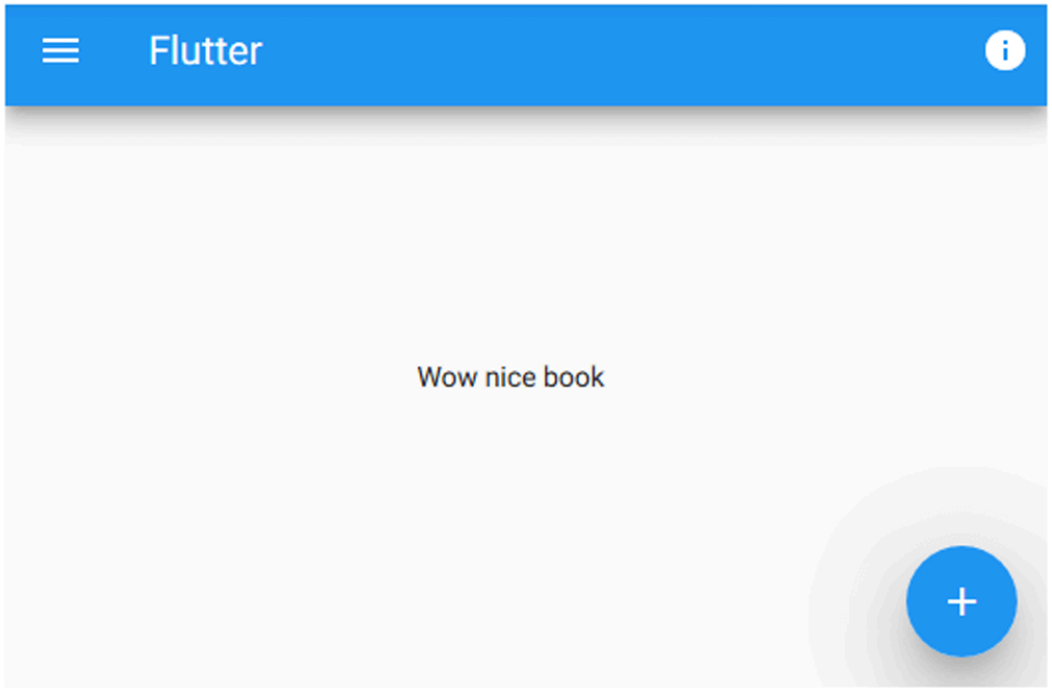
Day 5:
Building UIs in Flutter

Two Main Design Systems in Flutter

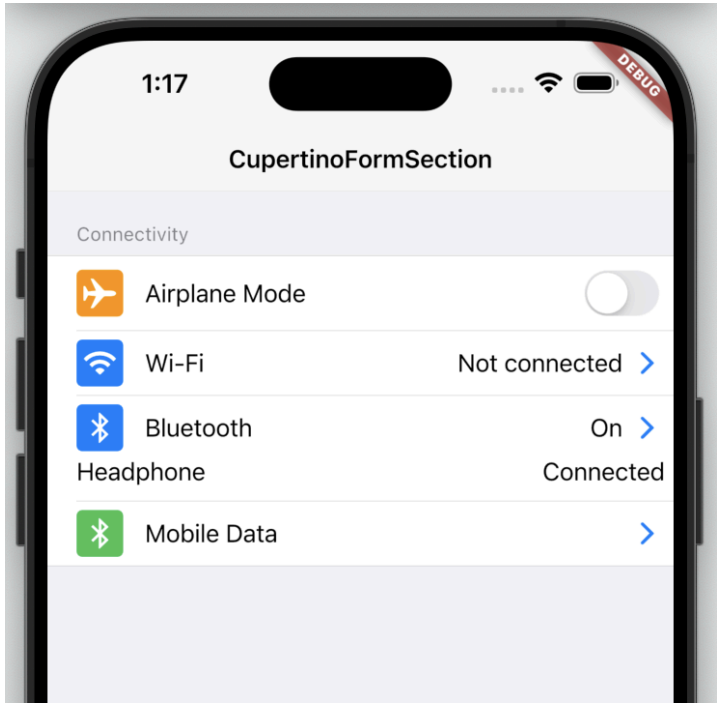
1. Material Design (Android):

Material Design is a design language developed by Google. The Material Design components in Flutter provide a comprehensive set of pre-designed widgets that follow the Material Design guidelines.



2. Cupertino (iOS):

Cupertino is a design language developed by Apple. The Cupertino components in Flutter provide a set of pre-designed widgets that follow the iOS design guidelines.



