9

Creating Scrolling Lists and Effects

WHAT YOU WILL LEARN IN THIS CHAPTER

- How `Card` is a great way to group information with the container having rounded corners and a drop shadow
- How to build a linear list of scrollable widgets with `ListView`
- How to display tiles of scrollable widgets in a grid format with `GridView`
- How `Stack` lets you overlap, position, and align its children widgets
- How to create custom scrolling effects using `CustomScrollView` and slivers

In this chapter, you'll learn to create scrolling lists that help users view and select information. You'll start with the `Card` widget in this chapter because it is commonly used in conjunction with list-capable widgets to enhance the user interface (UI) and group data. In the previous chapter, you took a look at using the basic constructor for the `ListView`, and in this chapter, you'll use the `ListView.builder` to customize the data. The `GridView` widget is a fantastic widget that displays a list of data by a fixed number of tiles (groups of data) in the cross axis. The `Stack` widget is commonly used to overlap, position, and align widgets to create a custom look. A good example is a shopping cart with the number of items to purchase on the upper-right side.

The `CustomScrollView` widget allows you to create custom scrolling effects by using a list of slivers widgets. Slivers are handy, for instance, if you have a journal entry with an image on the top of the page and the diary description below. When the user swipes to read more, the description scrolling is faster than the image scrolling, creating a parallax effect.

Beginning Flutter®: A Hands On Guide To App Development, First Edition.
Marco L. Napoli.

USING THE CARD

The `Card` widget is part of the Material Design and has minimal rounded corners and shadows. To group and lay out data, the `Card` is a perfect widget to enhance the look of the UI. The `Card` widget is customizable with properties such as `elevation`, `shape`, `color`, `margin`, and others. The `elevation` property is a value of `double`, and the higher the number, the larger the shadow that is cast. You learned in Chapter 3, "Learning Dart Basics," that a double is a number that requires decimal point precision, such as 8.50. To customize the shape and borders of the `Card` widget, you modify the `shape` property. Some of the `shape` properties are `StadiumBorder`, `UnderlineInputBorder`, `OutlineInputBorder`, and others.

```
Card(
  elevation: 8.0,
  color: Colors.white,
  margin: EdgeInsets.all(16.0),
  child: Column(
    mainAxisAlignment: MainAxisAlignment.center,
    children: <Widget>[
      Text('
        'Barista',
        textAlign: TextAlign.center,
        style: TextStyle(
          fontWeight: FontWeight.bold,
          fontSize: 48.0,
          color: Colors.orange,
        ),
      '),
      Text(
        'Travel Plans',
        textAlign: TextAlign.center,
        style: TextStyle(color: Colors.grey),
      ),
    ],
  ),
),
```

The following are a few ways to customize the `Card`'s `shape` property (Figure 9.1):

```
// Create a Stadium Border
shape: StadiumBorder(),

// Create Square Corners Card with a Single Orange Bottom Border
shape: UnderlineInputBorder(borderSide: BorderSide(color: Colors.deepOrange)),

// Create Rounded Corners Card with Orange Border
shape: OutlineInputBorder(borderSide: BorderSide(color: Colors.deepOrange.
withOpacity(0.5)),),
```

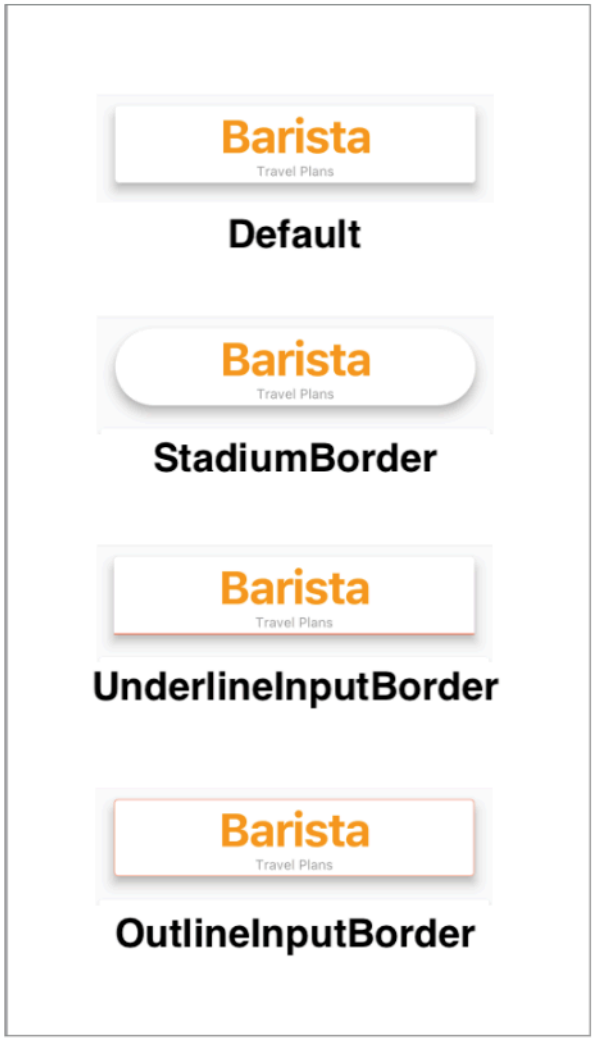


FIGURE 9.1: Card customizations

USING THE LISTVIEW AND LISTTILE

The constructor `ListView.builder` is used to create an on-demand linear scrollable list of widgets (Figure 9.2). When you have a large set of data, the `builder` is called only for visible widgets, which is great for performance. Within the `builder`, you use the `itemBuilder` callback to create the list of children widgets. Keep in mind the `itemBuilder` is called only if the `itemCount` argument is greater than zero, and it is called as many times as the `itemCount` value. Remember, the `List` starts at row 0, not 1. If you have 20 items in the `List`, it loops from row 0 to 19. The `scrollDirection` argument defaults to `Axis.vertical` but can be changed to `Axis.horizontal`.

The `ListTile` widget is commonly used with the `ListView` widget to easily format and organize icons, titles, and descriptions in a linear layout. Among the main properties are `leading`, `trailing`, `title`, and `subtitle` properties, but there are others. You can also use the `onTap` and `onLongPress` callbacks to execute an action when the user taps the `ListTile`. Usually the `leading` and `trailing` properties are implemented with icons, but you can add any type of widget.

In Figure 9.2, the first three `ListTiles` show an icon for the `leading` property, but for the `trailing` property, a `Text` widget shows a percentage value. The remaining `ListTiles` show both the `leading` and `trailing` properties as icons. Another scenario is to use the `subtitle` property to show a progress bar instead of additional text description since these properties accept widgets.

The first code example here shows a `Card` with the `child` property as a `ListTile` to give it a nice frame and shadow effect. The second example shows a base `ListTile`.

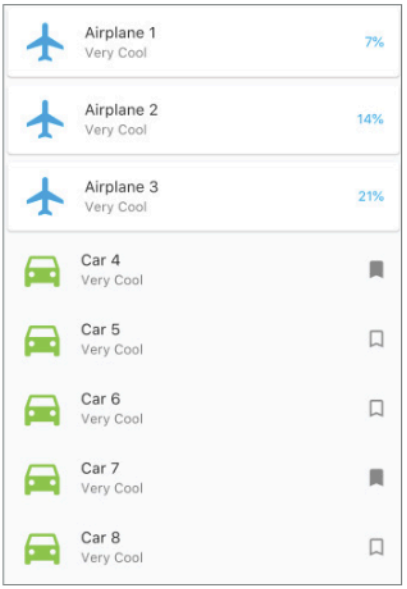


FIGURE 9.2: ListView linear layout using the ListTile

```
// Card with a ListTile widget
Card(
  child: ListTile(
    leading: Icon(Icons.flight),
    title: Text('Airplane $index'),
    subtitle: Text('Very Cool'),
    trailing: Text('${index * 7}%'),
    onTap: ()=> print('Tapped on Row $index'),
  ),
);

// ListTile
ListTile(
  leading: Icon(Icons.directions_car),
  title: Text('Car $index'),
  subtitle: Text('Very Cool'),
  trailing: Icon(Icons.bookmark),
  onTap: ()=> print('Tapped on Row $index'),
);
```

TRY IT OUT Creating the ListView App

In this example, the `ListView` widget uses the `builder` to display a `Card` for the header and two variations of the `ListTile` for the data list. The `ListTile` can display `leading` and `trailing` widgets. The `leading` property shows an `Icon` but could have displayed an `Image`. For the `trailing` property, the first type of `ListTile` shows data as a percentage, and the second `ListTile` shows a selected or unselected bookmark `Icon`. You also set a `title` and `subtitle`, and for the `onTap` you use the `print` statement to show the tapped row index value.

