# Flutter Development

MNNIT Computer Coding Club

## Objective

To make you answer the following questions yourselves:

1. How stateful widget works?
2. How to draw a widget tree?
3. How can you layout a widget?
4. What is the difference between a responsive app and an adaptive app?
5. How to build a responsive app?
6. How to build an adaptive app?
7. How Navigation works in Flutter?

## Stateless widgets

1. Stateless Widgets in Flutter are those widgets whose state once created cannot be changed.
2. In simple words, if a widget doesn’t do anything it is a Stateless Widget. They are static in nature.
3. We use it when the UI relies on the information inside the object itself.
4. You can also say that stateless widgets are “DATALESS” widgets. As they don’t store any real-time data.
5. For example, if you have a simple Text widget on the screen, but it doesn’t do anything then its Stateless Widget.

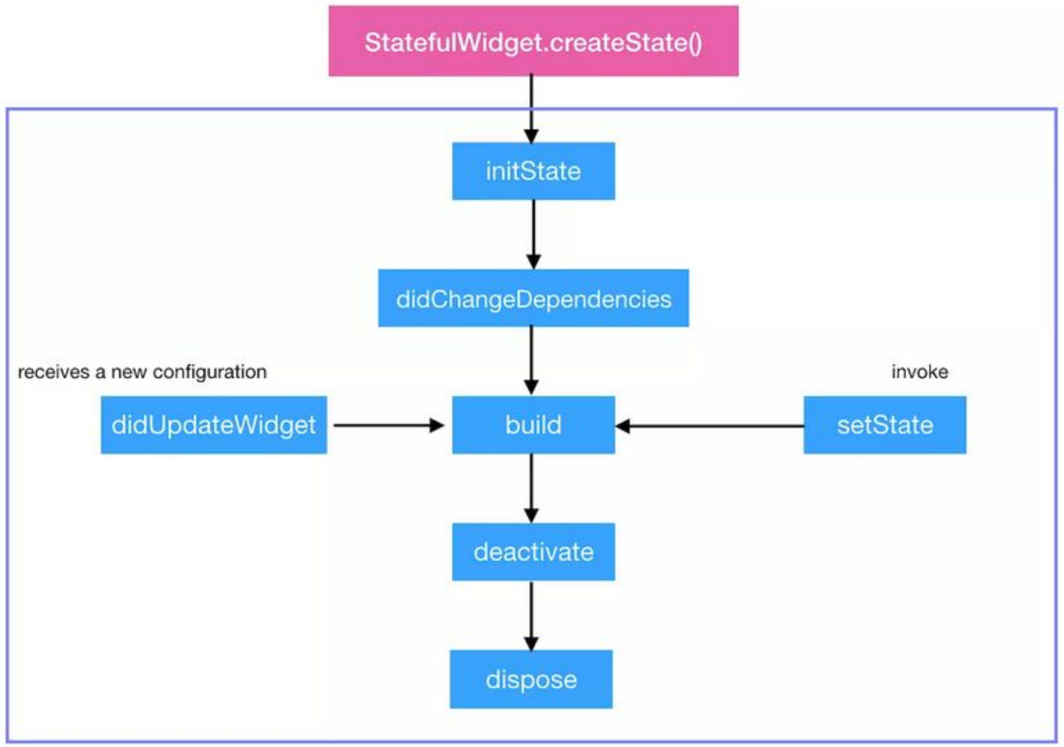
## Stateful Widgets

1. A Stateful widget maintains data and responds to whatever the data does inside the application.
2. It is a mutable widget, so it is drawn multiple times in its lifetime.
3. In simple words, if a widget does anything then its Stateful Widget.
4. A Stateful widget is dynamic in nature. That means it can keep track of changes and update the UI based on those changes.
5. We use this when the user dynamically updates the application screen.
6. For example, If you press a button and it performs any task its a Stateful Widget.

## State

1. State is information that can be read synchronously when the widget is built and might change during the lifetime of the widget.
2. State objects are created by the framework by calling the StatefulWidget.createState method when inflating a StatefulWidget to insert it into the tree.
3. Why Are StatefulWidget and State Separate Classes? In one word: performance.
4. The State objects are long lived, but StatefulWidgets (and all Widget subclasses) are thrown away and rebuilt whenever configuration changes.

## Life cycle of Stateful widget



## Life cycle : Constructor

This function is not part of the life cycle, because this time the State of the widget property is empty, if you want to access the widget properties in the constructor will not work. But the constructor must be the first call.

## Life cycle : createState

When Flutter is instructed to build a StatefulWidget, it immediately calls createState().

## Life cycle : initState

1. Called when this object is inserted into the tree.
2. When inserting the render tree when invoked, this function is called only once in the life cycle. Here you can do some initialization, such as initialization State variables.
3. initState is called once and only once. It must called super.initState().

## Life cycle : setState

1. The setState() method is called often from the Flutter framework itself and from the developer.
2. Notify the framework that the internal state of this object has changed.
3. Calling setState notifies the framework that the internal state of this object has changed in a way that might impact the user interface in this subtree, which causes the framework to schedule a build for this State object.

## Life cycle : didChangeDependencies

1. Called when a dependency of this [State] object changes.
2. This method is called immediately after initState on the first time the widget is built.

## Life cycle : didUpdateWidget

1. Called whenever the widget configuration changes.
2. If the parent widget changes and has to rebuild this widget (because it needs to give it different data), but it's being rebuilt with the same runtimeType, then this method is called.
3. This is because Flutter is re-using the state, which is long lived.
4. In this case, you may want to initialize some data again, as you would in initState.

## Life cycle : deactive

1. Called when this object is removed from the tree. Before dispose, we will call this function.
2. But it might be reinserted before the current frame change is finished.
3. This method exists basically because State objects can be moved from one point in a tree to another.

## Lifecycle : dispose

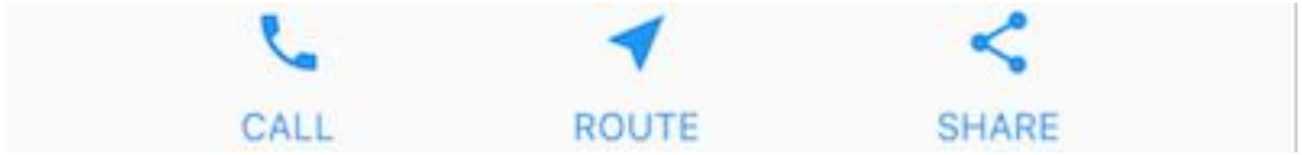
1. Called when this object is removed from the tree permanently.
2. This method is where you should unsubscribe and cancel all animations, streams, etc.
3. This stage of the lifecycle is terminal: there is no way to remount a State object that has been disposed.
4. After the framework calls dispose, the State object is considered unmounted and the mounted property is false.
5. It is an error to call setState at this point.

## Life cycle : didChangeAppLifecycleState

Called when the system puts the app in the background or returns the app to the foreground.

