

[Get started](#)

[Samples & tutorials](#)

[Development](#)

- ▶ [User interface](#)
- ▶ [Data & backend](#)
- ▶ [Accessibility & internationalization](#)
- ▶ [Platform integration](#)
- ▶ [Packages & plugins](#)
- ▶ [Add Flutter to existing app](#)
- ▼ [Tools & techniques](#)
 - [Android Studio & IntelliJ](#)
 - [Visual Studio Code](#)
 - ▼ [DevTools](#)
 - [Overview](#)
 - [Install from Android Studio & IntelliJ](#)
 - [Install from VS Code](#)
 - [Install from command line](#)
 - [Flutter inspector](#)
 - [Timeline view](#)
 - [Memory view](#)
 - [Performance view](#)
 - [Debugger](#)
 - [Logging view](#)
 - ▶ [Flutter SDK](#)
 - [Hot reload](#)
 - [Code formatting](#)

[Testing & debugging](#)

[Performance & optimization](#)

[Deployment](#)

[Resources](#)

[Reference](#)

- [Widget index](#)
- [API reference](#)
- [Package site](#)

Using the Flutter inspector

[Docs](#) > [Development](#) > [Tools](#) > [DevTools](#) > [Using the Flutter inspector](#)

Contents

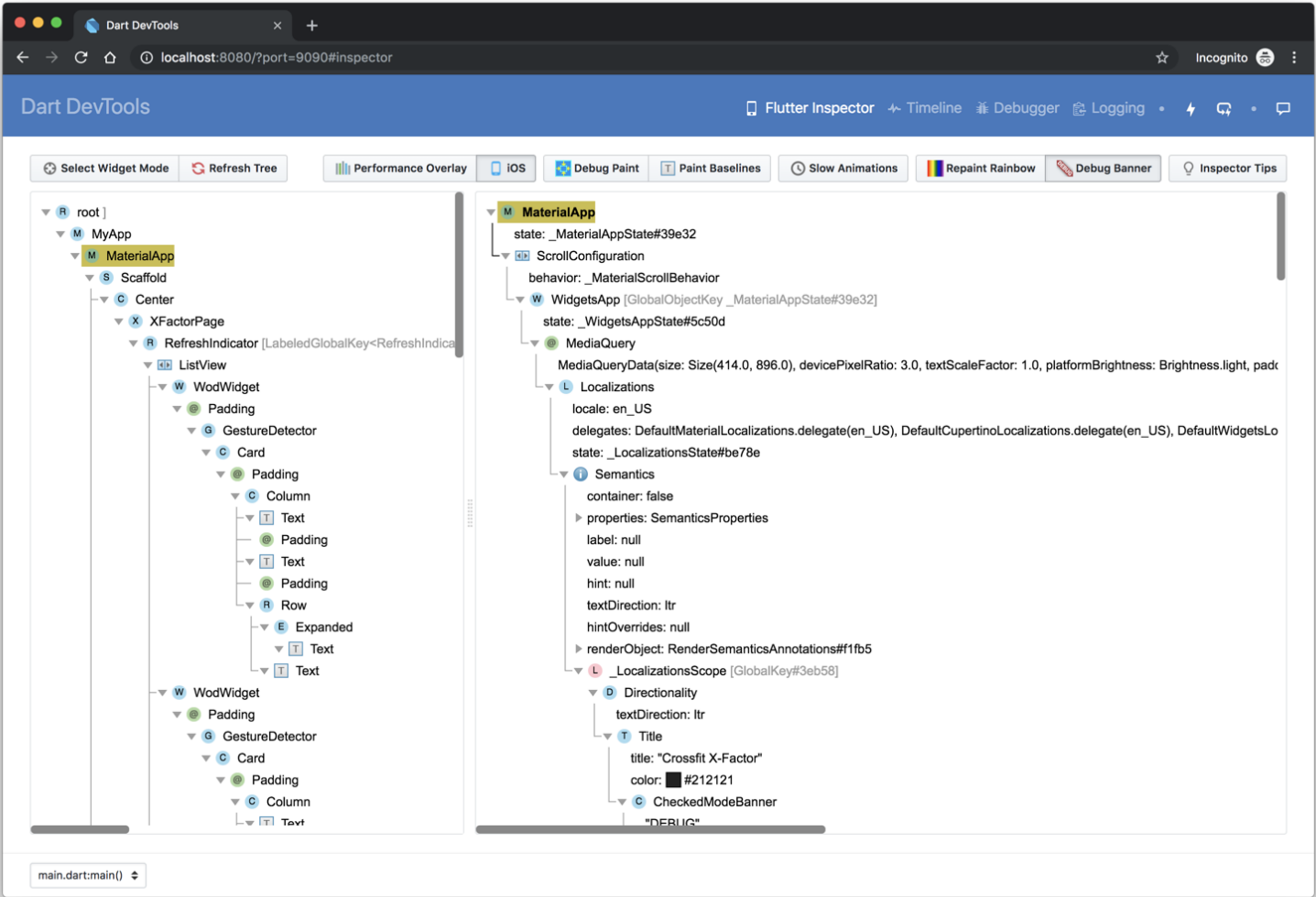
- [What is it?](#)
- [Get started](#)
 - [Debugging layout issues visually](#)
- [Inspecting a widget](#)
- [Flutter Layout Explorer](#)
 - [Using the Layout Explorer](#)
 - [Interactive Properties](#)
 - [mainAxisAlignment](#)
 - [crossAxisAlignment](#)
 - [FlexParentData.flex](#)
- [Track widget creation](#)
- [Other resources](#)