1. Create a new Flutter project and name it `ch9_listview`. You can follow the instructions in Chapter 4, "Creating a Starter Project Template." For this project, you need to create only the `pages` and `widgets` folders. Create the `Home Class` as a `StatelessWidget` since the data does not require changes.

2. Open the `home.dart` file and add to the `body` a `SafeArea` with `ListView.builder()` as a `child`.

```
body: SafeArea(
  child: ListView.builder(),
),
```

3. Set the `ListView.builder` with the `itemCount` argument set to 20. For this example, you specify 20 rows of data. For the `itemBuilder` callback, it passes the `BuildContext` and the widget `index` as an `int` value.

   To show how to create different types of widgets to list each row of data, let's check the first row for the index value of zero and call the `HeaderWidget(index: index)` widget class. This class shows a `Card` with the text "Barista Travel Plan."

   For the first, second, and third rows, the widget class `RowWithCardWidget(index: index)` is called to show a `ListTile` as a `child` of a `Card`. For the rest of the rows, you call the `RowWidget(index: index)` widget class to show a default `ListTile`.

   It's important to understand that the widget classes you call from the `itemBuilder` create a unique widget with the index value passed. The `itemBuilder` loops the `itemCount` value, in this example 20 times.

   You'll create the three widget classes in step 5.

```
body: SafeArea(
  child: ListView.builder(
    itemCount: 20,
    itemBuilder: (BuildContext context, int index) {
      if (index == 0) {
        return HeaderWidget(index: index);
      } else if (index >= 1 && index <= 3) {
        return RowWithCardWidget(index: index);
      } else {
        return RowWidget(index: index);
      }
    },
  ),
),
```

4. Add to the top of the file the `import` statements for the `header.dart`, `row_with_card.dart`, and `row.dart` widget classes that you'll create next.

```
import 'package:flutter/material.dart';
import 'package:ch9_listview/widgets/header.dart';
import 'package:ch9_listview/widgets/row_with_card.dart';
import 'package:ch9_listview/widgets/row.dart';
```

5. Create a new Dart file in the `widgets` folder. Right-click the `widgets` folder and then select New ➪ Dart File, enter `header.dart`, and click the OK button to save.

6. Import the `material.dart` library, add a new line, and then start typing st; the autocompletion help opens, so select the `stless` (`StatelessWidget`) abbreviation and give it a name of `HeaderWidget`.

7. Modify the `HeaderWidget` widget class to return a `Container`. Import the `material.dart` library.

   The `Container child` is a `Card` with an `elevation` of 8.0 to show a deep shadow. The `Card children` list of widgets returns two `Text` widgets. I purposely left three `shape` types commented out for you to test and see how they change the `shape` and borders of the `Card`.

   I wanted to point out that the `ListView itemBuilder` in the `main.dart` file calls this class, the `HeaderWidget(index: index)` for each row item. For every row, a `Widget` is created and added to the widget tree.

   Note I commented out three different ways to customize the default `Card` shape for your testing.

```
import 'package:flutter/material.dart';

class HeaderWidget extends StatelessWidget {
  const HeaderWidget({
    Key key,
    @required this.index,
  }) : super(key: key);

  final int index;

  @override
  Widget build(BuildContext context) {
    return Container(
      padding: EdgeInsets.all(16.0),
      height: 120.0,
      child: Card(
        elevation: 8.0,
        color: Colors.white,
        //shape: StadiumBorder(),
        //shape: UnderlineInputBorder(borderSide: BorderSide(color: Colors
.deepOrange)),
        //shape: OutlineInputBorder(borderSide: BorderSide(color:
Colors.deepOrange.withOpacity(0.5)),),
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text(
              'Barista',
              textAlign: TextAlign.center,
              style: TextStyle(
                fontWeight: FontWeight.bold,
                fontSize: 48.0,
                color: Colors.orange,
              ),
            ),
            Text(
              'Travel Plans',
              textAlign: TextAlign.center,
              style: TextStyle(color: Colors.grey),
            ),
          ],
```

```
        ),
      ),
    );
  }
}
```

8. Create a new Dart file in the `widgets` folder. Right-click the `widgets` folder and then select New ➪ Dart File, enter `row_with_card.dart`, and click the OK button to save.

9. Import the `material.dart` library, add a new line, and then start typing st; the autocompletion help opens, so select the `stless` (`StatelessWidget`) abbreviation and give it a name of `RowWithCardWidget`.

10. Modify the `RowWithCardWidget` class widget to return a `Card`. Import the `material.dart` library.

    The `Card child` is a `ListTile`, which is great to align content easily. For the `leading` property, return an `Icon`. The `trailing` property returns a `Text` widget with string interpolation that takes the `index` times seven to obtain a number. The `title` property returns a `Text` widget with the `index` value. The `subtitle` property returns a `Text` widget. For the `onTap` property, you use a `print` statement to show the tapped row `index`.

    As a reminder, a widget is created and added to the widget tree for each row.

```
import 'package:flutter/material.dart';

class RowWithCardWidget extends StatelessWidget {
  const RowWithCardWidget({
    Key key,
    @required this.index,
  }) : super(key: key);

  final int index;

  @override
  Widget build(BuildContext context) {
    return Card(
      child: ListTile(
        leading: Icon(
          Icons.flight,
          size: 48.0,
          color: Colors.lightBlue,
        ),
        title: Text('Airplane $index'),
        subtitle: Text('Very Cool'),
        trailing: Text(
          '${index * 7}%',
          style: TextStyle(color: Colors.lightBlue),
        ),
        //selected: true,
        onTap: () {
          print('Tapped on Row $index');
        },
      ),
    );
  }
}
```

11. Create a new Dart file in the `widgets` folder. Right-click the `widgets` folder and then select New ➪ Dart File, enter `row.dart`, and click the OK button to save.

12. Import the `material.dart` library, add a new line, and then start typing st; the autocompletion help opens, so select the `stless` (`StatelessWidget`) abbreviation and give it a name of `RowWidget`.

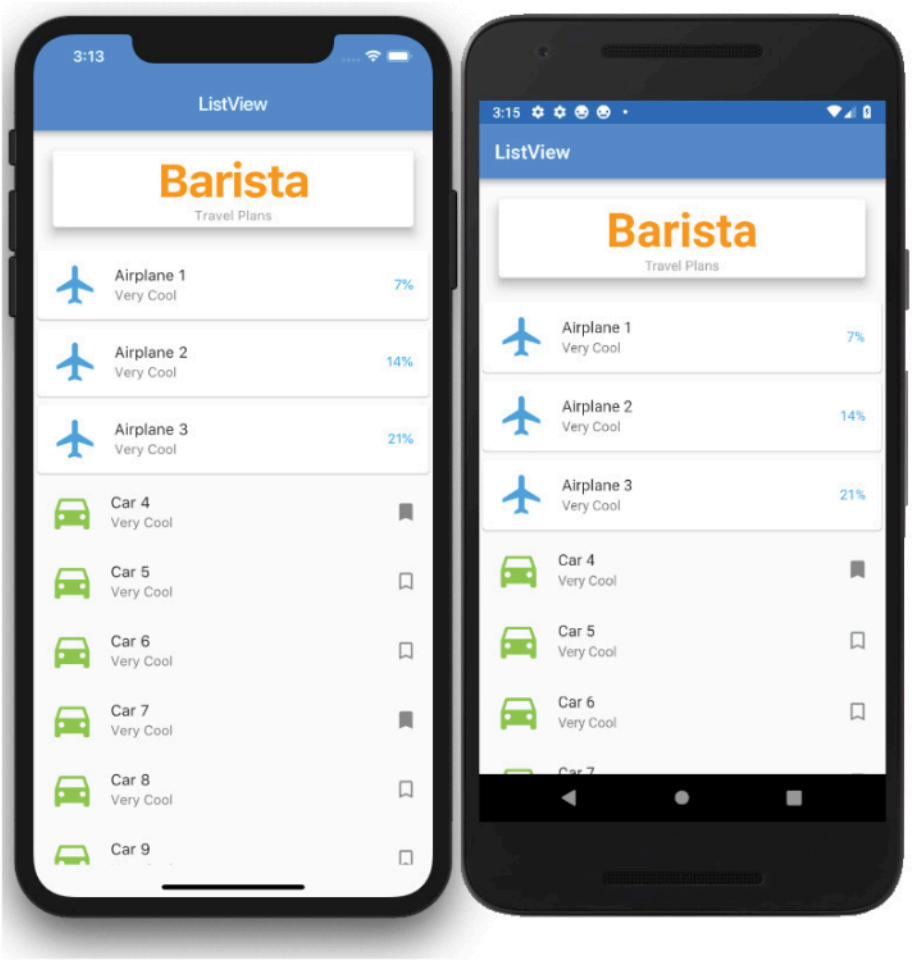
13. Modify the `RowWidget` widget class to return a `ListTile`. Import the `material.dart` library.