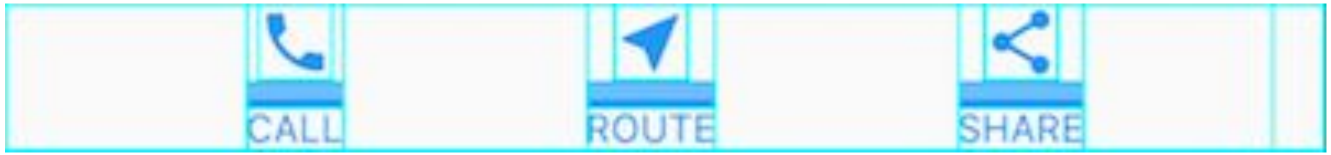
## Layouts in Flutter

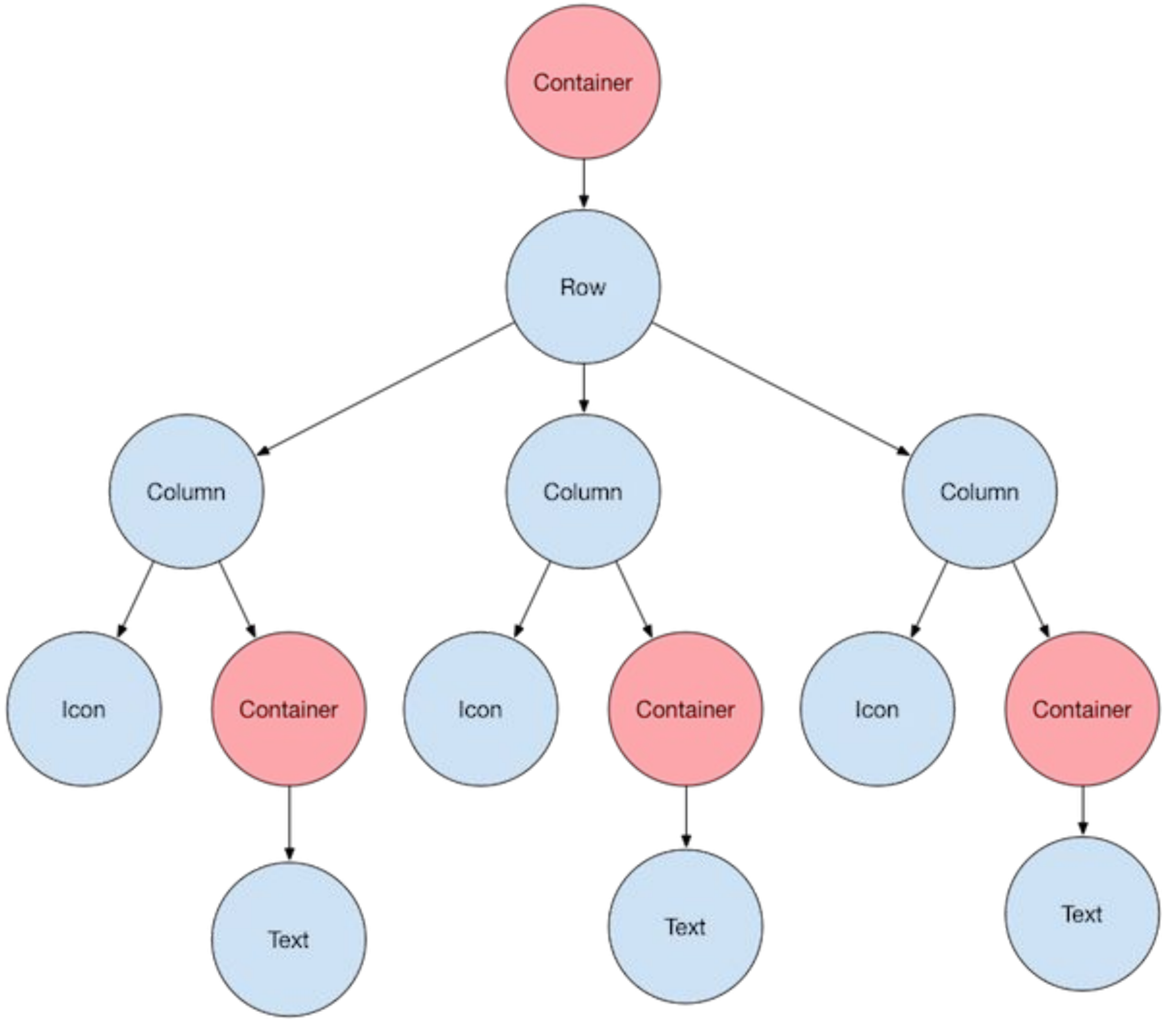
1. The core of Flutter's layout mechanism is widgets.
2. In Flutter, almost everything is a widget - images, icons, and text that you see in a Flutter app are all widgets.
3. But things you don't see are also widgets, such as the rows, columns, and grids that arrange, constrain, and align the visible widgets.
4. You create a layout by composing widgets to build more complex widgets.



## Widget Tree

1. The widget tree is how developers create their user interface; developers position widgets within each other to build simple and complex layouts.
2. Widgets are arranged into a tree of parent and child widget.
3. Widgets are nested inside of each other to form your app.
4. The Entire widget tree forms a layout that you see on the screen.




Container
Row
Column
Column
Column
Icon
Container
Icon
Container
Icon
Container
Text
Text
Text

## Layout a Widget

1. How do you lay out a single widget in Flutter?
2. We will learn this with a Hello World App example.

## Step 1 : Select a Layout widget

1. Choose from a variety of layout widgets based on how you want to align or constrain the visible widget.
2. For details of Layout widgets: https://flutter.dev/docs/development/ui/widgets/layout
3. This example uses Center which centers its content horizontally and vertically.

## Step 2 : Create a Visible Widget

1. Those widgets which will be visible to you are called as visible widgets.
2. For example, a Text, or an Image, or an Icon etc.
3. This example uses Text which will display *Hello World*.

## Step 3 : Add the visible widget to the layout widget

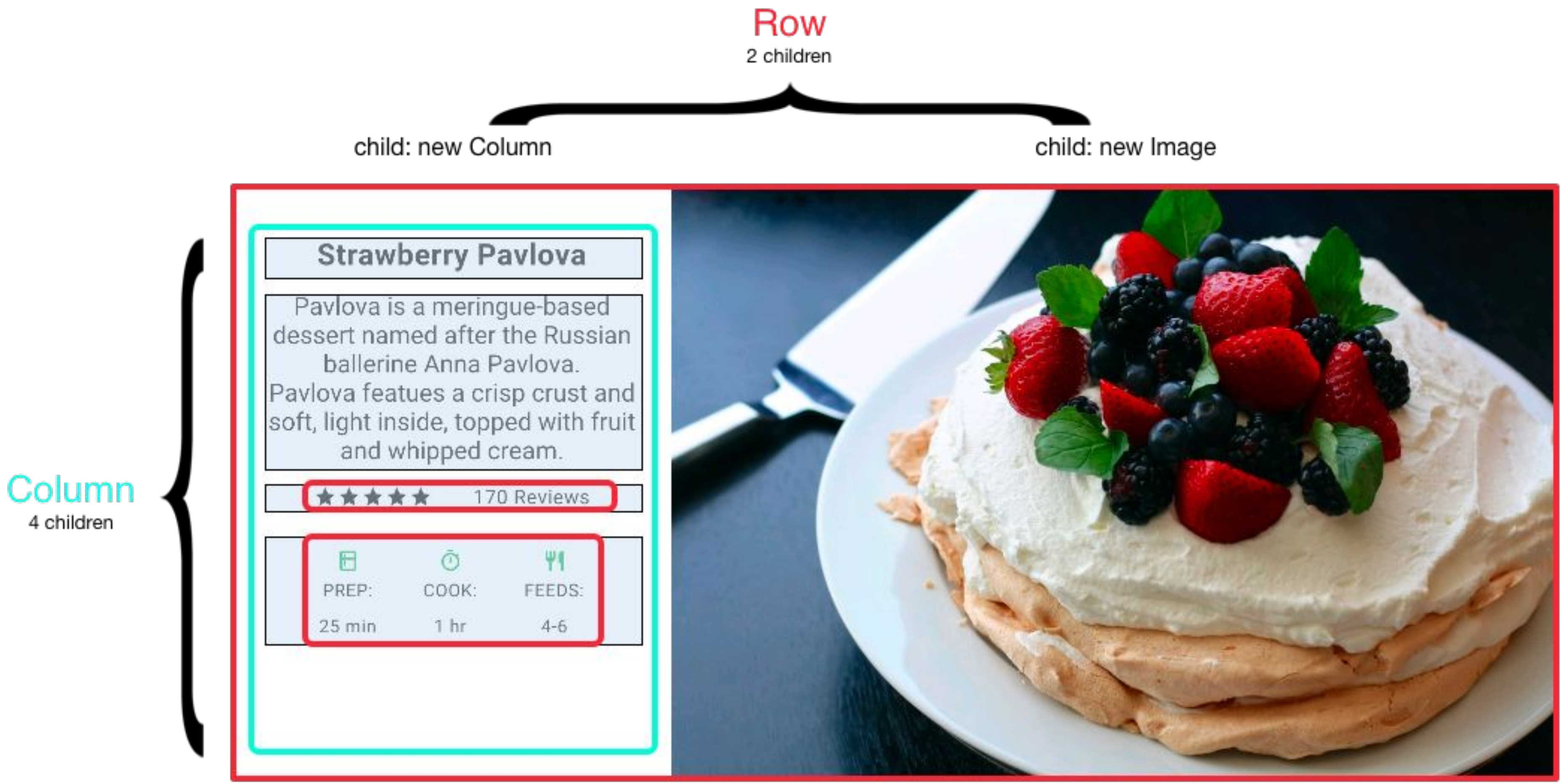
1. All layout widgets have either of the following:
   a. A child property if they take a single child—for example, Center or Container
   b. A children property if they take a list of widgets—for example, Row, Column, ListView, or Stack.
2. In our case, we will make our Text widget child of Center widget.