Note: The inspector works with Flutter mobile and web applications.

What is it?

The Flutter widget inspector is a powerful tool for visualizing and exploring Flutter widget trees. The Flutter framework uses widgets as the [core building block](#) for anything from controls (such as text, buttons, and toggles), to layout (such as centering, padding, rows and columns). The inspector helps you visualize and explore Flutter widget trees, and can be used for the following:

- understanding existing layouts
- diagnosing layout issues



Get started

To debug a layout issue, run the app in [debug mode](#) and open the inspector by clicking the **Flutter Inspector** tab on the DevTools toolbar.

[Get started](#)

[Samples & tutorials](#)

[Development](#)

▶ [User interface](#)

▶ [Data & backend](#)

▶ [Accessibility & internationalization](#)

▶ [Platform integration](#)

▶ [Packages & plugins](#)

▶ [Add Flutter to existing app](#)

▼ [Tools & techniques](#)

[Android Studio & IntelliJ](#)

[Visual Studio Code](#)

▼ [DevTools](#)

[Overview](#)

[Install from Android Studio & IntelliJ](#)

[Install from VS Code](#)

[Install from command line](#)

[Flutter inspector](#)

[Timeline view](#)

[Memory view](#)

[Performance view](#)

[Debugger](#)

[Logging view](#)

▶ [Flutter SDK](#)

[Hot reload](#)

[Code formatting](#)

▶ [Migration notes](#)

[Testing & debugging](#)

[Performance & optimization](#)

[Deployment](#)


[Resources](#)

[Reference](#)

[Widget index](#)

[API reference](#) 

[Package site](#) 

 **Note:** You can still access the Flutter inspector directly from Android Studio/IntelliJ, but you might prefer the more spacious view when running it from DevTools in a browser. Also note that the UI for the inspector varies slightly between these environments. This page describes the UI for the DevTools version of the inspector.

Debugging layout issues visually

The following is a guide to the features available in the inspector’s toolbar. When space is limited, the icon is used as the visual version of the label.

Select widget mode

Enable this button in order to select a widget on the device to inspect it. For more information, see [Inspecting a widget](#).

Refresh tree

Reload the current widget info.

Performance Overlay

Toggle display of performance graphs for the GPU & CPU threads. For more information on interpreting these graphs, see [The performance overlay](#) in [Flutter performance profiling](#).

iOS

Toggle rendering and gesture behaviors between Android and iOS.