14. For the `leading` property, return an `Icon`. For the `trailing` property, you return either a `bookmark_border` or a `bookmark` `Icon`. To randomize which `Icon` to return, use a ternary operator to calculate the `index` `modulus` (%) of 3 and check whether it's an even number. If the number is even or odd, the appropriate `Icon` is shown. The `title` property returns a `Text` widget with the `index` value. The `subtitle` property returns a `Text` widget.

    For the `onTap`, you use a `print` statement to show the tapped row `index`.

    Note that for each row a widget is added to the widget tree.

```
import 'package:flutter/material.dart';

class RowWidget extends StatelessWidget {
  const RowWidget({
    Key key,
    @required this.index,
  }) : super(key: key);

  final int index;

  @override
  Widget build(BuildContext context) {
    return ListTile(
      leading: Icon(
        Icons.directions_car,
        size: 48.0,
        color: Colors.lightGreen,
      ),
      title: Text('Car $index'),
      subtitle: Text('Very Cool'),
      trailing: (index % 3).isEven
          ? Icon(Icons.bookmark_border)
          : Icon(Icons.bookmark),
      selected: false,
      onTap: () {
        print('Tapped on Row $index');
      },
    );
  }
}
```



HOW IT WORKS

The `ListView.builder` constructor takes an `itemCount` and uses the `itemBuilder` to build a widget for each child record. Each child widget is added to the widget tree with the appropriate values. As each child widget is added, you can customize the `ListView` rows according to app specs.

Using the `ListTile` makes it extremely easy to align widgets. The `ListTile` takes a `leading`, `trailing`, `title`, `subtitle`, `onTap`, and other properties.

USING THE GRIDVIEW

The `GridView` (Figure 9.3) displays tiles of scrollable widgets in a grid format. The three constructors that I focus on are `GridView.count`, `GridView.extent`, and `GridView.builder`.

The `GridView.count` and `GridView.extent` are usually used with a fixed or smaller data set. Using these constructors means that all of the data, not just visible widgets, is loaded at `init`. If you have a large set of data, the user does not see the `GridView` until all data is loaded, which is not a great user experience (UX). Usually you use the `GridView.count` when you need a layout with a fixed number of tiles in the cross-axis. For example, it shows three tiles in portrait and landscape modes. You use the `GridView.extent` when you need a layout with the tiles that need a maximum cross-axis extent. For example, two to three tiles fit in portrait mode, and five to six tiles fit in landscape mode; in other words, it fits as many tiles that it can depending on screen size.

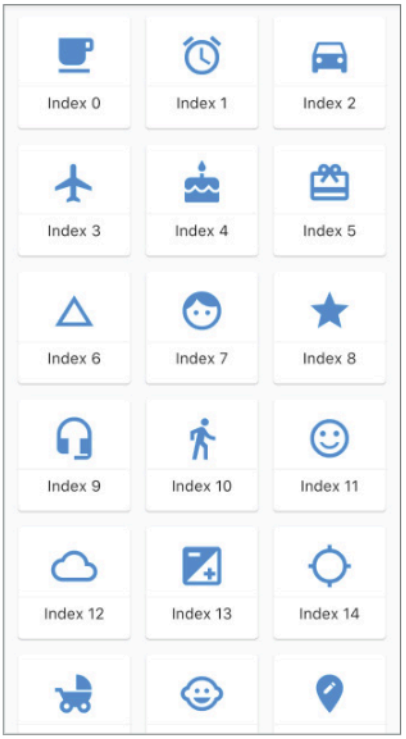


FIGURE 9.3: GridView layout

The `GridView.builder` constructor is used with a larger, infinite, or unknown size set of data. Like our `ListView.builder`, when you have a large set of data, the `builder` is called only for visible widgets, which is great for performance. Within the `builder`, you use the `itemBuilder` callback to create the list of children widgets. Keep in mind the `itemBuilder` is called only if the `itemCount` argument is greater than zero, and it is called as many times as the `itemCount` value. Remember, the `List` starts at row 0, not 1. If you have 20 items in the `List`, it loops from row 0 to 19. The `scrollDirection` argument defaults to `Axis.vertical` but can be changed to `Axis.horizontal` (Figure 9.3).

Using the GridView.count

The `GridView.count` requires setting the `crossAxisCount` and the `children` argument. The `crossAxisCount` sets the number of tiles to display (Figure 9.4), and `children` is a list of widgets. The `scrollDirection` argument sets the main axis direction for the `Grid` to scroll, either `Axis.vertical` or `Axis.horizontal`, and the default is `vertical`.

For the `children`, you use the `List.generate` to create your sample data, a list of values. Within the `children` argument, I added a `print` statement to show that the entire list of values is built at the same time, not just the visible rows like the `GridView.builder`. Note for the following sample code, 7,000 records are generated to show that the `GridView.count` does not show any data until all of the records are processed first.

```
GridView.count(
  crossAxisCount: 3,
  padding: EdgeInsets.all(8.0),
```

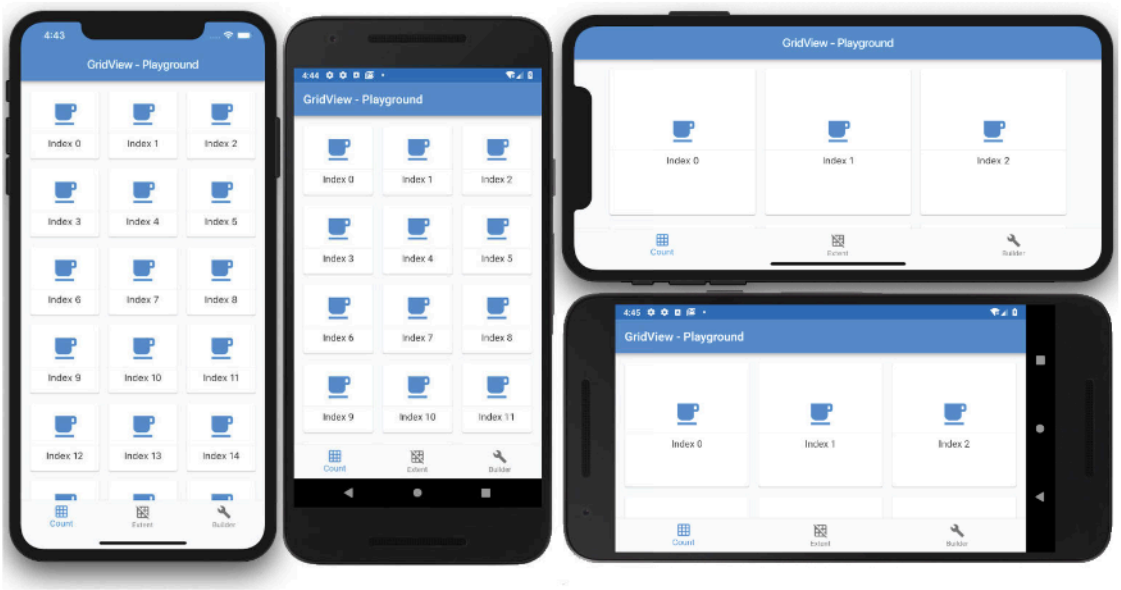


FIGURE 9.4: GridView count with three tiles in portrait and landscape mode

```
children: List.generate(7000, (index) {
  print('_buildGridView $index');

  return Card(
    margin: EdgeInsets.all(8.0),
    child: InkWell(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Icon(
            _iconList[0],
            size: 48.0,
            color: Colors.blue,
          ),
          Divider(),
          Text(
            'Index $index',
            textAlign: TextAlign.center,
            style: TextStyle(
              fontSize: 16.0,
            ),
          )
        ],
      ),
      onTap: () {
        print('Row $index');
      },
    ),
  );
},),
)
```

Using the GridView.extent

The `GridView.extent` requires you to set the `maxCrossAxisExtent` and `children` argument. The `maxCrossAxisExtent` argument sets the maximum size of each tile for the axis. For example, in portrait, it can fit two to three tiles, but when rotating to landscape, it can fit five to six depending on the screen size (Figure 9.5). The `scrollDirection` argument sets the main axis direction for the grid to scroll, either `Axis.vertical` or `Axis.horizontal`, and the default is `vertical`.