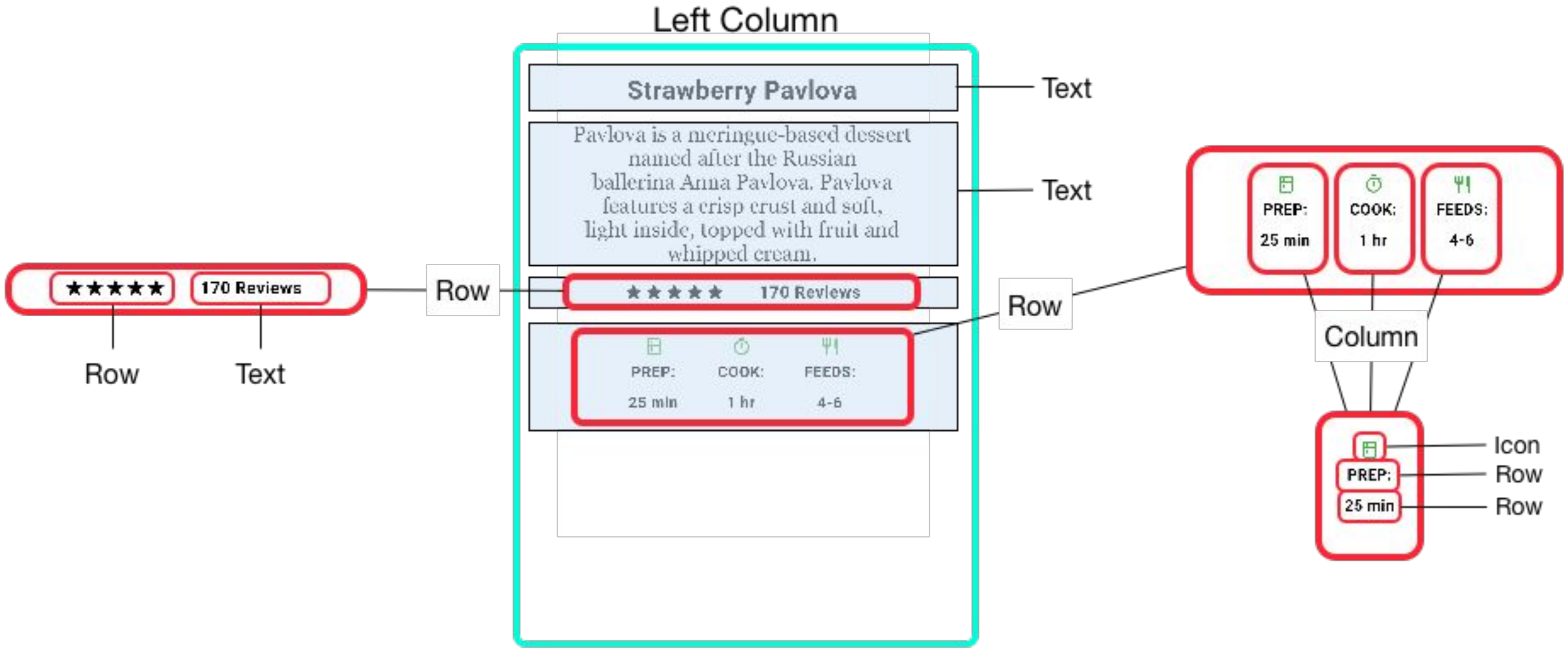
## Step 4 : Add the layout widget to the page

1. A Flutter app is itself a widget, and most widgets have a build() method.
2. Instantiating and returning a widget in the app's build() method displays the widget.
3. For a Material app, you can use a Scaffold widget; it provides a default banner, background color, and has API for adding drawers, snack bars, and bottom sheets.
4. Then you can add the Center widget directly to the body property for the home page.

## Lay out multiple widgets vertically and horizontally

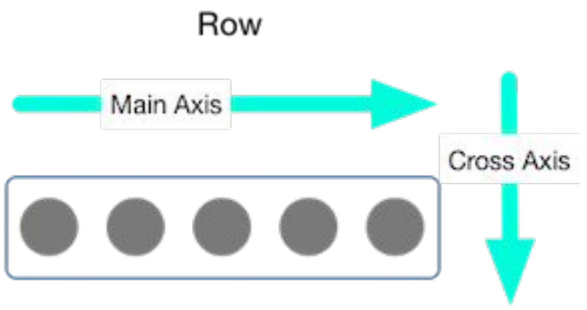
1. One of the most common layout patterns is to arrange widgets vertically or horizontally.
2. You can use a Row widget to arrange widgets horizontally, and a Column widget to arrange widgets vertically.
3. To create a row or column in Flutter, you add a list of children widgets to a Row or Column widget.
4. In turn, each child can itself be a row or column, and so on.
5. Let’s see an example for the same.

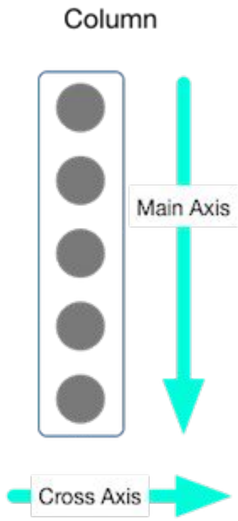

Row
2 children
child: new Column
child: new Image
Column
4 children
Strawberry Pavlova
Pavlova is a meringue-based dessert named after the Russian ballerine Anna Pavlova. Pavlova featues a crisp crust and soft, light inside, topped with fruit and whipped cream.
170 Reviews
PREP:
25 min
COOK:
1 hr
FEEDS:
4-6


Left Column
Strawberry Pavlova
Text
Pavlova is a meringue-based dessert named after the Russian ballerina Anna Pavlova. Pavlova features a crisp crust and soft, light inside, topped with fruit and whipped cream.
Text
★★★★★ 170 Reviews
Row
Row
Text
Row
PREP: 25 min
COOK: 1 hr
FEEDS: 4-6
Column
Icon
Row
Row

## Aligning widgets

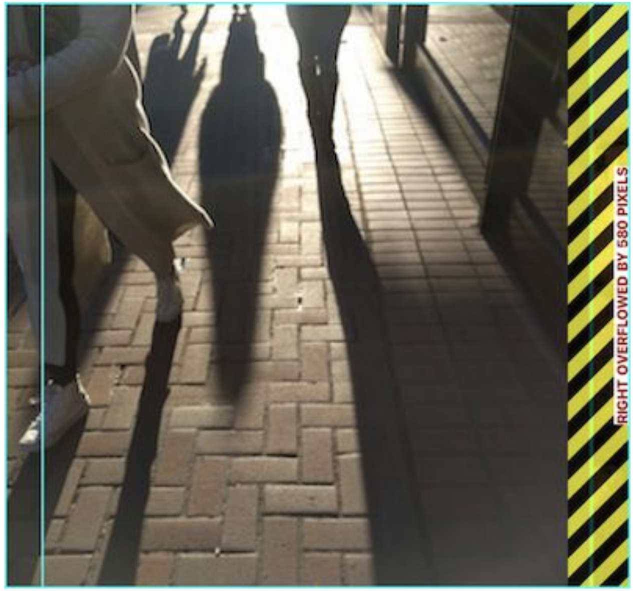
1. You control how a row or column aligns its children using the mainAxisAlignment and crossAxisAlignment properties.
2. The MainAxisAlignment and CrossAxisAlignment classes offer a variety of constants for controlling alignment.