Debug Paint

Add visual debugging hints to the rendering that display borders, padding, alignment, and spacers.

Paint Baselines

Cause each RenderBox to paint a line at each of its text baselines.

Slow Animations

Slow down animations to enable visual inspection.

Repaint Rainbow

Shows rotating colors on layers when repainting.

Debug Mode Banner

Toggles display of the debug banner even when running a debug build.


Inspecting a widget

You can browse the interactive widget tree to view nearby widgets and see their field values.

To locate individual UI elements in the widget tree, click the **Select Widget Mode** button in the toolbar. This puts the app on the device into a “widget select” mode. Click any widget in the app’s UI; this selects the widget on the app’s screen, and scrolls the widget tree to the corresponding node. Toggle the **Select Widget Mode** button again to exit widget select mode.

When debugging layout issues, the key fields to look at are the [size](#) and [constraints](#) fields. The constraints flow down the tree, and the sizes flow back up.

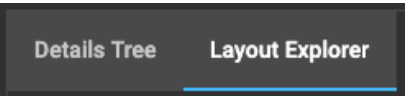
Flutter Layout Explorer

 **Note:** This feature is only available in the alpha version of [DevTools written in Flutter](#).

The Flutter Layout Explorer helps you to better understand Flutter layouts. Currently, the Layout Explorer only supports exploration of [flex layouts](#), but it may be extended to other types of layouts in the future.

Using the Layout Explorer

From the Flutter Inspector, select a flex widget (e.g., [Row](#), [Column](#), [Flex](#), etc.) or direct child of a flex widget. If you are using Flutter 1.12.16 or later, you will see an additional tab “Layout Explorer” next to “Details Tree”. Selecting this tab will display the new Layout Explorer feature.



The Layout Explorer visualizes how [Flex](#) widgets and their children are laid out. The explorer identifies the main axis and cross axis as well as the current alignment for each (e.g., start, end, spaceBetween, etc.). It also shows details like flex factor and layout constraints.

Additionally, the explorer shows layout constraint violations and render overflow errors. Violated layout constraints are colored red and overflow errors are presented in the standard “yellow-tape” pattern, as you would see on a running device. These visualizations aim to improve understanding of why overflow errors occur as well as how to fix them.

[Get started](#)

[Samples & tutorials](#)

[Development](#)

- ▶

User interface
- ▶

Data & backend
- ▶

Accessibility & internationalization
- ▶

Platform integration
- ▶

Packages & plugins
- ▶

Add Flutter to existing app
- ▼

Tools & techniques

Android Studio & IntelliJ

Visual Studio Code

▼ DevTools

Overview

Install from Android Studio & IntelliJ

Install from VS Code

Install from command line

Flutter inspector

Timeline view

Memory view

Performance view

Debugger

Logging view

▶

Flutter SDK

Hot reload

Code formatting

▶

Migration notes
- [Testing & debugging](#)
- [Performance & optimization](#)
- [Deployment](#)
- [Resources](#)
- [Reference](#)
- Widget index

