Scrollable is a superclass of such staple widgets as ListView, CustomScrollView, SingleChildScrollView, and many others. In this article, we are going to try to understand what is going on under the hood.

First, let's start with scroll update notifications.

1. What is a Notification?

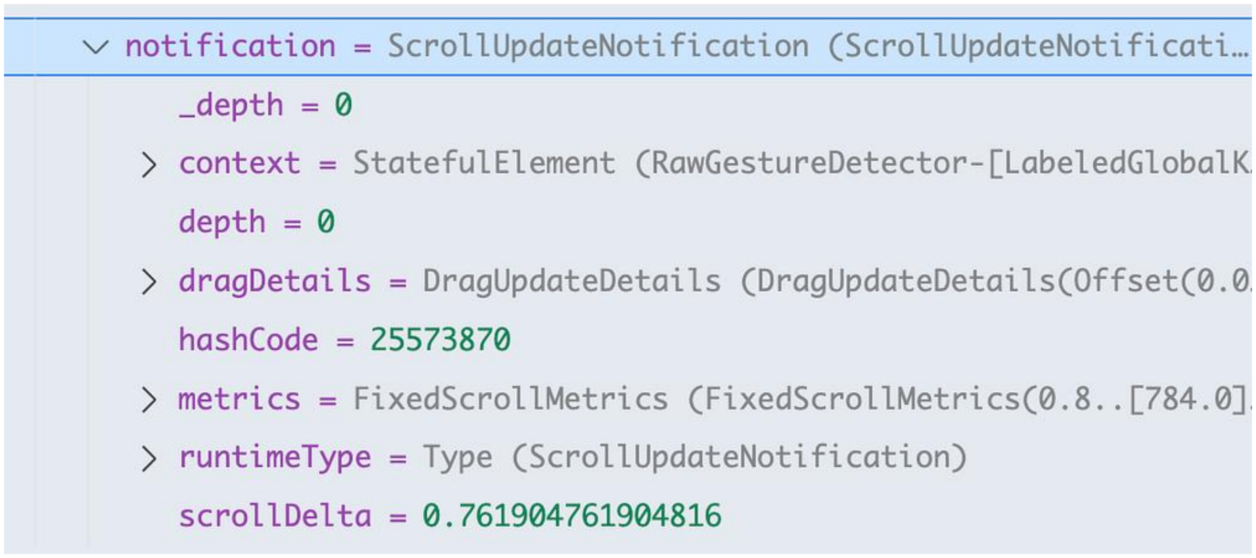
It is possible to send notifications up the widget tree. Flutter sends notifications on events such as scrolling, size changes, and layout changes.

In other words, whenever something scrolls or changes its size, the ancestors are notified about that.

Let's see what's inside of a notification sent when scrolling happens.

```
NotificationListener<ScrollUpdateNotification>(
 onNotification: (notification) {
  return false; // <- putting a debugger breakpoint here
 },
 child: ListView(
  children: [
   const SizedBox(height: 1000),
  ],
 ),
)
```

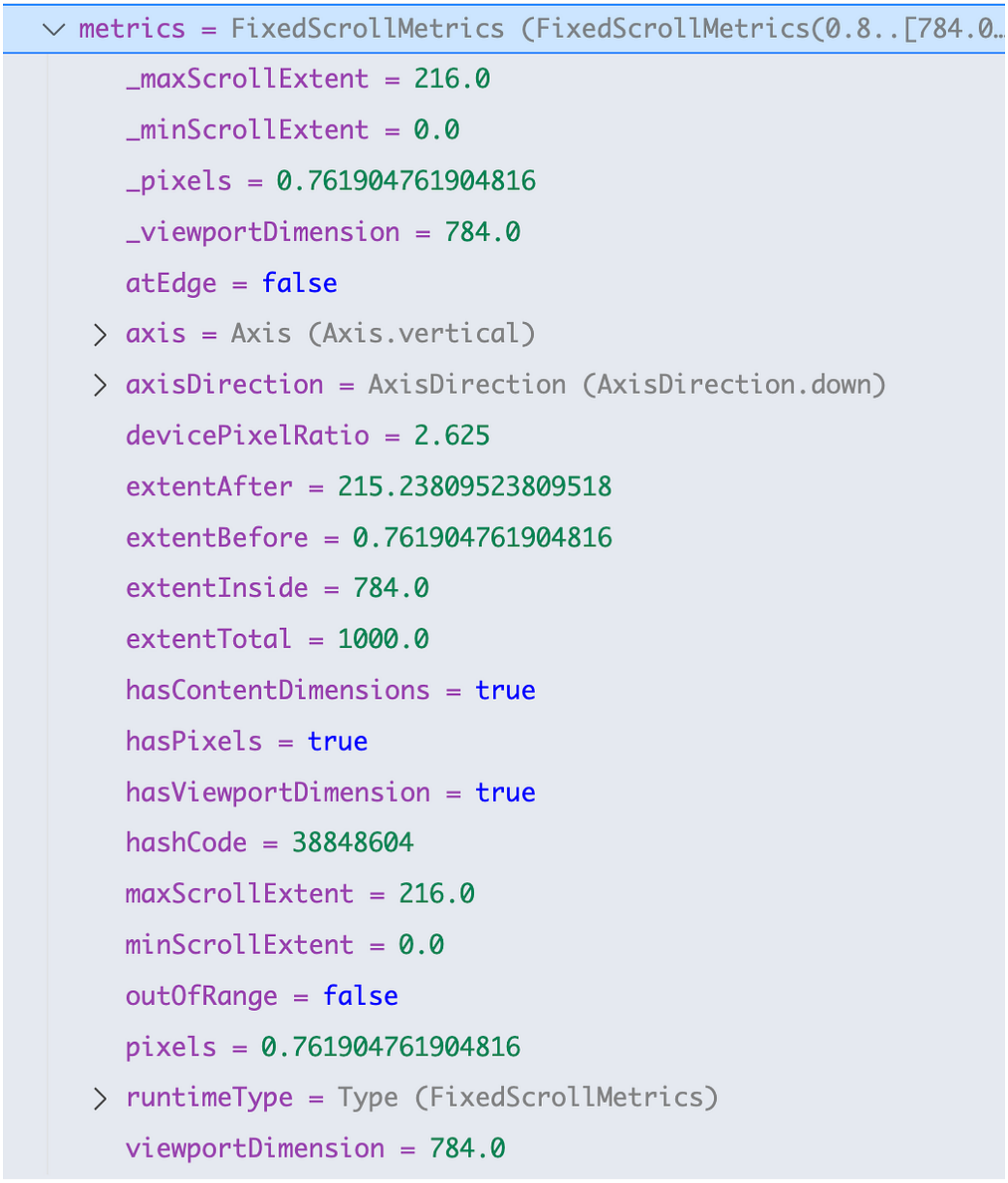
Press enter or click to view image in full size



```
∨ notification = ScrollUpdateNotification (ScrollUpdateNotificati…
    _depth = 0
  > context = StatefulElement (RawGestureDetector-[LabeledGlobalK
    depth = 0
  > dragDetails = DragUpdateDetails (DragUpdateDetails(Offset(0.0
    hashCode = 25573870
  > metrics = FixedScrollMetrics (FixedScrollMetrics(0.8..[784.0]
  > runtimeType = Type (ScrollUpdateNotification)
    scrollDelta = 0.761904761904816
```