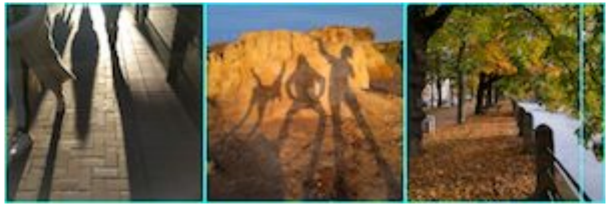
## Sizing widgets

1. When a layout is too large to fit a device, a yellow and black striped pattern appears along the affected edge. Here is an example of a row that is too wide:



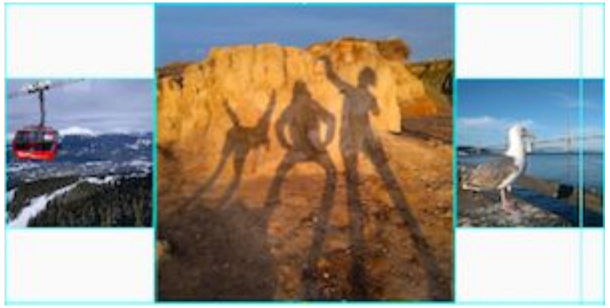
## Sizing Widgets

1. Widgets can be sized to fit within a row or column by using the Expanded widget.
2. To fix the previous example where the row of images is too wide for its render box, wrap each image with an Expanded widget.



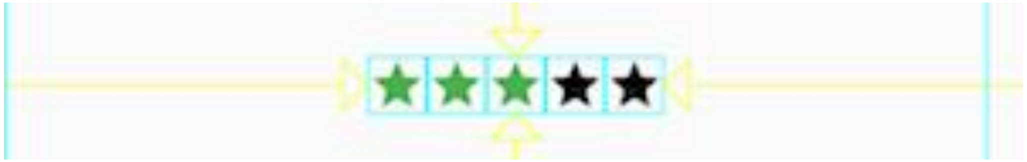
## Sizing Widgets

1. Perhaps you want a widget to occupy twice as much space as its siblings.
2. For this, use the Expanded widget flex property, an integer that determines the flex factor for a widget.
3. The default flex factor is 1.  In middle image, we will set flex factor to 2 for middle image.



## Packing widgets

1. By default, a row or column occupies as much space along its main axis as possible, but if you want to pack the children closely together, set its mainAxisSize to MainAxisSize.min.

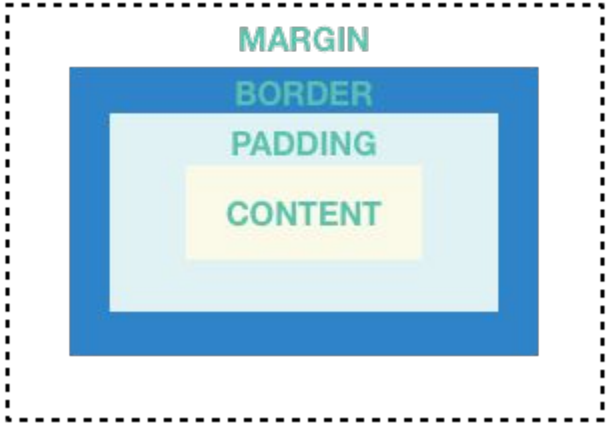


## Common Layout widgets

1. Standard widgets: These widgets are from the widgets library. It includes Container, GridView, ListView and Stack.
2. Material widgets: These widgets are from the Material library. It includes Card and ListTile.

## Container

1. Add padding, margins, borders.
2. Change background color or image.
3. Contains a single child widget, but that child can be a Row, Column, or even the root of a widget tree.



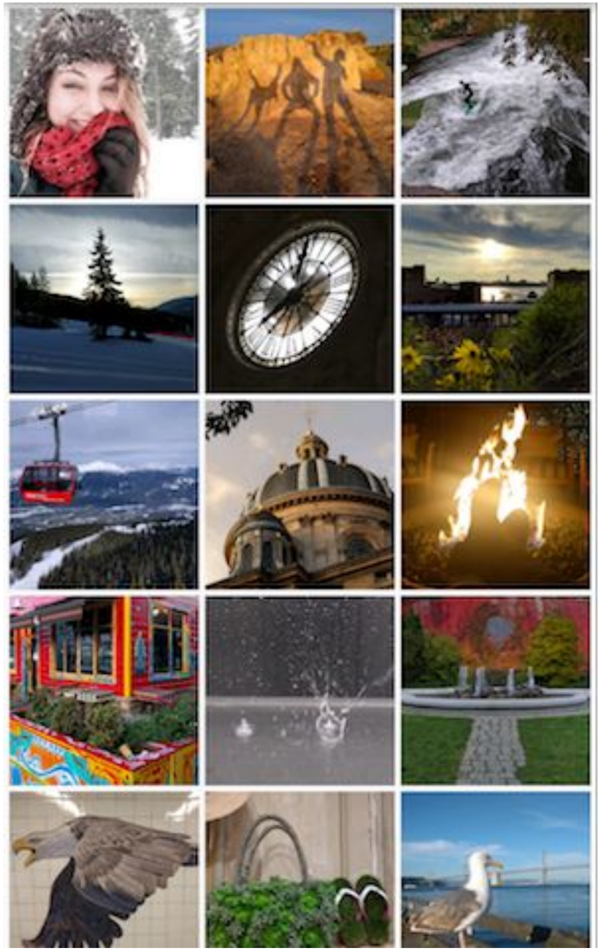
## GridView

1. Lays widgets out in a grid.
2. Detects when the column content exceeds the render box and automatically provides scrolling.
3. Build your own custom grid, or use one of the provided grids:
   a. GridView.count allows you to specify the number of columns.
   b. GridView.extent allows you to specify the maximum pixel width of a tile.

**Note:** When displaying a two-dimensional list where it’s important which row and column a cell occupies (for example, it’s the entry in the “calorie” column for the “avocado” row), use Table or DataTable.

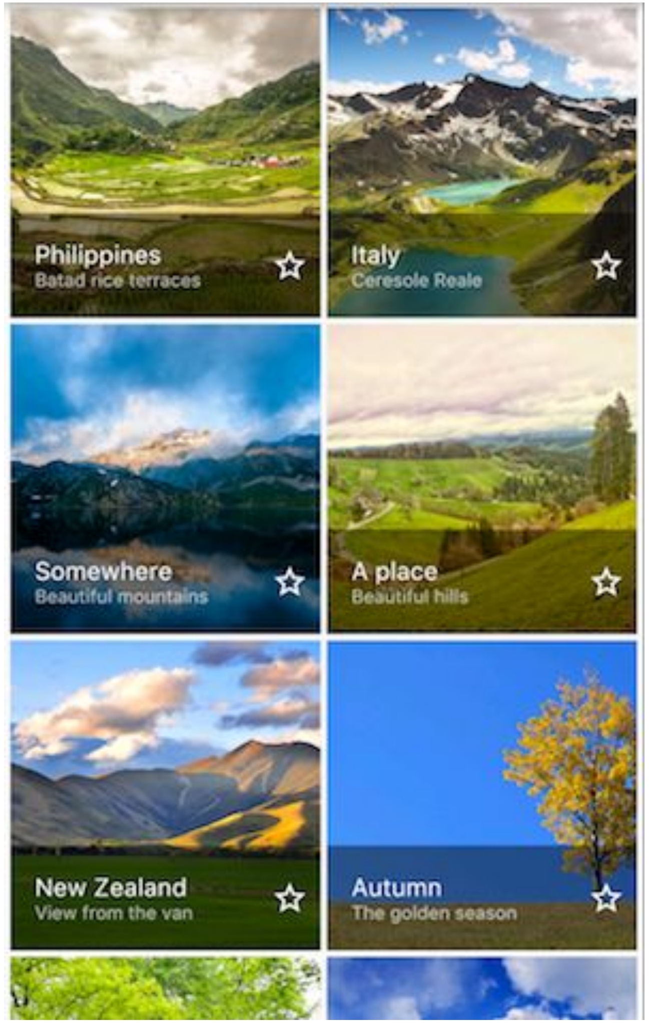
## GridView.extent

Uses GridView.extent to create a grid with tiles a maximum 150 pixels wide.