API reference

Package site
- The image shows the Dart DevTools interface on the left and a mobile device on the right. The Dart DevTools window has the 'Flutter Inspector' tab selected. It shows a widget tree on the left with a 'Row' widget selected. The 'Layout Explorer' on the right shows a visual representation of the widget layout with dimensions and constraints. The mobile device on the right displays a boarding pass app with a teal header, flight details (YYC to YOW), a table of flight information, and a boarding pass barcode.
- Clicking on a widget in the layout explorer will mirror the selection on the on-device inspector. “Select Widget Mode” needs to be enabled for this. To enable it, click on the “Select Widget Mode” button in the inspector.
- A dark blue button with a white gear icon and the text 'Select Widget Mode' in white.
- For some properties, like flex factor and alignment, you can modify the value via dropdown lists in the explorer. When modifying a widget property, you will see the new value reflected not only in the Layout Explorer, but also on the device running your Flutter app. The explorer animates on property changes so that the effect of the change is clear. Widget property changes made from the layout explorer do not modify your source code and are reverted on hot reload.
- ## Interactive Properties
- Layout Explorer supports modifying [mainAxisAlignment](#), [crossAxisAlignment](#), and [FlexParentData.flex](#). In the future, we may add support for additional properties such as [mainAxisSize](#), [textDirection](#), and [FlexParentData.fit](#).
- ### mainAxisAlignment
- The image shows the Dart DevTools interface with the 'Layout Explorer' tab selected. It displays a visual representation of a widget layout with dimensions and constraints. The 'Main Axis' is highlighted with a red arrow pointing to the right. The 'Cross Axis' is highlighted with a green arrow pointing down. The 'Total Flex Factor' is shown as 1. The layout consists of a 'Row' widget containing three 'Column' widgets and a striped background. The dimensions and constraints are shown for each widget, such as 'w=471.0; (w=unconstrained)' for the first 'Column' and 'h=152.0; (h=unconstrained)' for the first 'Column'.
- Supported values:
- MainAxisAlignment.start
 - MainAxisAlignment.end
 - MainAxisAlignment.center
 - MainAxisAlignment.spaceBetween
 - MainAxisAlignment.spaceAround
 - MainAxisAlignment.spaceEvenly
- ### crossAxisAlignment

[Get started](#)

[Samples & tutorials](#)

[Development](#)

- ▶ [User interface](#)
- ▶ [Data & backend](#)
- ▶ [Accessibility & internationalization](#)
- ▶ [Platform integration](#)
- ▶ [Packages & plugins](#)
- ▶ [Add Flutter to existing app](#)
- ▼ [Tools & techniques](#)
 - [Android Studio & IntelliJ](#)
 - [Visual Studio Code](#)

- ▼ [DevTools](#)
 - [Overview](#)
 - [Install from Android Studio & IntelliJ](#)
 - [Install from VS Code](#)
 - [Install from command line](#)
 - [Flutter inspector](#)
 - [Timeline view](#)
 - [Memory view](#)
 - [Performance view](#)
 - [Debugger](#)
 - [Logging view](#)
 - ▶ [Flutter SDK](#)
 - [Hot reload](#)
 - [Code formatting](#)

▶ [Migration notes](#)

[Testing & debugging](#)