First of all, let's pay attention to scrollDelta. This is the number of pixels added to the scroll position compared to the previous state. If it's positive, the user was scrolling along the main direction, if it is negative, he was scrolling backwards.

Press enter or click to view image in full size



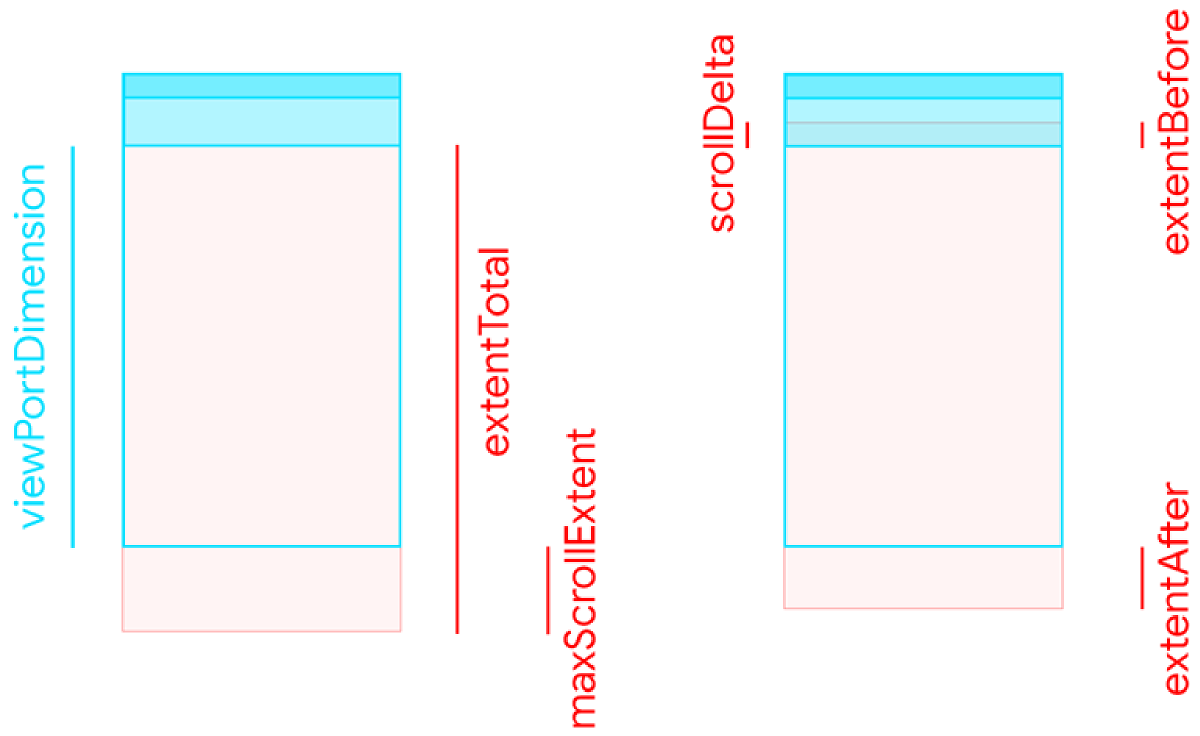
```
∨ metrics = FixedScrollMetrics (FixedScrollMetrics(0.8..[784.0…
      _maxScrollExtent = 216.0
      _minScrollExtent = 0.0
      _pixels = 0.761904761904816
      _viewportDimension = 784.0
      atEdge = false
    > axis = Axis (Axis.vertical)
    > axisDirection = AxisDirection (AxisDirection.down)
      devicePixelRatio = 2.625
      extentAfter = 215.23809523809518
      extentBefore = 0.761904761904816
      extentInside = 784.0
      extentTotal = 1000.0
      hasContentDimensions = true
      hasPixels = true
      hasViewportDimension = true
      hashCode = 38848604
      maxScrollExtent = 216.0
      minScrollExtent = 0.0
      outOfRange = false
      pixels = 0.761904761904816
    > runtimeType = Type (FixedScrollMetrics)
      viewportDimension = 784.0
```