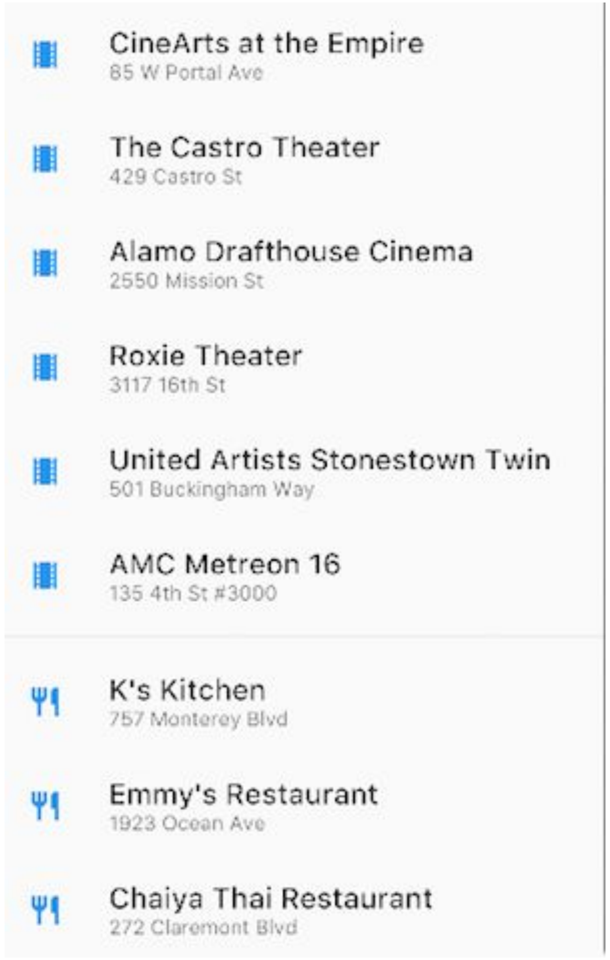
# GridView.count

Uses GridView.count to create a grid that's 2 tiles wide in portrait mode, and 3 tiles wide in landscape mode.



# ListView

1. A specialized Column for organizing a list of boxes.
2. Can be laid out horizontally or vertically.
3. Detects when its content won't fit and provides scrolling.
4. Less configurable than Column, but easier to use and supports scrolling.



CineArts at the Empire
85 W Portal Ave

The Castro Theater
429 Castro St

Alamo Drafthouse Cinema
2550 Mission St

Roxie Theater
3117 16th St

United Artists Stonestown Twin
501 Buckingham Way

AMC Metreon 16
135 4th St #3000

K's Kitchen
757 Monterey Blvd

Emmy's Restaurant
1923 Ocean Ave

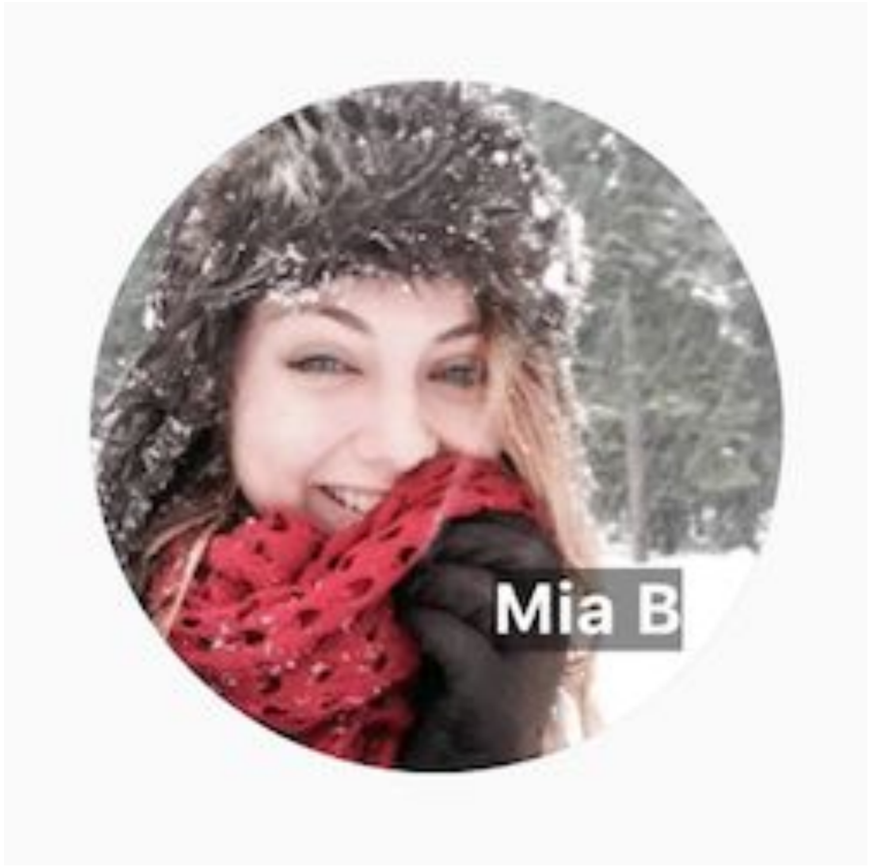
Chaiya Thai Restaurant
272 Claremont Blvd

## Stack

1. Use for widgets that overlap another widget.
2. The first widget in the list of children is the base widget; subsequent children are overlaid on top of that base widget.
3. A Stack's content can't scroll.
4. You can choose to clip children that exceed the render box.

## Example of Stack

Uses Stack to overlay a Container (that displays its Text on a translucent black background) on top of a CircleAvatar. The Stack offsets the text using the alignment property and Alignments.

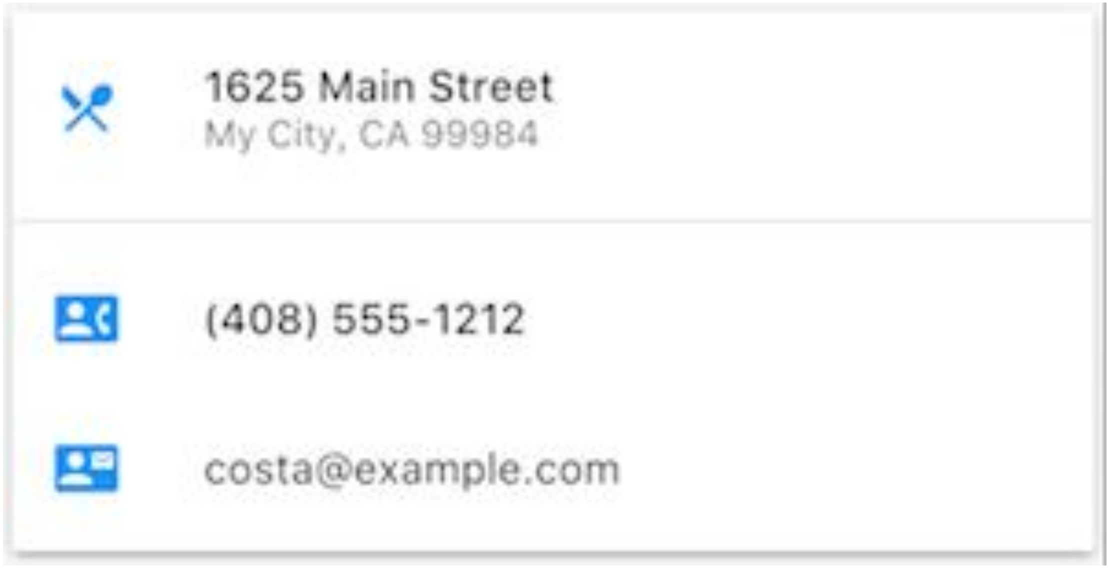


## Card

1. Implements a Material card. (https://material.io/components/cards)
2. Used for presenting related nuggets of information.
3. Accepts a single child, but that child can be a Row, Column, or other widget that holds a list of children.
4. A Card's content can't scroll.
5. By default, a Card shrinks its size to 0 by 0 pixels. You can use SizedBox to constrain the size of a card.
6. Displayed with rounded corners and a drop shadow.
7. Changing a Card's elevation property allows you to control the drop shadow effect. (https://material.io/design/environment/elevation.html)

## Example of Card

A Card containing 3 ListTiles and sized by wrapping it with a SizedBox. A Divider separates the first and second ListTiles.