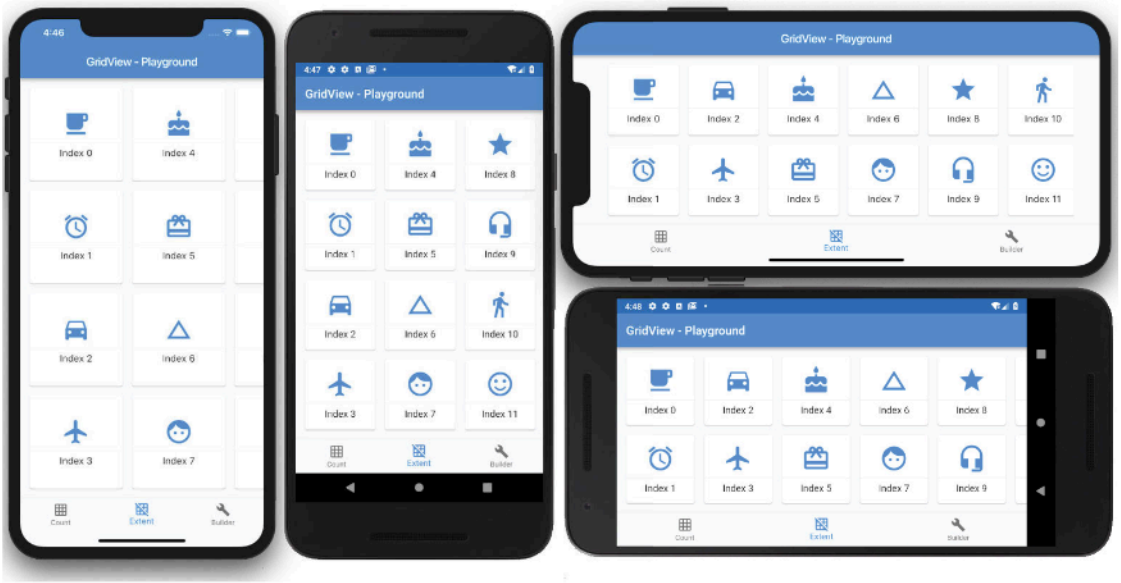


FIGURE 9.5: GridView extent showing the maximum number of tiles that can fit according to screen size

For the `children`, you use `List.generate` to create your sample data, which is a list of values. Within the `children` argument, I added a `print` statement to show that the entire list of values is built at the same time, not just the visible rows like the `GridView.builder`.

```
GridView.extent(
  maxCrossAxisExtent: 175.0,
  scrollDirection: Axis.horizontal,
  padding: EdgeInsets.all(8.0),
  children: List.generate(20, (index) {
    print('_buildGridViewExtent $index');

    return Card(
      margin: EdgeInsets.all(8.0),
      child: InkWell(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Icon(
              _iconList[index],
              size: 48.0,
              color: Colors.blue,
            ),
```

```
            Divider(),
            Text(
              'Index $index',
              textAlign: TextAlign.center,
              style: TextStyle(
                fontSize: 16.0,
              ),
            )
          ],
        ),
        onTap: () {
          print('Row $index');
        },
      ),
    );
  }),
)
```

Using the GridView.builder

The `GridView.builder` requires you to set the `itemCount`, `gridDelegate`, and `itemBuilder` arguments. The `itemCount` sets the number of tiles to build. The `gridDelegate` is a `SilverGridDelegate` responsible for laying out the children list of widgets for the `GridView`. The `gridDelegate` argument cannot be null; you need to pass the `maxCrossAxisExtent` size, for example, 150.0 pixels.

For example, to display three tiles across the screen you specify the `gridDelegate` argument with the `SliverGridDelegateWithFixedCrossAxisCount` class to create a grid layout with a fixed number of tiles for the cross axis. If you need to display tiles that have a maximum width of 150.0 pixels, you specify the `gridDelegate` argument with the `SliverGridDelegateWithMaxCrossAxisExtent` class to create a grid layout with tiles that have a maximum cross-axis extent, the maximum width of each tile.

The `GridView.builder` is used when you have a large set of data because the `builder` is called only for visible tiles, which is great for performance. Using the `GridView.builder` constructor results in lazily building a list for visible tiles, and when the user scrolls to the next visible tiles they are lazily built as needed.

TRY IT OUT Creating the GridView.builder App

In this example, the `GridView` widget uses the `builder` to display a `Card` that shows each `Grid` item with an `Icon` and a `Text` showing the index location. The `onTap` will print the index of the tapped `Grid` item.

1. Create a new Flutter project and name it `ch9_gridview`, following the instructions in Chapter 4. For this project, you need to create only the `pages`, `classes`, and `widgets` folders. Create the `Home Class` as a `StatelessWidget` since the data does not require changes.

2. Open the `home.dart` file and add to the `body` a `SafeArea` with the `GridViewBuilderWidget()` widget class as a `child`.

```
body: SafeArea(
  child: const GridViewBuildWidget(),
),
```

3. Add to the top of the file the `import` statement for the `gridview_builder.dart` widget class that you'll create next.

```
import 'package:flutter/material.dart';
import 'package:ch9_gridview/widgets/gridview_builder.dart';
```

4. Create a new Dart file in the `widgets` folder. Right-click the `classes` folder and then select New ➪ Dart File, enter `grid_icons.dart`, and click the OK button to save.

5. Import the `material.dart` library, add a new line, and create the `GridIcons Class`. The `GridIcons Class` holds a `List` of `IconData` called `iconList`.

6. Create the `getIconList()` method that creates the `List` of `IconData` that is used later in the `GridView.builder`.

```
class GridIcons {
  List<IconData> iconList = [];

  List<IconData> getIconList() {
    iconList
      ..add(Icons.free_breakfast)
      ..add(Icons.access_alarms)
      ..add(Icons.directions_car)
      ..add(Icons.flight)
      ..add(Icons.cake)
      ..add(Icons.card_giftcard)
      ..add(Icons.change_history)
      ..add(Icons.face)
      ..add(Icons.star)
      ..add(Icons.headset_mic)
      ..add(Icons.directions_walk)
      ..add(Icons.sentiment_satisfied)
      ..add(Icons.cloud_queue)
      ..add(Icons.exposure)
      ..add(Icons.gps_not_fixed)
      ..add(Icons.child_friendly)
      ..add(Icons.child_care)
      ..add(Icons.edit_location)
      ..add(Icons.event_seat)
      ..add(Icons.lightbulb_outline);
    return iconList;
  }
}
```

7. Create a new Dart file under the `widgets` folder. Right-click the `widgets` folder and then select New ➪ Dart File, enter `gridview_builder.dart`, and click the OK button to save.

8. Import the `material.dart` library, add a new line, and then start typing st; the autocompletion help opens, so select the `stless` (`StatelessWidget`) abbreviation and give it a name of `GridViewBuilderWidget`.

9. Modify the `GridViewBuilderWidget` widget class to return a `GridView.builder` with the `itemCount` argument set to 20. For this example, you specify to list 20 rows of data.

10. For the `gridDelegate` argument, use `SliverGridDelegateWithMaxCrossAxisExtent(maxCrossAxisExtent: 150.0)`. Your other option is to use the `SliverGridDelegateWithFixedCrossAxisCount` instead, which works in the same manner as the `GridView.count` constructor, where you pass the number of tiles to display.

11. For the `itemBuilder` callback, it passes the `BuildContext` and the widget `index` as an `int` value. In the `itemBuilder` first line, place a `print` statement to show each item index being built according to the visible space.

12. Return a `Card` with the `child` as an `InkWell`. The `InkWell onTap` has a `print` statement to show the tapped `Card` item, with the `Row` selected.