If we go deeper and check what's inside of metrics, there will be a lot of useful data. Let's visualize this data.

The scrollable content is painted red, while the static widgets are blue.

The values not changing while scrolling are shown in the left half of the picture, and dynamic ones are on the right.

Press enter or click to view image in full size



So, extentTotal is the total height of Scrollable content. maxScrollExtent is the height of the content that could not fit in the viewport, scrollDelta is the raw number of pixels that have been scrolled since the previous notification, ententBefore and extentAfter correspond to the remaining height from the start and the end of the Scrollable.

viewPortDimension is the height of the widget that contains the Scrollable.

2. What happens if the content is smaller than the viewport?

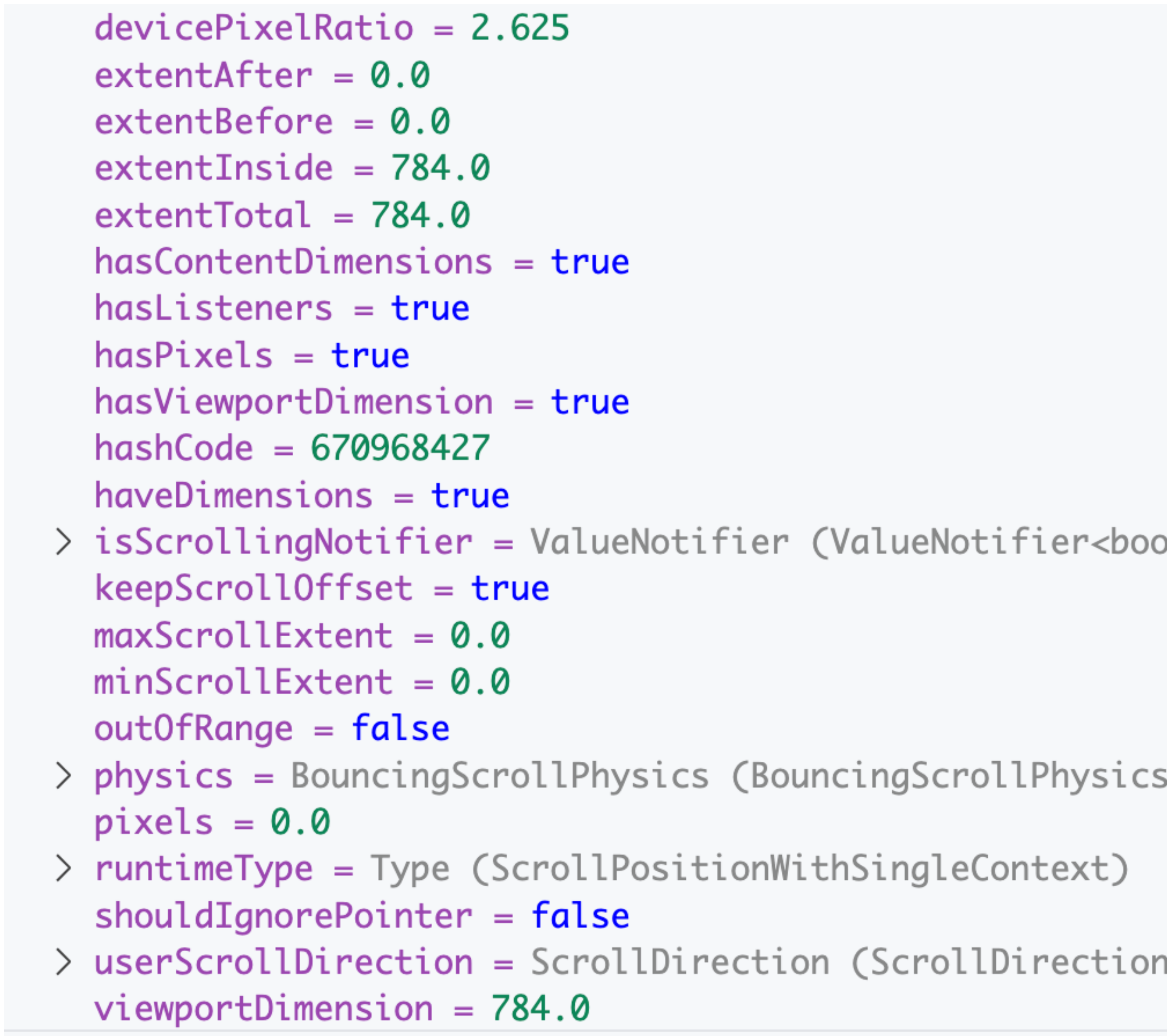
Let's change the SizedBox height from 1000 to 200. Now, the notification events are not sent because there's no scrolling happening — the scrolling content size is smaller than the viewport. How do we access it if we need these values?

```
class _ScrollableExampleState extends State<ScrollableExample> {
  final _controller = ScrollController(); // <-- add this

  @override
  void initState() {
    super.initState();
    WidgetsBinding.instance.addPostFrameCallback((timeStamp) {
      _controller.position; // <-- setting a debug breakpoint here
  });
 }

 ...
  ListView(
   controller: _controller, // <-- add this
   ...
  )
}
```