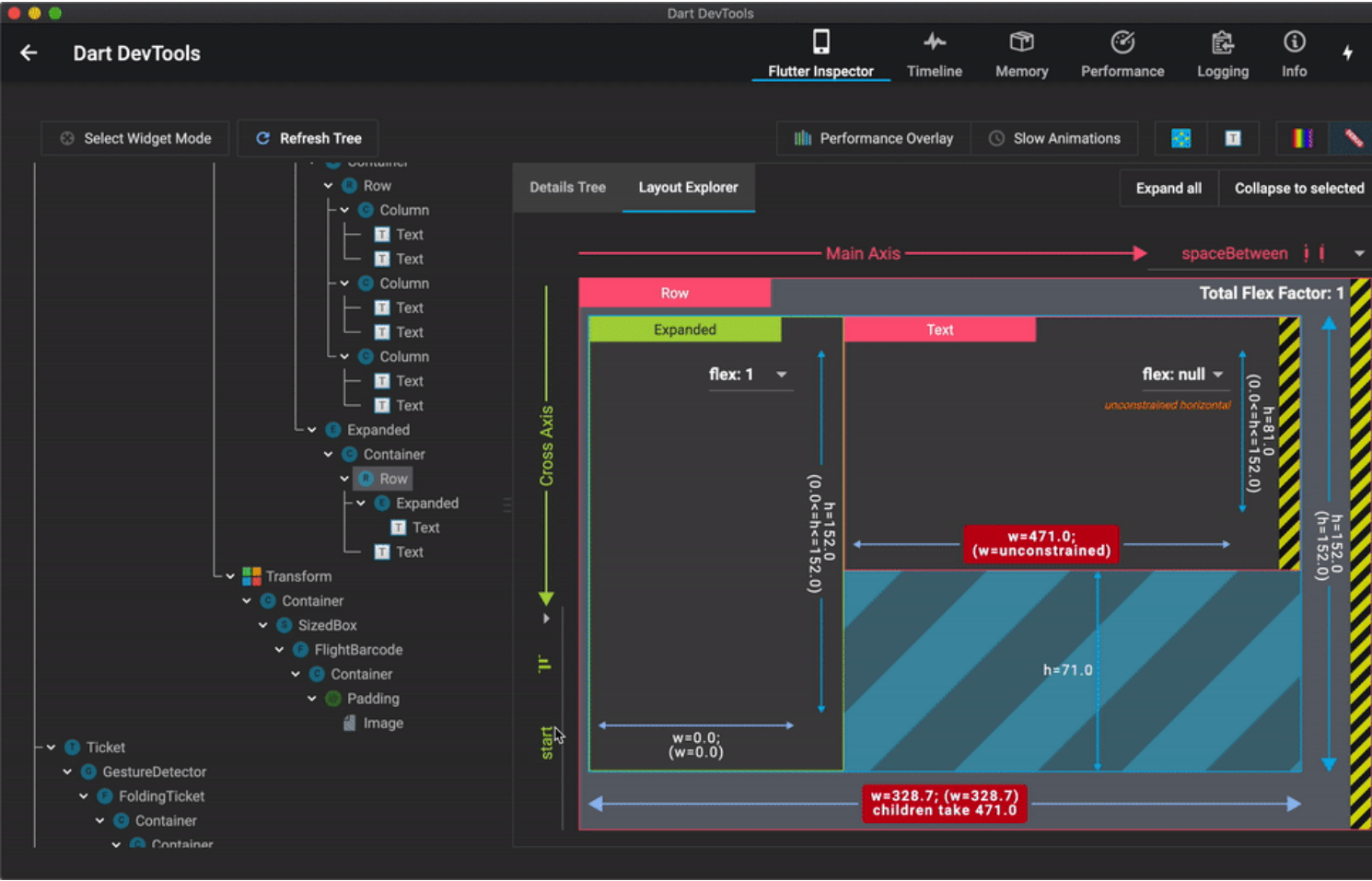
[Performance & optimization](#)

[Deployment](#)

[Resources](#)

[Reference](#)