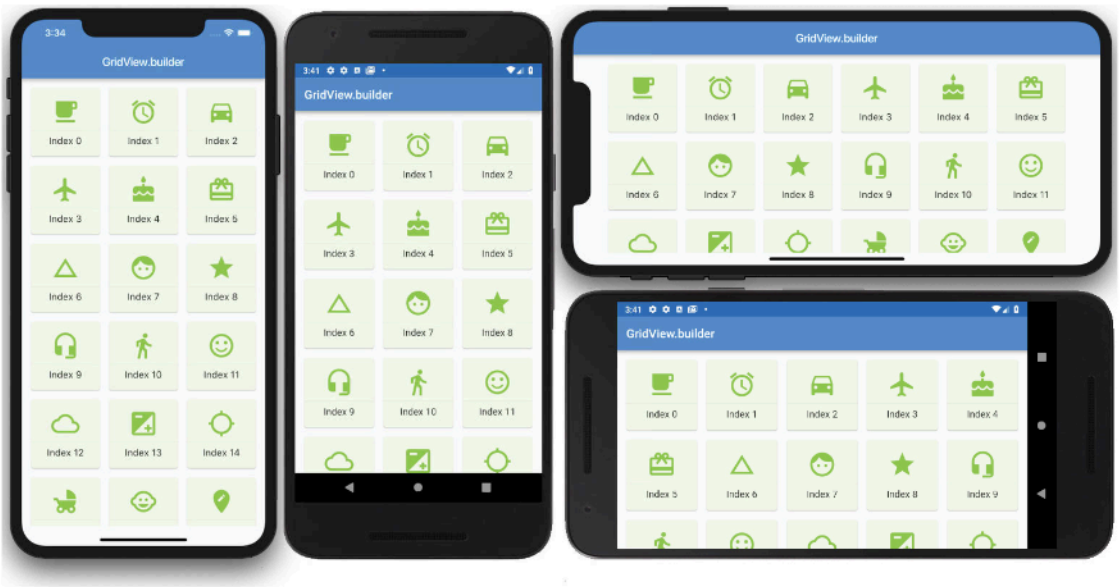
13. For the `InkWell child` property, pass a `Column` with these `children`: `Icon`, `Divider`, and `Text` widgets. Note that the `itemBuilder` is called for each row item. For every row, a `Widget` is created and added to the widget tree.

    The `onTap` will print the index of the tapped `Grid` item.

14. Add to the top of the file the `import` statement for the `grid_icons.dart` class.

```
import 'package:flutter/material.dart';
import 'package:ch9_gridview/classes/grid_icons.dart';

class GridViewBuilderWidget extends StatelessWidget {
  const GridViewBuilderWidget({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    List<IconData> _iconList = GridIcons().getIconList();

    return GridView.builder(
      itemCount: 20,
      padding: EdgeInsets.all(8.0),
      gridDelegate: SliverGridDelegateWithMaxCrossAxisExtent(maxCrossAxisExtent:
150.0),
      itemBuilder: (BuildContext context, int index) {
        print('_buildGridViewBuilder $index');

        return Card(
          color: Colors.lightGreen.shade50,
          margin: EdgeInsets.all(8.0),
          child: InkWell(
            child: Column(
              mainAxisAlignment: MainAxisAlignment.center,
              children: <Widget>[
                Icon(
                  _iconList[index],
                  size: 48.0,
                  color: Colors.lightGreen,
                ),
                Divider(),
                Text(
```

```
                    'Index $index',
                    textAlign: TextAlign.center,
                    style: TextStyle(
                      fontSize: 16.0,
                    ),
                  )
                ],
              ),
              onTap: () {
                print('Row $index');
              },
            ),
          );
        },
      );
    }
  }
```



HOW IT WORKS

The `GridView.builder` constructor takes an `itemCount` and uses the `itemBuilder` to build a widget for each child record. Each child widget is added to the widget tree with the appropriate values. As each child widget is added, you can customize the rows according to app specs. In this example, you used a `Card` to give each `Grid Row` item a nice look and used an `InkWell` to use the Material Design tap animation and the `onTap` property. The `InkWell child` is a `Column`, and its `children` items show an `Icon` and `Text`.

USING THE STACK

The `Stack` widget is commonly used to overlap, position, and align widgets to create a custom look. A good example is a shopping cart with the number of items to purchase on the upper-right side. The `Stack` `children` list of the widget is either positioned or nonpositioned. When you use a `Positioned` widget, each `child` widget is placed at the appropriate location.

The `Stack` widget resizes itself to accommodate all of the nonpositioned children. The nonpositioned children are positioned to the `alignment` property (`Top-Left` or `Top-Right` depending on the left-to-right or right-to-left environment). Each `Stack` child widget is drawn in order from bottom to top, like stacking pieces of paper on top of each other. This means the first widget drawn is at the bottom of the `Stack`, and then the next widget is drawn above the previous widget and so on. Each child widget is positioned on top of each other in the order of the `Stack` `children` list. The `RenderStack` class handles the stack layout.

To align each child in the `Stack`, you use the `Positioned` widget. By using the `top`, `bottom`, `left`, and `right` properties, you align each child widget within the `Stack`. The `height` and `width` properties of the `Positioned` widget can also be set (Figure 9.6).

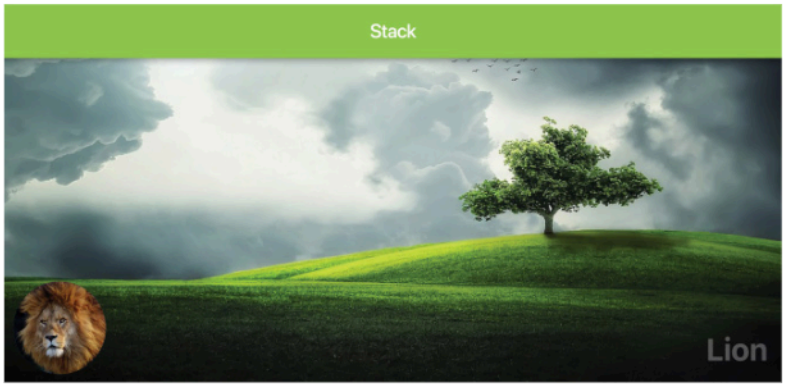


FIGURE 9.6: Stack layout showing Image and Text widgets stacked over the background image

You'll also learn how to implement the `FractionalTranslation` class to position a widget fractionally outside the parent widget. You set the `translation` property with the `Offset(dx, dy)` (double type value for x-and y-axis) class that's scaled to the child's size, resulting in moving and positioning the widget. For example, to show a favorite icon moved a third of the way to the upper right of the parent widget, you set the `translation` property with the `Offset(0.3, -0.3)` value.

The following example (Figure 9.7) shows a `Stack` widget with a background image, and by using the `FractionalTranslation` class, you set the `translation` property to the `Offset(0.3, -0.3)` value, placing the star icon one-third to the right of the x-axis and a negative one-third (move icon upward) on the y-axis.

```
Stack(
  children: <Widget>[
    Image(image: AssetImage('assets/images/dawn.jpg')),
    Positioned(
```

```
          top: 0.0,
          right: 0.0,
          child: FractionalTranslation(
            translation: Offset(0.3, -0.3),
            child: CircleAvatar(
              child: Icon(Icons.star),
            ),
          ),
        ),
        Positioned(/* Eagle Image */),
        Positioned(/* Bald Eagle */),
      ],
    ),
```

FIGURE 9.7: FractionalTranslation class showing favorite icon moved to the upper right

TRY IT OUT Creating the Stack App

In this example, the `Stack` widget children list of widgets lays out a background `Image` and two `Positioned` widgets with `CircleAvatar` and `Text` widgets. To show an alternate layout, you use the same previous `Stack` layout and add a `Positioned` widget with the `child` property as a `FractionalTranslation` class to show a `CircleAvatar` pinned to the upper-right corner halfway outside the `Stack`. A `ListView` is used to create the sample list, and each row displays an alternate `Stack` widget.

1. Create a new Flutter project and name it `ch9_stack`. Again, follow the instructions in Chapter 4. For this project, you need to create the `pages`, `widgets`, and `assets/images` folders. Create the `Home Class` as a `StatelessWidget` since your data does not require changes.
2. Open the `pubspec.yaml` file to add resources. In the `assets` section, add the `assets/images/` folder.

```
# To add assets to your application, add an assets section, like this:
assets:
  - assets/images/
```

3. Click the Save button, and depending on the editor you are using, it automatically runs the `flutter packages get`, and once finished, it will show a message of `Process finished with exit code 0`. If it does not automatically run the command for you, open the Terminal window (located at the bottom of your editor) and type `flutter packages get`.

4. Add the folder `assets` and subfolder `images` at the project's root, and then copy the `dawn.jpg`, `eagle.jpg`, `lion.jpg`, and `tree.jpg` files to the `images` folder.