Press enter or click to view image in full size



```
   devicePixelRatio = 2.625
   extentAfter = 0.0
   extentBefore = 0.0
   extentInside = 784.0
   extentTotal = 784.0
   hasContentDimensions = true
   hasListeners = true
   hasPixels = true
   hasViewportDimension = true
   hashCode = 670968427
   haveDimensions = true
 > isScrollingNotifier = ValueNotifier (ValueNotifier<boo
   keepScrollOffset = true
   maxScrollExtent = 0.0
   minScrollExtent = 0.0
   outOfRange = false
 > physics = BouncingScrollPhysics (BouncingScrollPhysics
   pixels = 0.0
 > runtimeType = Type (ScrollPositionWithSingleContext)
   shouldIgnorePointer = false
 > userScrollDirection = ScrollDirection (ScrollDirection
   viewportDimension = 784.0
```

As we can see, viewport height is equal to extentTotal, and maxScrollExtent is 0, because there's no scrolling possible in this case.

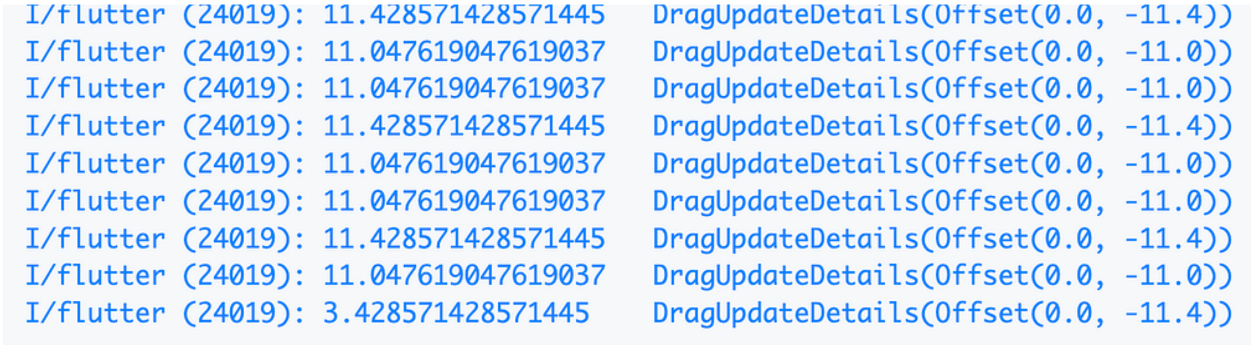
3. What are DragDetails?

If we go back to the notification object, we will notice that there is dragDetails property. This object is similar to the object sent in GestureDetector update callbacks. Let’s do some scrolling on the screen and watch the data being sent:

print("$scrollDelta\t$dragDetails");

Let’s scroll the widget with a quick flick and look at the data:

Press enter or click to view image in full size

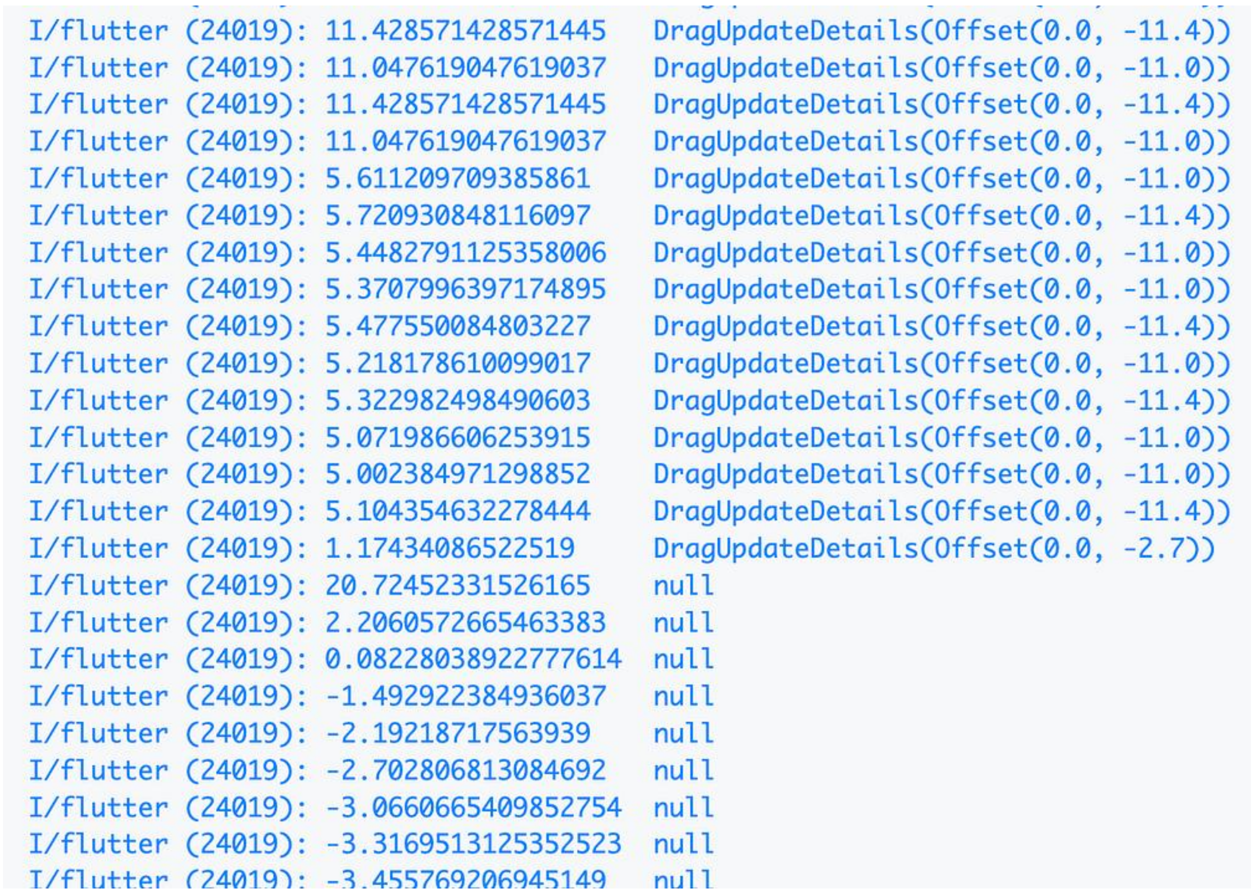


```
I/flutter (24019): 11.428571428571445   DragUpdateDetails(Offset(0.0, -11.4))
I/flutter (24019): 11.047619047619037   DragUpdateDetails(Offset(0.0, -11.0))
I/flutter (24019): 11.047619047619037   DragUpdateDetails(Offset(0.0, -11.0))
I/flutter (24019): 11.428571428571445   DragUpdateDetails(Offset(0.0, -11.4))
I/flutter (24019): 11.047619047619037   DragUpdateDetails(Offset(0.0, -11.0))
I/flutter (24019): 11.047619047619037   DragUpdateDetails(Offset(0.0, -11.0))
I/flutter (24019): 11.428571428571445   DragUpdateDetails(Offset(0.0, -11.4))
I/flutter (24019): 11.047619047619037   DragUpdateDetails(Offset(0.0, -11.0))
I/flutter (24019): 3.428571428571445    DragUpdateDetails(Offset(0.0, -11.4))
```

At first sight, it looks like scrollDelta and dragDetails.y have negated, but similar values, but check the last one. Can you guess what happened here? Hint: this was done on Android.

Let’s run that on an iPhone and see:

Press enter or click to view image in full size

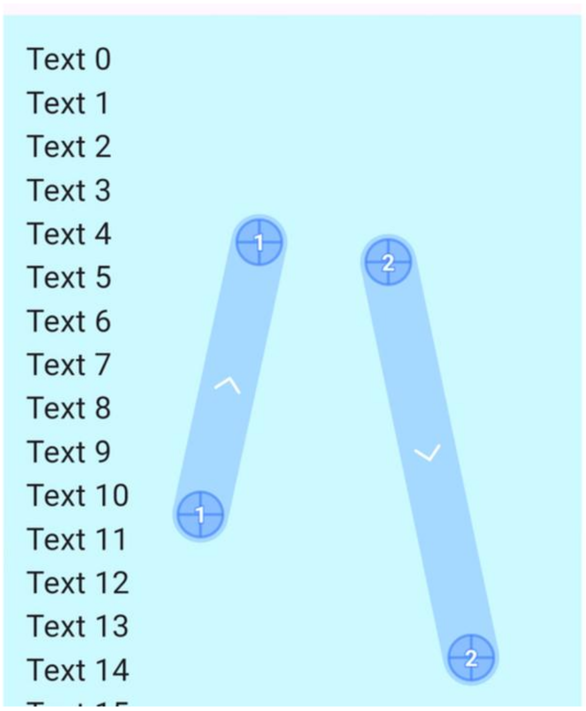


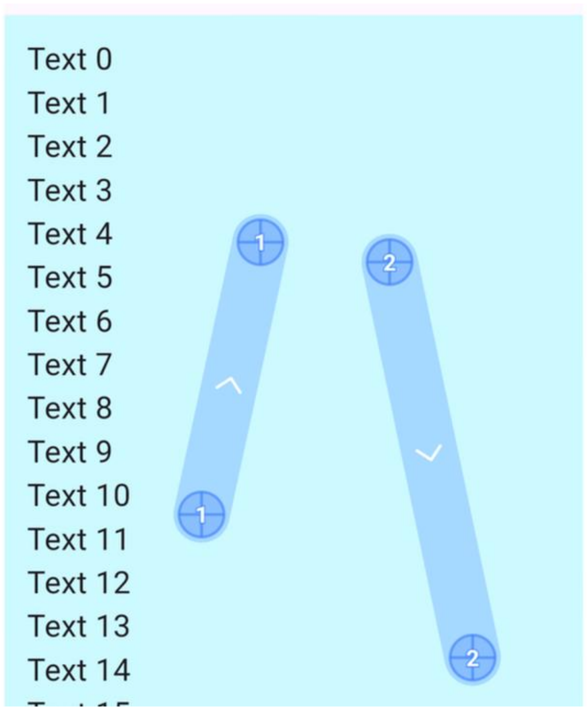
```
I/flutter (24019): 11.428571428571445   DragUpdateDetails(Offset(0.0, -11.4))
I/flutter (24019): 11.047619047619037   DragUpdateDetails(Offset(0.0, -11.0))
I/flutter (24019): 11.428571428571445   DragUpdateDetails(Offset(0.0, -11.4))
I/flutter (24019): 11.047619047619037   DragUpdateDetails(Offset(0.0, -11.0))
I/flutter (24019): 5.611209709385861    DragUpdateDetails(Offset(0.0, -11.0))
I/flutter (24019): 5.720930848116097    DragUpdateDetails(Offset(0.0, -11.4))
I/flutter (24019): 5.4482791125358006   DragUpdateDetails(Offset(0.0, -11.0))
I/flutter (24019): 5.3707996397174895   DragUpdateDetails(Offset(0.0, -11.0))
I/flutter (24019): 5.477550084803227    DragUpdateDetails(Offset(0.0, -11.4))
I/flutter (24019): 5.218178610099017    DragUpdateDetails(Offset(0.0, -11.0))
I/flutter (24019): 5.322982498490603    DragUpdateDetails(Offset(0.0, -11.4))
I/flutter (24019): 5.071986606253915    DragUpdateDetails(Offset(0.0, -11.0))
I/flutter (24019): 5.002384971298852    DragUpdateDetails(Offset(0.0, -11.0))
I/flutter (24019): 5.104354632278444    DragUpdateDetails(Offset(0.0, -11.4))
I/flutter (24019): 1.17434086522519     DragUpdateDetails(Offset(0.0, -2.7))
I/flutter (24019): 20.72452331526165    null
I/flutter (24019): 2.2060572665463383   null
I/flutter (24019): 0.08228038922777614  null
I/flutter (24019): -1.492922384936037   null
I/flutter (24019): -2.19218717563939    null
I/flutter (24019): -2.702806813084692   null
I/flutter (24019): -3.0660665409852754  null
I/flutter (24019): -3.3169513125352523  null
I/flutter (24019): -3.455769206945149   null
```

Drag data is similar, but scrollDelta is different. After that drag was finished, the notifications were still sent, but without drag update details.

Of course, the reason is having different ScrollPhysics applied by default in Scrollable. By default, BouncingScrollPhysics is applied on iOS and ClampingScrollPhysics is applied on Android.

Press enter or click to view image in full size





BouncingScrollPhysics vs ClampingScrollPhysics

4. So how does ScrollPhysics work?

- Receives drag details and scroll delta
- Processes through simulation layers
- Applies boundaries if needed
- Outputs scroll position and velocity
- Scrollable sends calculated value as a notification to the listeners

In Flutter you can choose between different physics or create a custom one. Besides, it is possible to apply several scroll physics at once like this:

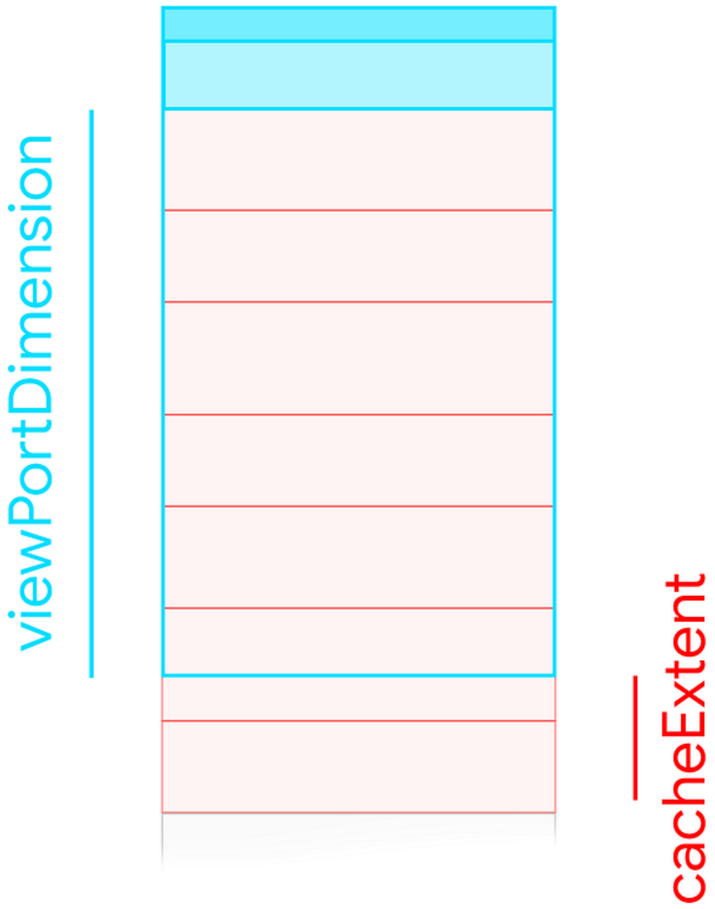
BouncingScrollPhysics(parent: AlwaysScrollableScrollPhysics())

This example will first apply simulations of BouncingScrollPhysics and then the ones of AlwaysScrollableScrollPhysics.