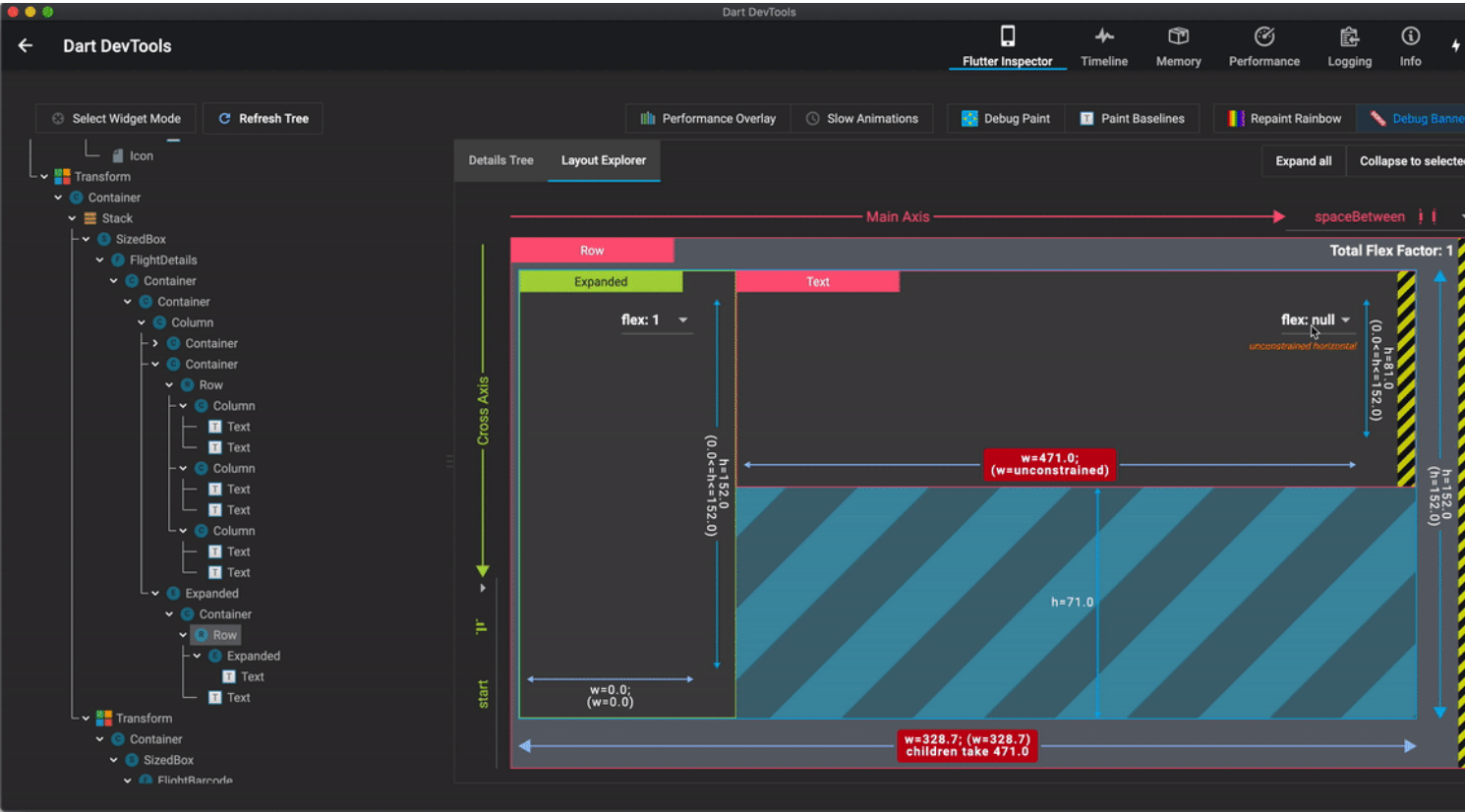
- [Widget index](#)
- [API reference](#)
- [Package site](#)



Supported values:

- `CrossAxisAlignment.start`
- `CrossAxisAlignment.center`
- `CrossAxisAlignment.end`
- `CrossAxisAlignment.stretch`

FlexParentData.flex



Layout Explorer supports 7 flex options in the UI (null, 0, 1, 2, 3, 4, 5), but technically the flex factor of a flex widget's child can be a `double`.

Track widget creation

Part of the functionality of the Flutter inspector is based on instrumenting the application code in order to better understand the source locations where widgets are created. The source instrumentation allows the Flutter inspector to present the widget tree in a manner similar to how the UI was defined in your source code. Without it, the tree of nodes in the widget tree are much deeper, and can be more difficult to understand how the runtime widget hierarchy corresponds to your application's UI.

When launching an application from an IDE, the source instrumentation happens by default. For command line launches, you need to opt-in to the source instrumentation. To do this, run the app with the `--track-widget-creation` flag:

```
flutter run --track-widget-creation
```

If you launch without the flag, you can still use the inspector—you'll see an inline, dismissable reminder message about using the source instrumentation flag.

Here are examples of what your widget tree might look like with and without track widget creation enabled.