5. Open the `home.dart` file and add to the `body` a `SafeArea` with `ListView.builder()` as a `child`.

```
body: SafeArea(
  child: ListView.builder(),
),
```

6. Add to the top of the file the `import` statement for the `stack.dart` and `stack_favorite.dart` widget classes that you'll create next.

```
import 'package:flutter/material.dart';
import 'package:ch9_stack/widgets/stack.dart';
import 'package:ch9_stack/widgets/stack_favorite.dart';
```

7. Add to the `ListView.builder` the `itemCount` argument with a value set to 7. For this example, you specify to list seven rows of data. For the `itemBuilder` callback, it passes the `BuildContext` and the widget `index` as an `int` value.

   With each `Stack` layout, you alternate between them by checking whether the index value is even or odd and then call the widget classes `StackWidget()` and `StackFavoriteWidget()`, respectively.

   I wanted to show that you can customize which widgets you present to the user. Let's say you have an app that you distribute as freeware and every ten records you show an advertisement or a tip embedded in the list. This technique is not as intrusive as a pop-up while the user views records.

```
body: SafeArea(
  child: ListView.builder(
    itemCount: 7,
    itemBuilder: (BuildContext context, int index) {
      if (index.isEven) {
        return const StackWidget();
      } else {
        return const StackFavoriteWidget();
      }
    },
  ),
),
```

8. Create a new Dart file in the `widgets` folder. Right-click the `widgets` folder and then select New ➪ Dart File, enter `stack.dart`, and click the OK button to save.

9. Import the `material.dart` library, add a new line, and then start typing st; the autocompletion help opens, where you can select the `stless` (`StatelessWidget`) abbreviation and give it a name of `StackWidget`.

10. Modify the `StackWidget` widget class to return a `Stack`. The `Stack` `children` list of widgets consists of an `Image` with the `AssetImage` `tree.jpg`.

11. Add a `Positioned` widget with `bottom` and `left` properties with a value of `10.0`. The `child` is a `CircleAvatar` with a `radius` of `48.0` and has the `backgroundImage` property set to `AssetImage` `lion.jpg`.

12. Add another `Positioned` widget with `bottom` and `right` properties with a value of `16.0`. The `child` is a `Text` widget with a string value of `Lion`, and it has a `style` property with a `TextStyle` class with a `fontSize` set to `32.0` pixels, `color` set to `white30`, and `fontWeight` set to `bold`.

```
import 'package:flutter/material.dart';

class StackWidget extends StatelessWidget {
  const StackWidget({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Stack(
      children: <Widget>[
        Image(
          image: AssetImage('assets/images/tree.jpg'),
        ),
        Positioned(
          bottom: 10.0,
          left: 10.0,
          child: CircleAvatar(
            radius: 48.0,
            backgroundImage: AssetImage('assets/images/lion.jpg'),
          ),
        ),
        Positioned(
          bottom: 16.0,
          right: 16.0,
          child: Text(
            'Lion',
            style: TextStyle(
              fontSize: 32.0,
              color: Colors.white30,
              fontWeight: FontWeight.bold,
            ),
          ),
        ),
      ],
    );
  }
}
```

13. Create a new Dart file in the `widgets` folder. Right-click the `widgets` folder and then select New ➪ Dart File, enter `stack_favorite.dart`, and click the OK button to save.

14. Import the `material.dart` library, add a new line, and then start typing st; autocompletion help opens, so select the `stless` (`StatelessWidget`) abbreviation and give it a name of `StackFavoriteWidget`.

15. Modify the `StackFavoriteWidget` widget class to return a `Container`. Use a `Container` to set `color` to `black87` and for the `child` use a `Padding` with `EdgeInsets.all(16.0)`. This will create a dark frame effect around the `Stack`.

16. For the `Stack children` list of widgets, add an `Image` set to `AssetImage dawn.jpg`.

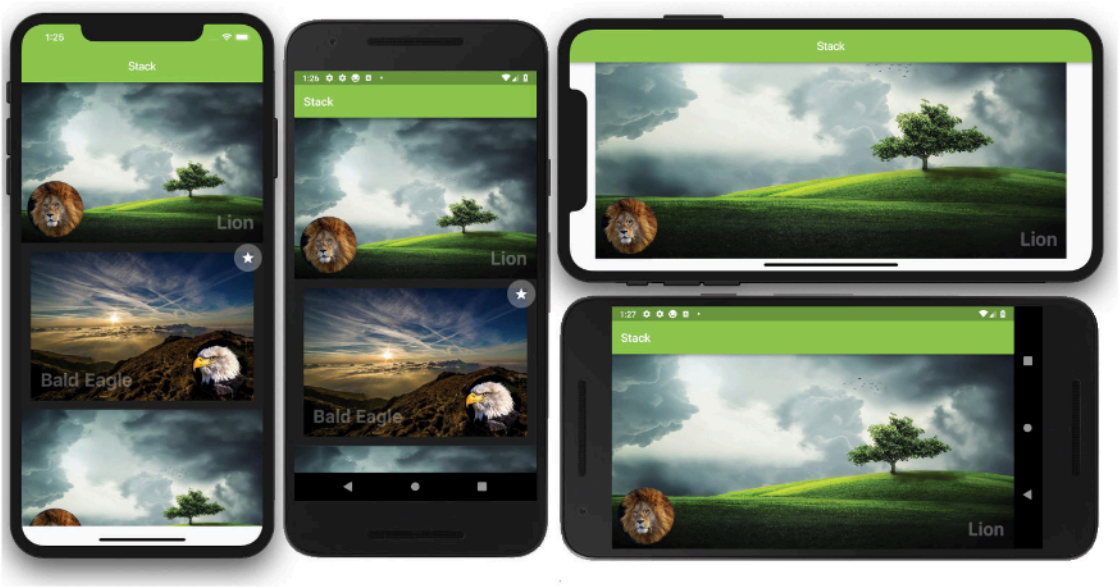
17. Add a `Positioned` widget with `bottom` and `right` properties with the values `0.0`. The `child` is a `FractionalTranslation` class with a `translation` property of `Offset(0.3,-0.3)`. The `child` is a `CircleAvatar`, and by using the `Offset`, it shows it pinned to the upper-right corner halfway outside of the `Stack`.

18. Add a `Positioned` widget with `bottom` and `right` properties with a value of `10.0`. The `child` is a `CircleAvatar` with a `radius` of `48.0` and `backgroundImage` with the `AssetImage eagle.jpg`.

19. Add another `Positioned` widget with `bottom` and `right` properties that have values of `16.0`. The `child` is a `Text` widget with a string value of `Bald Eagle`, and it has a `style` property with a `TextStyle` with `fontSize` set to `32.0` pixels, `color` set to `white30`, and `fontWeight` set to `bold`.

```
import 'package:flutter/material.dart';

class StackFavoriteWidget extends StatelessWidget {
  const StackFavoriteWidget({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Container(
      color: Colors.black87,
      child: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Stack(
          children: <Widget>[
            Image(
              image: AssetImage('assets/images/dawn.jpg'),
            ),
            Positioned(
              top: 0.0,
              right: 0.0,
              child: FractionalTranslation(
                translation: Offset(0.3, -0.3),
                child: CircleAvatar(
                  radius: 24.0,
                  backgroundColor: Colors.white30,
                  child: Icon(
                    Icons.star,
                    size: 24.0,
                    color: Colors.white,
                  ),
                ),
              ),
            ),
            Positioned(
              bottom: 10.0,
```

```
                right: 10.0,
                child: CircleAvatar(
                  radius: 48.0,
                  backgroundImage: AssetImage('assets/images/eagle.jpg'),
                ),
              ),
              Positioned(
                bottom: 16.0,
                left: 16.0,
                child: Text(
                  'Bald Eagle',
                  style: TextStyle(
                    fontSize: 32.0,
                    color: Colors.white30,
                    fontWeight: FontWeight.bold,
                  ),
                ),
              ),
            ],
          ),
        ),
      );
    }
  }
```



HOW IT WORKS

The `Stack` takes a `children` list of widgets and sizes itself to accommodate all of the nonpositioned widgets. When you use nonpositioned widgets in the `Stack`, they are automatically positioned to the setting of `alignment` (`Top-Left` or `Top-Right` depending on environment). Each `Stack` child widget is drawn in order from bottom to top, meaning each child widget is positioned on top of each other.

Using the `Positioned` widget allows each child widget to be aligned by using the `top`, `bottom`, `left`, and `right` properties. You learned how to position the favorite icon to the upper right of the parent widget by using the `FractionalTranslation` class's `translation` property with the `Offset(0.3,-03)` value.