5. Is the total extent always known?

No, there are cases when we calculate the size of every item in the Scrollable, for example, by using ListView.builder when item size depends on its contents, say, there is a Text widget that can have either 1 or 2 line height.

Press enter or click to view image in full size



In this case, we can’t calculate the totalExtent because that would mean that we have to do the calculation for the whole list, which defeats the purpose of lazy loading. In this case, Flutter will only instantiate items visible in the viewport + cacheExtent, which is a parameter of ListView. Note, that if an item would fit in the cacheExtent only partially, it will still be instantiated.

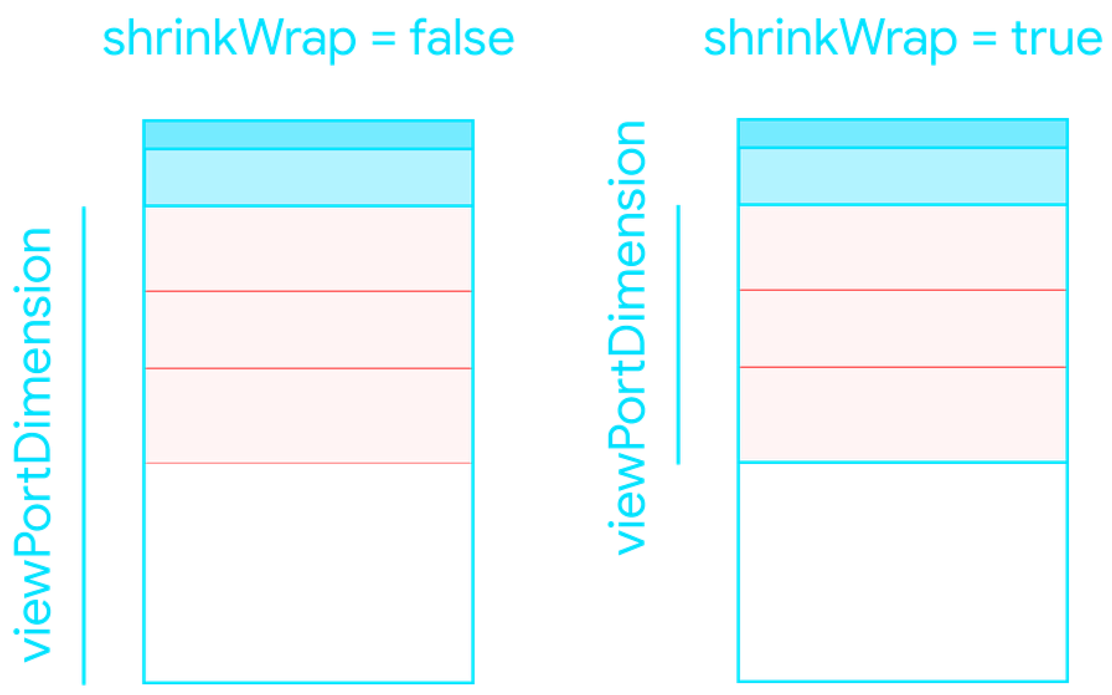
Get Roman Ismagilov’s stories in your inbox

Join Medium for free to get updates from this writer.

Subscribe

When the content is smaller than the parent widget, there are two options for calculating the viewPort size.

Press enter or click to view image in full size



By setting shrinkWrap to true the renderer will be forced to calculate the height of the viewPort based on its children. This way, the lazy loading will be disabled and that could lead to performance issues. Use it only if you are sure that there won't be a lot of objects in the list.

6. So why can't I put a Spacer or a Flexible in a Scrollable?

```
SingleChildScrollView(
  child: Column(
    children: [
      Text("Content"),
      const Spacer(), // <- don't do that
      ElevatedButton(onPressed: () {}, child: Text("Button"))
    ],
  ),
)
```

There is a conflict: Spacer wants to take as much space as possible, while Scrollable allows its children to take as much space as they need. In this example, Spacer needs to know the parent's size to calculate its height, but it's impossible because the parent's size depends on the child.

```
LayoutBuilder(builder: (context, constraints) {
  return SingleChildScrollView(
    child: ConstrainedBox(
      constraints: BoxConstraints(
        minHeight: constraints.maxHeight,
        maxHeight: double.infinity,
      ),
      child: IntrinsicHeight(
        child: Column(
          children: [
            Text("Content"),
            const Spacer(),
            ElevatedButton(onPressed: () {}, child: Text("Button"))
          ],
        ),
      ),
    ),
  );
})
```

LayoutBuilder:

- Gets parent constraints
- Provides size context

SingleChildScrollView:

- Enables scrolling when content overflows

ConstrainedBox:

- Sets minimum height = parent height
- Allows infinite maximum height
- Prevents column from collapsing

IntrinsicHeight:

- Forces column to calculate the proper height
- Helps with child sizing

Column:

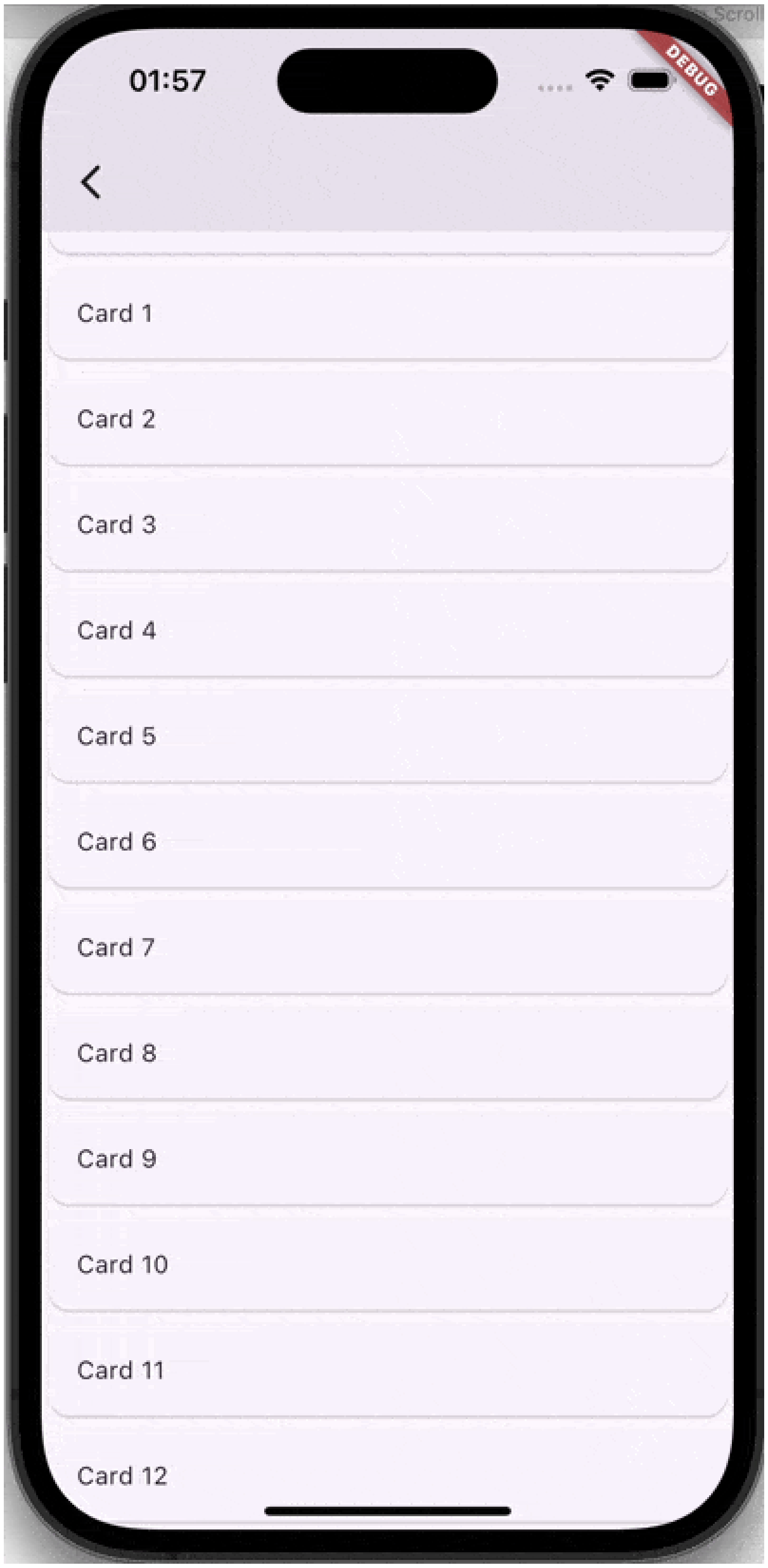
- Arranges children vertically
- Expands to maximum space

As a result, we can use a Spacer or a Flexible in a Scrollable.

You can also use ScrollableColumn widget from this package.

7. How to use Scrollable and Transform?

Let's implement a scroll listener that would update a view in the app bar on scroll.


01:57
DEBUG
Card 1
Card 2
Card 3
Card 4
Card 5
Card 6
Card 7
Card 8
Card 9
Card 10
Card 11
Card 12

Let’s start with creating a StatefulWidget. It could be Stateless as well, depending on the state management approach you use. I want it to be as simple as possible in this example, so I am just using a ValueNotifier as a property of the StatefulWidget.

```
class _ScrollableZoomerState extends State<ScrollableZoomer> {
 ValueNotifier<double> scrollPosition = ValueNotifier(0.0);

 …
}
```

It will store a normalized scroll position, where 0.0 is the start and 1.0 is fully scrolled.

```
 @override
 Widget build(BuildContext context) {
  return Scaffold(
   appBar: ...,
   body: NotificationListener<ScrollUpdateNotification>(
    onNotification: (notification) {
      scrollPosition.value = min(1, notification.metrics.pixels /
        notification.metrics.maxScrollExtent);

     return true;
    },
    child: widget.child,
   ),
  );
 }
```

Pretty simple, just dividing scrolled pixels by max scroll extent and making sure it won’t exceed 100%. Then apply some simple transformations to the Next button.

```
appBar: AppBar(
 actions: [
  ValueListenableBuilder(
   valueListenable: scrollPosition,
   builder: (context, value, _) => Opacity(
    opacity: value > 0.1 ? value : 0,
    child: Transform.translate(
     offset: Offset(0, 40 * (1 - value)),
     child: TextButton(
      onPressed: ...,
      child: Text("Next"),
     ),
    ),
   ),
    )
  ],
),
```

You can play around with different values and math operations, the only limit is imagination.

Hope you’ve found this article useful. I will update it with more information whenever I find something useful. Follow me on Twitter to get the latest updates.