
563            maxMenuHeight: constraints.maxHeight * 0.5,
564            scaffoldKey: scaffoldKey,
565            onMenuVisibleChanged: (isVisible) => setState(() {
566              _isToolPanelPopOverOpen = isVisible;
567            }),
568          ), // DevicePreviewSmallLayout
569        ), // Positioned
570        AnimatedPositioned(
571          key: const Key('preview'),
572          duration: const Duration(milliseconds: 200),
573          left: 0,
574          right: rightPanelOffset,
575          top: 0,
576          bottom: bottomPanelOffset,
577          child:  ClipRRect(
578            borderRadius: borderRadius,
579            child: Builder(
580              builder: _buildPreview
581            ), // Builder
582          ), // ClipRRect
583        ), // AnimatedPositioned
584      ], // <Widget>[]
585    ); // Stack
```

In a Stack widget, the widgets positioned further behind in the children array are given a higher Z-order, similar to CSS. This means they'll appear closer to the front of the screen. If they are not arranged in the correct order, Appium may not be able to recognize it as a UI element, even if they appear fine on screen. Therefore, swap the positions of the Positioned widget and the AnimatedPositioned widget.

```dart
return Stack(
  children: <Widget>[
    AnimatedPositioned(
      key: const Key('preview'),
      duration: const Duration(milliseconds: 200),
      left: 0,
      right: rightPanelOffset,
      top: 0,
      bottom: bottomPanelOffset,
      child:  ClipRRect(
        borderRadius: borderRadius,
        child: Builder(
          builder: _buildPreview
        ), // Builder
      ), // ClipRRect
    ), // AnimatedPositioned
    Positioned(
      key: const Key('Small'),
      bottom: 0,
      right: 0,
      left: 0,
      child: DevicePreviewSmallLayout(
        slivers: widget.tools,
        maxMenuHeight: constraints.maxHeight * 0.5,
        scaffoldKey: scaffoldKey,
        onMenuVisibleChanged: (isVisible) => setState(() {
          _isToolPanelPopOverOpen = isVisible;
        }),
      ), // DevicePreviewSmallLayout
    ), // Positioned
  ], // <Widget>[]
); // Stack
```

By doing this, the switch next to the label "Device Preview" could be recognized.



Thus, by appropriately re-setting the Z-order between widgets, previously unrecognizable elements can be recognized.

# 4. Implementation Tutorial

This chapter outlines the specific steps to make a Flutter app E2E testable.

## 4-1. Upgrade Flutter

First, upgrade the version of Flutter you are using to version 3.3.0 or later.

## 4-2. List UI Elements that Aren't Recognized Properly

Next, launch the Flutter app that is the subject of the test using either Appium Inspector, uiautomatorviewer, or MagicPod. Open the screens that will be used in the E2E test one by one. Check each UI element used in the test if it’s correctly recognized. List the elements that are not recognized correctly. Appium Inspector is recommended. Since the way elements are recognized is basically the same for both iOS and Android, it's sufficient to do the listing process on just one platform.

For specific verification procedures, please refer to Chapter 2: "Issues with E2E Test Automation for Flutter Apps.".

- When using Appium Inspector, please refer to https://support.magic-pod.com/hc/en-us/articles/4408926683033#sec3_2 for setup and usage instructions.
- When using uiautomatorviewer, please refer to https://support.magic-pod.com/hc/en-us/articles/4408926683033#sec3_1 for usage instructions. In this case, it’s necessary to use Android application.
- When using MagicPod, there might be issues on the MagicPod side causing target UI elements to be obscured by other larger UI elements. Following the procedures in https://support.magic-pod.com/hc/en-us/articles/4409255879961 If you can find the element by hiding the larger elements on the top, then there is no issue with the Flutter app.

This task can be carried out by non-engineers with test automation experience.

## 4-3. App Code Correction

Once completed with the listing process, review the app code for each element that has an issue. Refer to section 3-2 "Review the Implementation of UI Elements in the App", and correct the app code for each element so that the element is recognized properly.

# 5. Impact on Accessibility Information

In section 3-2-1, "Isolate Widgets as SemanticsNode for Desired Testing Units," elements that are isolated as SemanticsNode will be read aloud by the screen reader. In most cases, the UI elements that you want to manipulate and retrieve in E2E automated testing are the same elements you would want the screen reader to vocalize. Normally, this shouldn't cause any problems. However, when reviewing the implementation of UI elements, please be mindful of the impact on accessibility, such as regarding screen reader output.

# Flutter App Implementation Guidelines for E2E Testing

Version: July 31, 2023

**MagicPod Inc.**
〒103-0015
4F The Shore Nihonbashi Kayabacho,
1-2 Nihonbashi Hakozaki-cho, Chuo-ku, Tokyo

https://magicpod.com/en/contact/