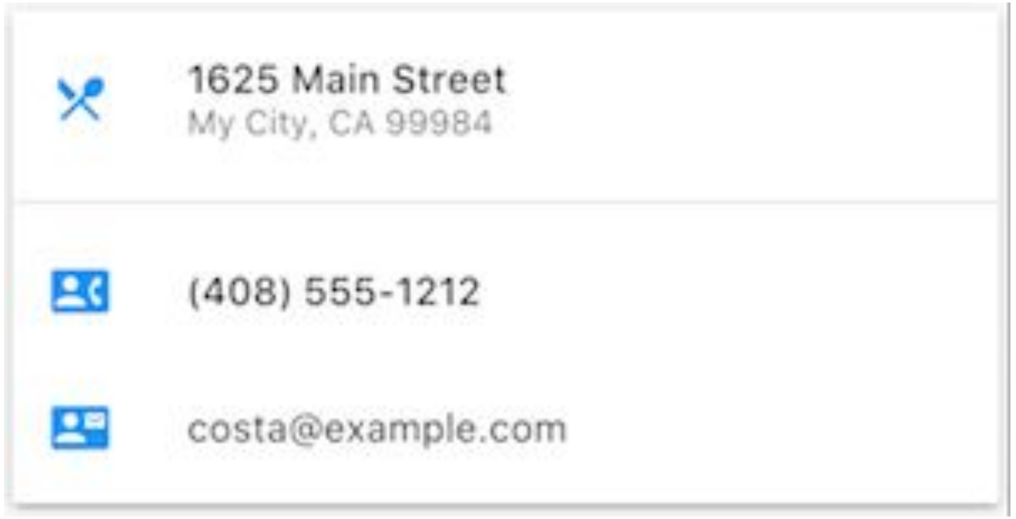
1625 Main Street
My City, CA 99984

(408) 555-1212

costa@example.com

## ListTile

1. A specialized row that contains up to 3 lines of text and optional icons.
2. Less configurable than Row, but easier to use.
3. ListTile is most commonly used in Card or ListView, but can be used elsewhere.



1625 Main Street
My City, CA 99984

(408) 555-1212

costa@example.com

## Creating responsive and adaptive apps

1. One of Flutter’s primary goals is to create a framework that allows you to develop apps from a single codebase that look and feel great on any platform.
2. This means that your app may appear on screens of many different sizes, from a watch, to a foldable phone with two screens, to a high def monitor.
3. Two terms that describe concepts for this scenario are adaptive and responsive. Ideally, you’d want your app to be both but what, exactly, does this mean? These terms are similar, but they are not the same.

## The difference between an adaptive and a responsive

1. Adaptive and responsive can be viewed as separate dimensions of an app: you can have an adaptive app that is not responsive, or vice versa. And, of course, an app can be both, or neither.
2. Let us understand both of them one by one.

## Responsive

1. A responsive app has had its layout tuned for the available screen size.
2. Often this means (for example), re-laying out the UI if the user resizes the window, or changes the device's orientation.
3. This is especially necessary when the same app can run on a variety of devices, from a watch, phone, tablet, to a laptop or desktop computer.

## Adaptive

1. Adapting an app to run on different device types, such as mobile and desktop, requires dealing with mouse and keyboard input, as well as touch input.
2. It also means there are different expectations about the app’s visual density, how component selection works, using platform-specific features (such as top-level windows), and more.

## Creating a responsive Flutter app

1. Flutter allows you to create apps that self-adapt to the device’s screen size and orientation.
2. There are two basic approaches to creating Flutter apps with responsive design:
   a. Use the LayoutBuilder class.
   b. Use the MediaQuery.of() method in your build functions.

## Use the LayoutBuilder class

1. From its builder property, you get a BoxConstraints object.
2. Examine the constraint's properties to decide what to display.

## Use the MediaQuery.of() method in your build functions

1. This gives you the size, orientation, etc, of your current app.
2. This is more useful if you want to make decisions based on the complete context rather than on just the size of your particular widget.
3. Again, if you use this, then your build function automatically runs if the user somehow changes the app’s size.

## Other useful widgets and classes for responsive UI

AspectRatio, CustomSingleChildLayout, CustomMultiChildLayout, FittedBox, FractionallySizedBox, LayoutBuilder, MediaQuery, MediaQueryData and OrientationBuilder.

## Building adaptive apps

1. Flutter provides new opportunities to build apps that can run on mobile, desktop, and the web from a single codebase. However, with these opportunities, come new challenges.
2. You want your app to feel familiar to users, adapting to each platform by maximizing usability and ensuring a comfortable and seamless experience. That is, you need to build apps that are not just multiplatform, but are fully platform adaptive.
3. There are many considerations for developing platform-adaptive apps, but they fall into three major categories: Layout, Input, Idioms and Norms.

## Building adaptive layouts

One of the first things you must consider when bringing your app to multiple platforms is how to adapt it to the various sizes and shapes of the screens that it will run on.

1. Layout widgets.
2. Visual density.
3. Contextual widgets.
4. Single source of truth while styling.
5. Design to the strengths of each form factor.
6. Use desktop build targets for rapid testing.
7. Solve touch first.

## Layout widgets

1. If you've been building apps or websites, you're probably familiar with creating responsive interfaces. Luckily for Flutter developers, there are a large set of widgets to make this easier.
2. Some of Flutter's most useful layout widgets include:
    a. Single child: LayoutBuilder, Expanded, Align, ConstrainedBox etc.
    b. Multi child: Column, Row, Stack, ListView, GridView, Table etc.
3. [https://flutter.dev/docs/development/ui/widgets/layout](https://flutter.dev/docs/development/ui/widgets/layout)

## Visual density

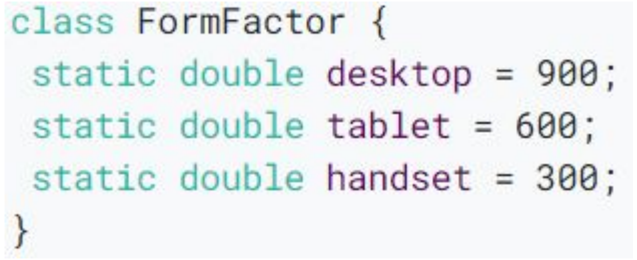
1. Different input devices offer various levels of precision, which necessitate differently sized hit areas.
2. Flutter’s VisualDensity class makes it easy to adjust the density of your views across the entire application, for example, by making a button larger (and therefore easier to tap) on a touch device.
3. To set a custom visual density, inject the density into your MaterialApp theme.

## Contextual layout

If you need more than density changes and can’t find a widget that does what you need, you can take a more procedural approach to adjust parameters, calculate sizes, swap widgets, or completely restructure your UI to suit a particular form factor.

## Contextual layout: Screen-based breakpoints

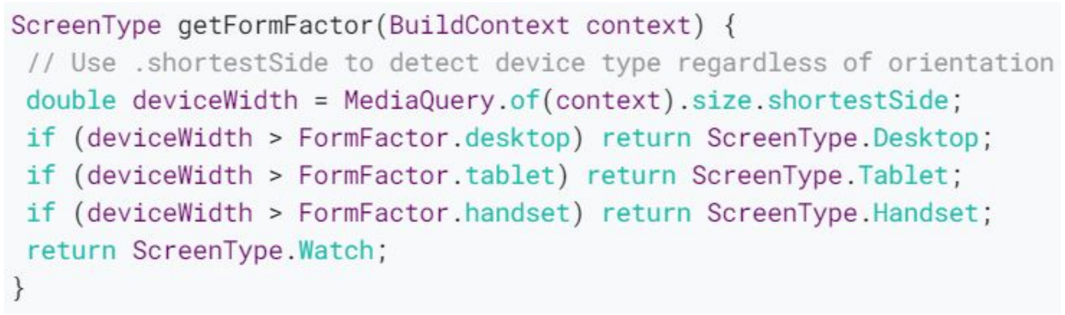
1. The simplest form of procedural layouts uses screen-based breakpoints.
2. In Flutter, this can be done with the MediaQuery API.
3. There are no hard and fast rules for the sizes to use here, but these are general values:



```
class FormFactor {
 static double desktop = 900;
 static double tablet = 600;
 static double handset = 300;
}
```

## Contextual layout: Screen-based breakpoints

4. Using breakpoints, you can set up a simple system to determine the device type:



```
ScreenType getFormFactor(BuildContext context) {
 // Use .shortestSide to detect device type regardless of orientation
 double deviceWidth = MediaQuery.of(context).size.shortestSide;
 if (deviceWidth > FormFactor.desktop) return ScreenType.Desktop;
 if (deviceWidth > FormFactor.tablet) return ScreenType.Tablet;
 if (deviceWidth > FormFactor.handset) return ScreenType.Handset;
 return ScreenType.Watch;
}
```

5. Screen-based breakpoints are best used for making top-level decisions in your app. Changing things like visual density, paddings, or font-sizes are best when defined on a global basis.