CUSTOMIZING THE CUSTOMSCROLLVIEW WITH SLIVERS

The `CustomScrollView` widget creates custom scrolling effects by using a list of slivers. Slivers are a small portion of something larger. For example, slivers are placed inside a view port like the `CustomScrollView` widget. In the previous sections, you learned how to implement the `ListView` and `GridView` widgets separately. But what if you needed to present them together in the same list? The answer is that you can use a `CustomScrollView` with the `slivers` property list of widgets set to the `SliverSafeArea`, `SliverAppBar`, `SliverList`, and `SliverGrid` widgets (slivers). The order in which you place them in the `CustomScrollView` `slivers` property is the order in which they are rendered. Table 9.1 shows commonly used slivers and sample code.

TABLE 9.1: Slivers

SLIVER	DESCRIPTION	CODE

`SliverSafeArea`	Adds padding to avoid the device notch usually located on the top of the screen	`SliverSafeArea(`
		`  sliver: SliverGrid(),`
		`)`

`SliverAppBar`	Adds an app bar	`SliverAppBar(`
		`  expandedHeight: 250.0,`
		`  flexibleSpace: FlexibleSpaceBar(`
		`  title: Text('Parallax'),`
		`  ),`
		`)`

`SliverList`	Creates a linear scrollable list of widgets	`SliverList(`
		`  delegate: SliverChildListDelegate(`
		`  List.generate(3, (int index) {`
		`  return ListTile();`
		`  }),`
		`  ),`
		`)`

`SliverGrid`	Displays tiles of scrollable widgets in a grid format	`SliverGrid(`
		`  delegate: SliverChildBuilderDelegate(`
		`  (BuildContext context, int index) {`
		`  return Card();`
		`  },`
		`  childCount: _rowsCount,`
		`  ),`
		`  gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(crossAxisCount: 3),`
		`)`

The `SliverList` and `SliverGrid` slivers use delegates to build the list of children explicitly or lazily. An explicit list builds all of the items first then displays them on the screen. A lazily built list only builds the visible items on the screen and when the user scrolls the next visible items are built (lazily) resulting in better performance. The `SliverList` has a `delegate` property and the `SliverGrid` has a `delegate` and a `gridDelegate` property.

The `SliverList` and `SliverGrid` `delegate` property can use the `SliverChildListDelegate` to build an explicit list or use the `SliverChildBuilderDelegate` to lazily build the list. The `SliverGrid` has an additional `gridDelegate` property to specify the size and position of the grid tiles. Specify the `gridDelegate` property with the `SliverGridDelegateWithFixedCrossAxisCount` class to create a grid layout with a fixed number of tiles for the cross axis; for example, show three tiles across. Specify the `gridDelegate` property with the `SliverGridDelegateWithMaxCrossAxisExtent` class to create a grid layout with tiles that have a maximum cross-axis extent, the maximum width of each tile; for example, 150.0 pixels maximum width for each tile.

Table 9.2 shows the `SliverList` and `SliverGrid` delegates to help you build lists.

TABLE 9.2: Sliver Delegates

SLIVER	DESCRIPTION	CODE

`SliverList`	`SliverChildListDelegate` builds a list of known number of rows (explicit). `SliverChildBuilderDelegate` lazily builds a list of unknown number of rows.	`SliverList(`
		`  delegate:`
		`SliverChildListDelegate(<Widget>[`
		`  ListTile(title: Text('One')),`
		`  ListTile(title: Text('Two')),`
		`  ListTile(title: Text('Three')),`
		`  ]),`
		`)`
		or
		`SliverList(`
		`  delegate: SliverChildListDelegate(`
		`  List.generate(30, (int index) {`
		`  return ListTile();`
		`  }),`
		`  ),`
		`)`
		or
		`SliverList(`
		`  delegate: SliverChildBuilderDeleg`
		`ate((BuildContext context,`
		`int index) {`
		`  return ListTile();`
		`  },`
		`  childCount: _rowsCount,`
		`  ),`
		`)`

SLIVER	DESCRIPTION	CODE

`SliverGrid`	`SliverChildListDelegate` builds an explicit list. `SliverChildBuilderDelegate` lazily builds a list of unknown number of tiles. The `gridDelegate` property controls the position and size of the children widgets.	`SliverGrid(`
		`  delegate: SliverChildListDelegate-`
		`(<Widget>[`
		`  Card(),`
		`  Card(),`
		`  Card(),`
		`  ]),`
		`  gridDelegate:`
		`SliverGridDelegateWithFixedCrossAxis-`
		`Count(crossAxisCount: 3),`
		`)`
		or
		`SliverGrid(`
		`  delegate: SliverChildListDelegate(`
		`  List.generate(30, (int index) {`
		`  return Card();`
		`  }),`
		`  ),`
		`  gridDelegate:`
		`SliverGridDelegateWithFixedCross-`
		`AxisCount(crossAxisCount: 3),`
		`)`
		or
		`SliverChildBuilderDelegate(`
		`  (BuildContext context,`
		`int index) {`
		`  return Card();`
		`  },`
		`  childCount: _rowsCount,`
		`  ),`
		`  gridDelegate:`
		`  SliverGridDelegateWithFixedCross-`
		`AxisCount(crossAxisCount: 3),`
		`)`

The `SliverAppBar` widget can have a parallax (Figure 9.8) effect by using the `expandedHeight` and `flexibleSpace` properties. The parallax effect scrolls a background image slower than the content in the foreground. If you need to show the `CustomScrollView` initially scrolled at a particular position, use a controller and set the `ScrollController.initialScrollOffset` property. For example, you would set the `initialScrollOffset` by initializing the `controller = ScrollController(initialScrollOffset: 10.0)`.

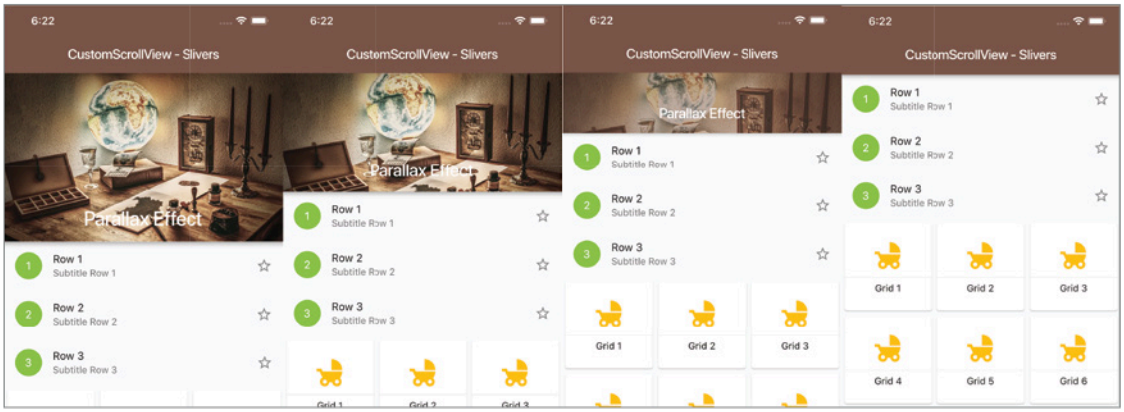


FIGURE 9.8: SliverAppBar scrolling parallax effect

TRY IT OUT Creating the CustomScrollView Slivers App

In this example, the `CustomScrollView` children list of widgets contains a `SliverAppBar`, `SliverList`, `SliverSafeArea`, and `SliverGrid`. The `SliverAppBar` widget uses the `flexibleSpace` with a background `Image` that has a parallax effect while scrolling. The `SliverList` generates three items with the `List.generate` constructor. To account for the device notch, you use a `SliverSafeArea` to wrap the `SliverGrid`, and you generate 12 (sample value can be more or less) items.

1. Create a new Flutter project and name it `ch9_customscrollview_slivers`; you can follow the instructions in Chapter 4. For this project, you only need to create the `pages` and `assets/images` folders. Create the `Home Class` as a `StatelessWidget` since the data does not require changes.

2. Open the `pubspec.yaml` file to add resources. In the `assets` section, add the `assets/images/` folder.

```
# To add assets to your application, add an assets section, like this:
assets:
  - assets/images/
```

3. Click the Save button, and depending on the editor you are using, it automatically runs `flutter packages get`. Once finished, it will show a message of `Process finished with exit code 0`. If it does not automatically run the command for you, open the Terminal window (located at the bottom of your editor) and type `flutter packages get`.