^


```

graph TD
    Stack[Stack] --> OverlayEntry1[_OverlayEntry [LabeledGlobalKey<_OverlayEntryState>#16dbd]]
    OverlayEntry1 --> IgnorePointer[IgnorePointer]
    IgnorePointer --> ModalBarrier[ModalBarrier]
    ModalBarrier --> BlockSemantics[BlockSemantics]
    BlockSemantics --> ExcludeSemantics[ExcludeSemantics]
    ExcludeSemantics --> GestureDetector[GestureDetector]
    GestureDetector --> RawGestureDetector[RawGestureDetector]
    RawGestureDetector --> _GestureSemantics[_GestureSemantics]
    _GestureSemantics --> Listener[Listener]
    Listener --> Semantics[Semantics]
    Semantics --> ConstrainedBox[ConstrainedBox]

    OverlayEntry1 --> OverlayEntry2[_OverlayEntry [LabeledGlobalKey<_OverlayEntryState>#63026]]
    OverlayEntry2 --> _ModalScope[_ModalScope <dynamic>-[LabeledGlobalKey<_ModalScopeState<dynamic>>#5b9b8]]]
    _ModalScope --> _ModalScopeStatus[_ModalScopeStatus]
    _ModalScopeStatus --> Offstage[Offstage]
    Offstage --> PageStorage[PageStorage]
    PageStorage --> FocusScope[FocusScope]
    FocusScope --> Semantics2[Semantics]
    Semantics2 --> _FocusMarker[_FocusMarker]
    _FocusMarker --> RepaintBoundary1[RepaintBoundary]
    RepaintBoundary1 --> AnimatedBuilder1[AnimatedBuilder]
    AnimatedBuilder1 --> _FadeUpwardsPageTransition[_FadeUpwardsPageTransition]
    _FadeUpwardsPageTransition --> SlideTransition[SlideTransition]
    SlideTransition --> FractionalTranslation[FractionalTranslation]
    FractionalTranslation --> FadeTransition[FadeTransition]
    FadeTransition --> IgnorePointer2[IgnorePointer]
    IgnorePointer2 --> RepaintBoundary2[RepaintBoundary [GlobalKey#69b7e]]]
    RepaintBoundary2 --> Builder[Builder]
    Builder --> Semantics3[Semantics]
    Semantics3 --> GalleryHome[GalleryHome]
    GalleryHome --> AnnotatedRegion[AnnotatedRegion <SystemUiOverlayStyle>]
    AnnotatedRegion --> Scaffold[Scaffold [LabeledGlobalKey<ScaffoldState>#9d78a]]]
    Scaffold --> _ScaffoldScope[_ScaffoldScope]
    _ScaffoldScope --> PrimaryScrollController[PrimaryScrollController]
    PrimaryScrollController --> Material[Material]
    Material --> AnimatedPhysicalModel[AnimatedPhysicalModel]
    AnimatedPhysicalModel --> PhysicalModel[PhysicalModel]
    PhysicalModel --> NotificationListener[NotificationListener <LayoutChangedNotification>]
    NotificationListener --> _InkFeatures[_InkFeatures [GlobalKey#53fbf ink renderer]]]
    _InkFeatures --> AnimatedDefaultTextStyle[AnimatedDefaultTextStyle]
    AnimatedDefaultTextStyle --> DefaultTextStyle[DefaultTextStyle]
    DefaultTextStyle --> AnimatedBuilder2[AnimatedBuilder]
    AnimatedBuilder2 --> CustomMultiChildLayout[CustomMultiChildLayout]
    CustomMultiChildLayout --> LayoutId[LayoutId [<_ScaffoldSlot.body>]]]
    LayoutId --> MediaQuery1[MediaQuery]
    MediaQuery1 --> SafeArea[SafeArea]
    SafeArea --> Padding[Padding]
    Padding --> MediaQuery2[MediaQuery]
    MediaQuery2 --> WillPopScope[WillPopScope]
    WillPopScope --> Backdrop[Backdrop]
    Backdrop --> LayoutBuilder[LayoutBuilder]
  
```

For a demonstration of what's generally possible with the inspector, see the [DartConf 2018 talk](#) demonstrating the IntelliJ version of the Flutter inspector.