## Contextual layout: Use LayoutBuilder for extra flexibility

1. Even though checking total screen size is great for full-screen pages or making global layout decisions, it's often not ideal for nested subviews. Often, subviews have their own internal breakpoints and care only about the space that they have available to render.
2. The simplest way to handle this in Flutter is using the LayoutBuilder class. LayoutBuilder allows a widget to respond to incoming local size constraints, which can make the widget more versatile than if it depended on a global value.

## Contextual layout: Device segmentation

1. There are times when you want to make layout decisions based on the actual platform you're running on, regardless of size. For example, when building a custom title bar, you might need to check the operating system type and tweak the layout of your title bar, so it doesn't get covered by the native window buttons.
2. To determine which combination of platforms you're on, you can use the Platform API along with the kIsWeb value:

```
bool get isMobileDevice => !kIsWeb && (Platform.isIOS || Platform.isAndroid);
bool get isDesktopDevice =>
    !kIsWeb && (Platform.isMacOS || Platform.isWindows || Platform.isLinux);
bool get isMobileDeviceOrWeb => kIsWeb || isMobileDevice;
bool get isDesktopDeviceOrWeb => kIsWeb || isDesktopDevice;
```

## Single source of truth for styling

1. You’ll probably find it easier to maintain your views if you create a single source of truth for styling values like padding, spacing, corner shape, font sizes, and so on.
2. This can be done easily with some helper classes:
3. These constants can then be used in place of  hard-coded numeric values:

```dart
class Insets {
 static const double xsmall = 4;
 static const double small = 8;
 // etc
}
```

```dart
return Padding(
     insets: EdgeInsets.all(Insets.small),
     child: Text('Hello!', style: TextStyles.body1)
)
```

## Single source of truth for styling

4. With all views referencing the same shared-design system rules, they tend to look better and more consistent. Making a change or adjusting a value for a specific platform can be done in a single place, instead of using an error-prone search and replace.

5. Using shared rules has the added benefit of helping enforce consistency on the design side.

6. Some common design system categories that can be represented this way are: Animation timings, Sizes and breakpoints, Insets and paddings, Corner radius, Shadows, Strokes, Font families, sizes, and styles.

## Design to the strengths of each form factor

1. Beyond screen size, you should also spend time considering the unique strengths and weaknesses of different form factors.
2. It isn’t always ideal for your multi platform app to offer identical functionality everywhere. Consider whether it makes sense to focus on specific capabilities, or even remove certain features, on some device categories.
3. For example, mobile devices are portable and have cameras, but they aren’t well suited for detailed creative work. With this in mind, you might focus more on capturing content and tagging it with location data for a mobile UI, but focus on organizing or manipulating that content for a tablet or desktop UI.
4. The key takeaway here is to think about what each platform does best and see if there are unique capabilities you can leverage.

## Use desktop build targets for rapid testing

1. One of the most effective ways to test adaptive interfaces is to take advantage of the desktop build targets.
2. When running on a desktop, you can easily resize the window while the app is running to preview various screen sizes. This, combined with hot reload, can greatly accelerate the development of a responsive UI.

## Solve touch first

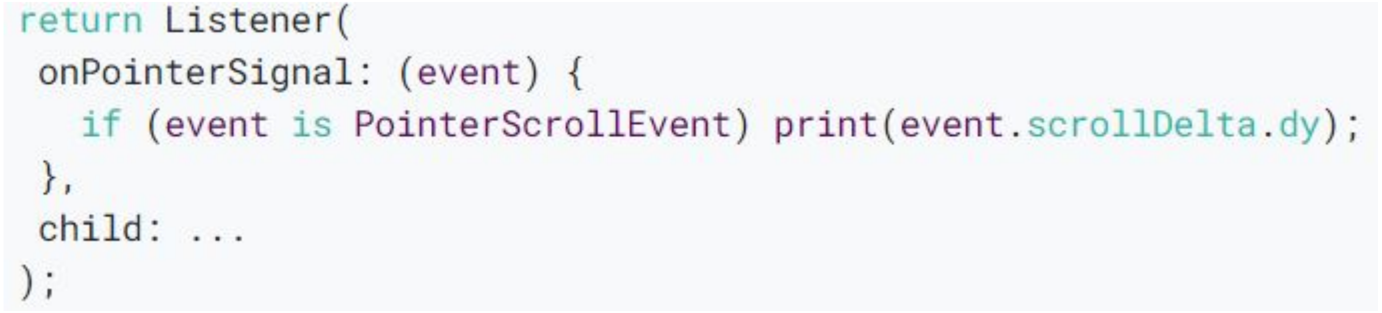
1. Building a great touch UI can often be more difficult than a traditional desktop UI due, in part, to the lack of input accelerators like right-click, scroll wheel, or keyboard shortcuts.
2. One way to approach this challenge is to focus initially on a great touch-oriented UI. You can still do most of your testing using the desktop target for its iteration speed. But, remember to switch frequently to a mobile device to verify that everything feels right.
3. After you have the touch interface polished, you can tweak the visual density for mouse users, and then layer on all the additional inputs.

## Input

1. Of course, it isn’t enough to just adapt how your app looks, you also have to support varying user inputs.
2. The mouse and keyboard introduce input types beyond those found on a touch device—like scroll wheel, right-click, hover interactions, tab traversal, and keyboard shortcuts.

## Input: Scroll wheel

1. Scrolling widgets like ScrollView or ListView support the scroll wheel by default, and because almost every scrollable custom widget is built using one of these, it works with them as well.
2. If you need to implement custom scroll behavior, you can use the Listener widget, which lets you customize how your UI reacts to the scroll wheel.



```
return Listener(
 onPointerSignal: (event) {
   if (event is PointerScrollEvent) print(event.scrollDelta.dy);
 },
 child: ...
);
```

## Input: Tab traversal and focus interactions

1. Users with physical keyboards expect that they can use the tab key to quickly navigate your application.
2. There are two considerations for tab interactions: how focus moves from widget to widget, known as traversal, and the visual highlight shown when a widget is focused.
3. Most built-in components, like buttons and text fields, support traversal and highlights by default. If you have your own widget that you want included in traversal, you can use the FocusableActionDetector widget to create your own controls.
4. To get more control over the order that widgets are focused on when the user presses tab, you can use FocusTraversalGroup to define sections of the tree that should be treated as a group when tabbing.

## Input: Keyboard accelerators

1. In addition to tab traversal, desktop and web users are accustomed to having various keyboard shortcuts bound to specific actions.
2. Whether it’s the Delete key for quick deletions or Control+N for a new document, be sure to consider the different accelerators your users expect.
3. If you have a single widget like a TextField or a Button that already has a focus node, you can wrap it in a RawKeyboardListener and listen for keyboard events.
4. If you’d like to apply a set of keyboard shortcuts to a large section of the tree, you can use the Shortcuts widget.

## Input: Mouse enter, exit, and hover

1. On desktop, it’s common to change the mouse cursor to indicate the functionality about the content the mouse is hovering over.
2. For example, you usually see a hand cursor when you hover over a button, or an I cursor when you hover over text.
3. The Material Component set has built-in support for your standard button and text cursors. To change the cursor from within your own widgets, use MouseRegion.
4. MouseRegion is also useful for creating custom rollover and hover effects.

## Idioms and norms

1. The final area to consider for adaptive apps is platform standards. Each platform has its own idioms and norms; these nominal or de facto standards inform user expectations of how an application should behave.
2. Thanks, in part to the web, users are accustomed to more customized experiences, but reflecting these platform standards can still provide significant benefits:
   a. Reduce cognitive load.
   b. Build trust.