4. Add the folder `assets` and subfolder `images` at the project's root and then copy the `desk.jpg` file to the `images` folder.

5. Open the `home.dart` file and add to the `body` a `CustomScrollView()`. For this project, set the `AppBar elevation` property to `0.0` because you enable the `SliverAppBar` shadow instead.

```
return Scaffold(
  appBar: AppBar(
    title: Text('CustomScrollView - Slivers'),
    elevation: 0.0,
  ),
```

```
    body: CustomScrollView(
      slivers: <Widget>[
      ],
    ),
  );
```

6. Add to the top of the file the `import` statement for the `sliver_app_bar.dart`, `sliver_list.dart`, and `sliver_grid.dart` widget classes that you'll create next.

```
import 'package:flutter/material.dart';
import 'package:ch9_customscrollview_slivers/widgets/sliver_app_bar.dart';
import 'package:ch9_customscrollview_slivers/widgets/sliver_list.dart';
import 'package:ch9_customscrollview_slivers/widgets/sliver_grid.dart';
```

7. Add calls to the `SliverAppBarWidget()`, `SliverListWidget()`, and `SliverGridWidget()` widget classes to the `CustomScrollView() slivers` property. Make sure the calls to the widget classes use the `const` keyword to take advantage of caching for performance gain.

```
return Scaffold(
  appBar: AppBar(
    title: Text('CustomScrollView - Slivers'),
    elevation: 0.0,
  ),
  body: CustomScrollView(
    slivers: <Widget>[
      const SliverAppBarWidget(),
      const SliverListWidget(),
      const SliverGridWidget(),
    ],
  ),
);
```

8. Create a new Dart file in the `widgets` folder. Right-click the `widgets` folder and then select New ➪ Dart File, enter `sliver_app_bar.dart`, and click the OK button to save.

9. Import the `material.dart` library, add a new line, and then start typing st; the autocompletion help opens, so select the `stless` (`StatelessWidget`) abbreviation and give it a name of `SliverAppBarWidget`.

10. Modify the `SliverAppBarWidget` widget class to return a `SliverAppBar`.

11. To show a shadow at the bottom of the bar, set the `forceElevated` property to `true`.

12. To create a parallax effect while scrolling, set `expandedHeight` to `250.0` pixels and `flexibleSpace` to `FlexibleSpaceBar`.

13. For the `FlexibleSpaceBar background` property, use the `Image` widget with the `desk.jpg` file and set `fit` to `BoxFit.cover`.

```
import 'package:flutter/material.dart';

class SliverAppBarWidget extends StatelessWidget {
  const SliverAppBarWidget({
    Key key,
  }) : super(key: key);
```

```
  @override
  Widget build(BuildContext context) {
    return SliverAppBar(
      backgroundColor: Colors.brown,
      forceElevated: true,
      expandedHeight: 250.0,
      flexibleSpace: FlexibleSpaceBar(
        title: Text(
          'Parallax Effect',
        ),
        background: Image(
          image: AssetImage('assets/images/desk.jpg'),
          fit: BoxFit.cover,
        ),
      ),
    );
  }
}
```

14. Create a new Dart file in the `widgets` folder. Right-click the `widgets` folder and then select New ➪ Dart File, enter `sliver_list.dart`, and click the OK button to save.

15. Import the `material.dart` library, add a new line, and then start typing st; the autocompletion help opens, so select the `stless` (`StatelessWidget`) abbreviation and give it a name of `SliverListWidget`.

16. Modify the `SliverListWidget` widget class to return a `SliverList`. For the `SliverList` `delegate` property, pass the `SliverChildListDelegate`.

17. Use the `List.generate` constructor to build your sample data list. The constructor takes two arguments: the `length` of the list and the `index`. Return a `ListTile` with a `leading` `CircleAvatar` with the `child` as a `Text` widget with string interpolation set with `${index + 1}`.

18. Also, set the `ListTile` `title`, `subtitle`, and `trailing` properties.

```
import 'package:flutter/material.dart';

class SliverListWidget extends StatelessWidget {
  const SliverListWidget({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return SliverList(
      delegate: SliverChildListDelegate(
        List.generate(3, (int index) {
          return ListTile(
            leading: CircleAvatar(
              child: Text("${index + 1}"),
              backgroundColor: Colors.lightGreen,
              foregroundColor: Colors.white,
            ),
            title: Text('Row ${index + 1}'),
```

```
            subtitle: Text('Subtitle Row ${index + 1}'),
            trailing: Icon(Icons.star_border),
          );
        }),
      ),
    );
  }
}
```

19. Create a new Dart file in the `widgets` folder. Right-click the `widgets` folder and then select New ➪ Dart File, enter `sliver_grid.dart`, and click the OK button to save.

20. Import the `material.dart` library, add a new line, and then start typing st; the autocompletion help opens, so select the `stless` (`StatelessWidget`) abbreviation and give it a name of `SliverGridWidget`.

21. Modify the `SliverGridWidget` widget class to return a `SliverSafeArea`. Since the `SliverGrid` does not handle the device notch automatically, you wrap it in a `SliverSafeArea`. The `SliverGrid` `delegate` property is a `SliverChildBuilderDelegate` taking the `BuildContext` and `int index`.

22. From the `SliverChildBuilderDelegate`, return a `Card` with the `child` as a `Column`. The `Column` children list of `Widget` has `Icon`, `Divider`, and `Text` widgets.

23. For the `childCount` property, pass `12` representing how many items the builder creates. The `gridDelegate` property is set to `SliverGridDelegateWithFixedCrossAxisCount(cross-AxisCount: 3)` showing three tiles.

```
import 'package:flutter/material.dart';

class SliverGridWidget extends StatelessWidget {
  const SliverGridWidget({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return SliverSafeArea(
      sliver: SliverGrid(
        delegate: SliverChildBuilderDelegate(
              (BuildContext context, int index) {
            return Card(
              child: Column(
                mainAxisAlignment: MainAxisAlignment.center,
                children: <Widget>[
                  Icon(Icons.child_friendly, size: 48.0, color: Colors.amber,),
                  Divider(),
                  Text('Grid ${index + 1}'),
                ],
              ),
            );
          },
          childCount: 12,
        ),
        gridDelegate:
```

```
            SliverGridDelegateWithFixedCrossAxisCount(crossAxisCount: 3),
          ),
        );
      }
    }
```

HOW IT WORKS

To create a custom scrolling effect, the `CustomScrollView` uses a list of `slivers`. You used the `SliverAppBar` to create a parallax scrolling effect by using a `FlexibleSpaceBar`. You also created a `SliverList` and set the `delegate` property to the `SliverChildListDelegate` class. To handle the device notch, the SliverGrid is wrapped in a `SliverSafeArea` widget. The `SliverGrid delegate` property uses the `SliverChildBuilderDelegate`, which takes a `BuildContext` and `int index`.

SUMMARY

In this chapter, you learned to use the `Card` to group information with the container having rounded corners and a shadow. You used the `ListView` to build a list of scrollable widgets and to align grouped data with the `ListTile`, and you used the `GridView` to display data in tiles, using the `Card` to group the data. You embedded a `Stack` in a `ListView` to show an `Image` as a background and stacked different widgets with the `Positioned` widget to overlap and position them at the appropriate locations by using the `top`, `bottom`, `left`, and `right` properties.

In the next chapter, you'll learn to build custom layouts by using `SingleChildScrollView`, `SafeArea`, `Padding`, `Column`, `Row`, `Image`, `Divider`, `Text`, `Icon`, `SizedBox`, `Wrap`, `Chip`, and `CircleAvatar`. You'll learn to take a high-level view as well as a detailed view to separate and nest widgets to create a custom UI.

▶ WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEY CONCEPTS
`Card`	This groups and organizes information into a container with rounded corners and casts a drop shadow.
`ListView` and `ListTile`	This is a linear list of scrollable widgets with either vertical or horizontal scrolling. To easily format how the list of records is displayed in rows, you took advantage of the `ListTile` widget to align grouped data with leading and trailing icons.
`GridView`	This displays tiles of scrollable widgets in a grid format. Scrolling can be vertical or horizontal.
`Stack`	This is commonly used to overlap, position, and align its children widgets to create a custom look.
`CustomScrollView` and slivers	This allows you to create custom scrolling effects by using a list of slivers widgets like a `SliverSafeArea`, `SliverAppBar`, `SliverList`, `SliverGrid`, and more.