## Common idioms and norms to consider

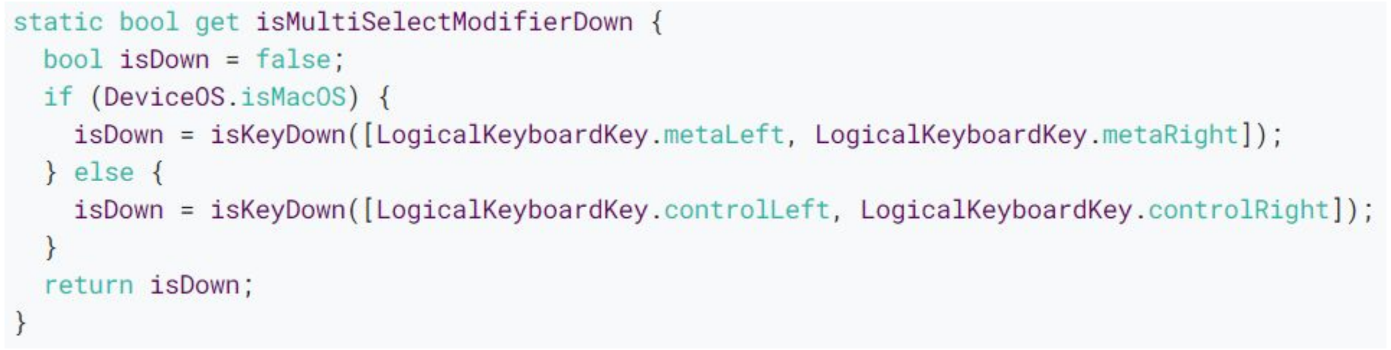
1. Scrollbar appearance and behavior.
2. Multi-select.
3. Selectable text.
4. Context menus and tooltips.
5. Horizontal button order.

## Scrollbar appearance and behavior

1. Desktop and mobile users expect scrollbars, but they expect them to behave differently on different platforms. Mobile users expect smaller scrollbars that only appear while scrolling, whereas desktop users generally expect omnipresent, larger scrollbars that they can click or drag.
2. Flutter comes with a built-in Scrollbar widget that already has support for adaptive colors and sizes according to the current platform. The one tweak you might want to make is to toggle alwaysShown when on a desktop platform.

## Multi-select

1. To perform a platform-aware check for control or command, you can write something like this:



```
static bool get isMultiSelectModifierDown {
  bool isDown = false;
  if (DeviceOS.isMacOS) {
    isDown = isKeyDown([LogicalKeyboardKey.metaLeft, LogicalKeyboardKey.metaRight]);
  } else {
    isDown = isKeyDown([LogicalKeyboardKey.controlLeft, LogicalKeyboardKey.controlRight]);
  }
  return isDown;
}
```

2. A final consideration for keyboard users is the Select All action

## Selectable text

1. A common expectation on the web (and to a lesser extent desktop) is that most visible text can be selected with the mouse cursor.
2. This is easy to support with the SelectableText widget.
3. To support rich text, then use TextSpan.

## Context menus and tooltips

1. On desktop, there are several interactions that manifest as a widget shown in an overlay, but with differences in how they're triggered, dismissed, and positioned:
   a. **Context menu:** Typically triggered by a right-click, a context menu is positioned close to the mouse, and is dismissed by clicking anywhere, selecting an option from the menu, or clicking outside it.
   b. **Tooltip:** Typically triggered by hovering for 200-400ms over an interactive element, a tooltip is usually anchored to a widget (as opposed to the mouse position) and is dismissed when the mouse cursor leaves that widget.

## Context menus and tooltips

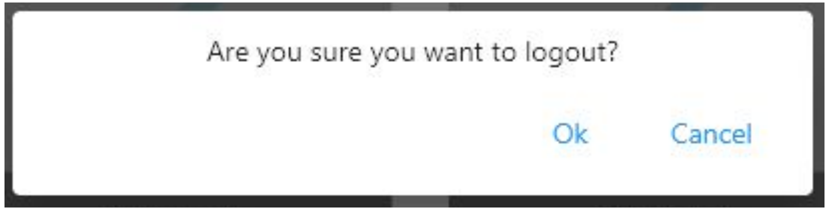
c. **Popup panel:** Similar to a tooltip, a popup panel is usually anchored to a widget. The main difference is that panels are most often shown on a tap event, and they usually don’t hide themselves when the cursor leaves. Instead, panels are typically dismissed by clicking outside the panel or by pressing a Close or Submit button.

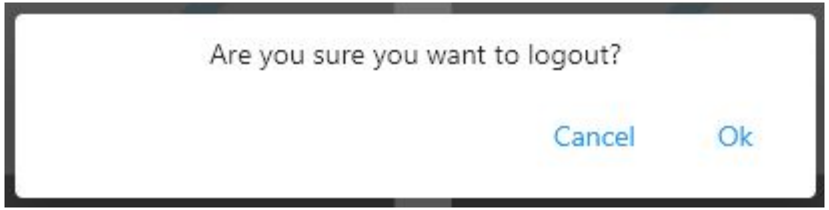
3. To show basic tooltips in Flutter, use the built-in Tooltip widget. Flutter also provides built-in context menus when editing or selecting text.

4. To show more advanced tooltips, popup panels, or create custom context menus, you either use third party packages, or build it yourself.

## Horizontal button order

1. On Windows, when presenting a row of buttons, the confirmation button is placed at the start of the row (left side). On all other platforms, it's the opposite. The confirmation button is placed at the end of the row (right side).
2. This can be easily handled in Flutter using the TextDirection property on Row.


Are you sure you want to logout?


Are you sure you want to logout?

## Navigation to a new screen and back

1. Most apps contain several screens for displaying different types of information.
2. In Flutter, screens and pages are called routes. The remainder of this recipe refers to routes.
3. In Android, a route is equivalent to an Activity. In iOS, a route is equivalent to a ViewController. In Flutter, a route is just a widget.
4. To switch to a new route, use the Navigator.push() method. The push() method adds a Route to the stack of routes managed by the Navigator.
5. By using the Navigator.pop() method we can return to first route.

## Navigate with named route

1. In the Navigate to a new screen and back, you learned how to navigate to a new screen by creating a new route and pushing it to the Navigator.
2. However, if you need to navigate to the same screen in many parts of your app, this approach can result in code duplication. The solution is to define a named route, and use the named route for navigation.
3. To work with named routes, use the Navigator.pushNamed() function.
4. In some cases, you might also need to pass arguments to a named route.
5. You can accomplish this task using the arguments parameter of the Navigator.pushNamed() method. Extract the arguments using the ModalRoute.of() method or inside an onGenerateRoute() function.

## Return data from a screen

1. In some cases, you might want to return data from a new screen.
2. You can do this with the Navigator.pop() method.

```
final result = await Navigator.push(
  context,
  MaterialPageRoute(builder: (context) => const SelectionScreen()),
);
```

```
Navigator.pop(context, 'Yep!');
```

## Deep linking

1. Flutter supports deep linking on iOS, Android, and web browsers.
2. Opening a URL displays that screen in your app.
3. If you’re running the app in a web browser, there’s no additional setup required. We need to enable them in Android and iOS.
4. For Android: Add a metadata tag and intent filter to AndroidManifest.xml inside the <activity> tag with the ".MainActivity" name:

```
<!-- Deep linking -->
<meta-data android:name="flutter_deeplinking_enabled" android:value="true" />
<intent-filter android:autoVerify="true">
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    <data android:scheme="http" android:host="flutterbooksample.com" />
    <data android:scheme="https" />
</intent-filter>
```

## Homework

1. https://flutter.dev/docs/cookbook/navigation/hero-animations
2. https://flutter.dev/docs/development/ui/assets-and-images
3. https://flutter.dev/docs/cookbook/images/network-image
4. https://flutter.dev/docs/cookbook/images/fading-in-images
5. https://flutter.dev/docs/cookbook/images/cached-images
6. https://flutter.dev/docs/cookbook/lists/long-lists
7. https://flutter.dev/docs/cookbook/lists/floating-app-bar
8. https://flutter.dev/docs/cookbook/forms

REACT NATIVE
XAMARIN
FLUTTER
imgflip.com

Thank You!