The MaterialApp Widget

Example:

```
MaterialApp(
  home: Scaffold(
    appBar: AppBar(title: Text("App")),
    body: Center(child: Text("Content")),
    floatingActionButton: FloatingActionButton(...),
  ),
)
```

Benefits:

- No need to build UI from scratch
- Consistent Material Design
- Built-in navigation & theming support

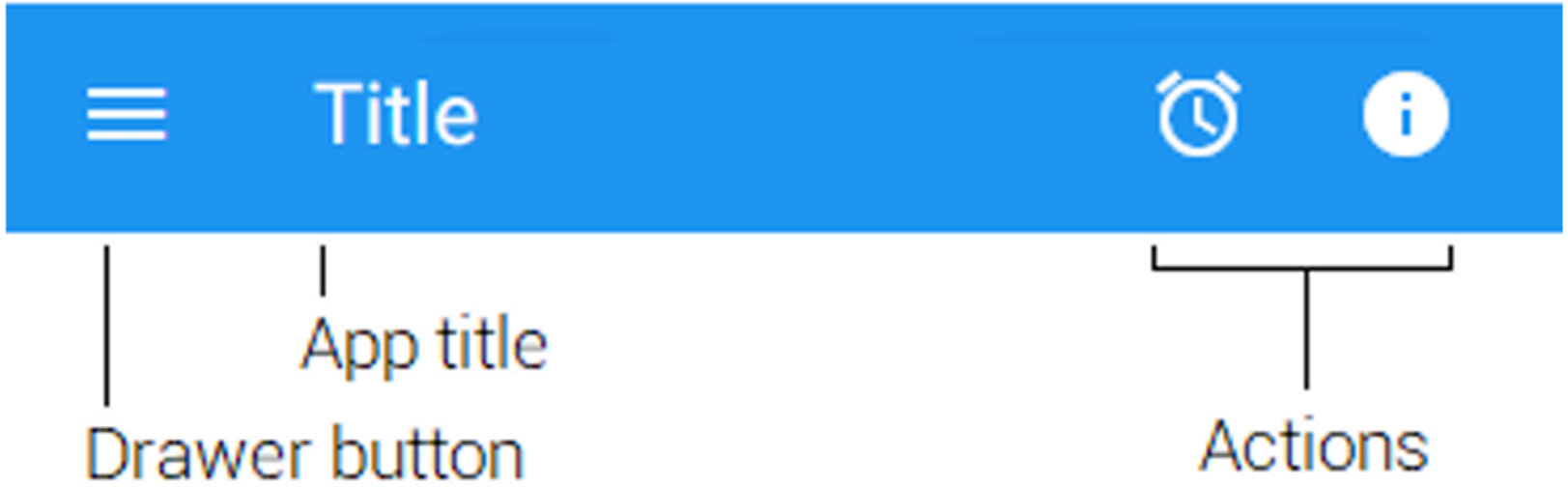
Scaffold Components

Common Scaffold Parts:

- **AppBar**:
- **Drawer**:
- **Floating Action Button**:
- **Body**:

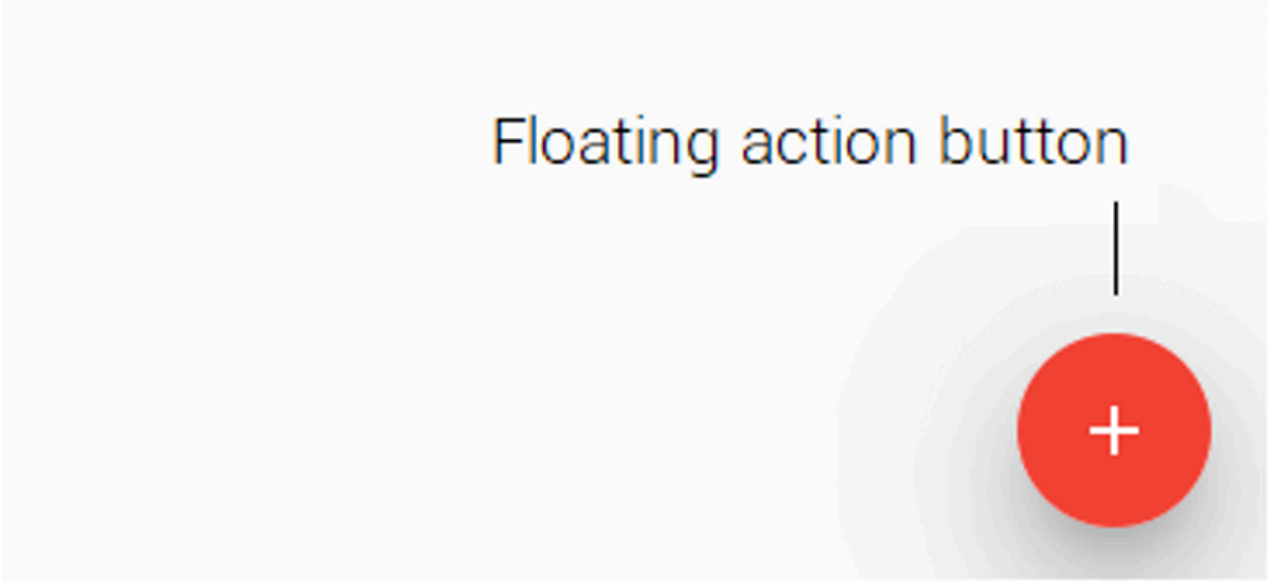
AppBar Example

AppBar is a horizontal bar at the top of the screen that typically contains the title of the screen and navigation actions.



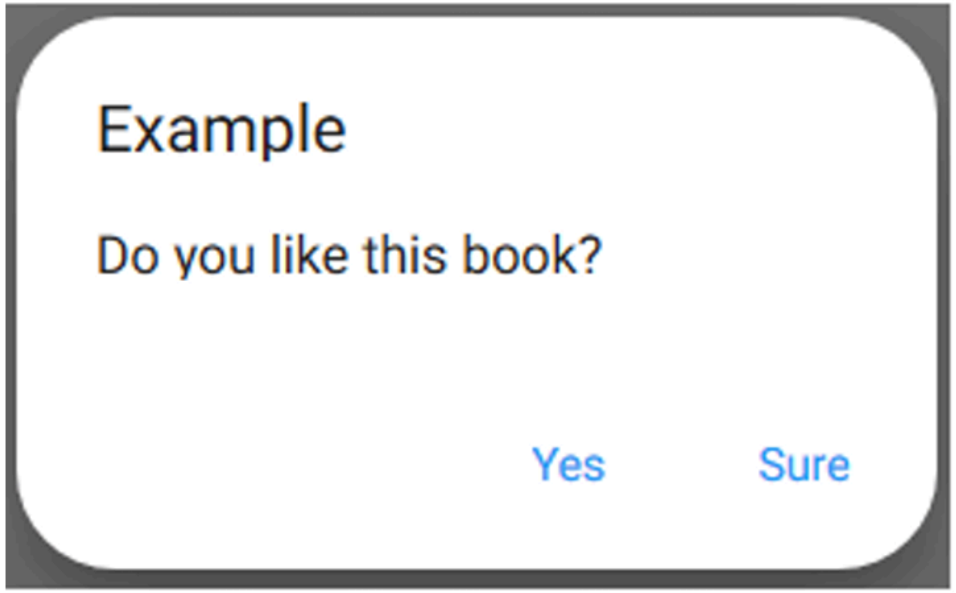
Floating Action Button Example

Floating Action Button is a circular button that floats above the content. It is used for primary actions in an application.



Dialogs

Dialogs are pop-up windows that require user interaction. In Material Design, dialogs are used to inform users about important information or to get user input.



Material Buttons

Common Button Types:

- OutlinedButton
- ElevatedButton
- TextButton
- IconButton

Example:

```
ElevatedButton(
  onPressed: () {},
  child: Text("Press Me"),
)
```

Building UIs

In Flutter, we can build UIs in several ways:

- Manual coding using Material/Cupertino widgets.
- FlutterFlow (drag-and-drop UI builder).
- Figma to Flutter plugins.

Two Approaches to Building UIs

Not Recommended:

- Create the entire layout from scratch using stateless and stateful widgets
- Requires dealing with screen dimensions, positioning, buttons, and more
- Quite a lot of work to do

Recommended:

- Import the `material.dart` package
- Use the `MaterialApp()` widget provided by Flutter
- Represents the "skeleton" of a UI following Material Design guidelines
- Very convenient and saves development time

Cupertino

Dialog

Button

iOS-Style Apps:

```
CupertinoApp(
  home: CupertinoPageScaffold(
    navigationBar: CupertinoNavigationBar(middle: Text("Title")),
    child: Center(child: Text("iOS Style")),
  ),
)
```

Key Widgets:

- `CupertinoPageScaffold` (with navigation bar)
- `CupertinoTabScaffold` (with tab bar)

Cupertino Tab Navigation

Tab-Based iOS Apps:

```
CupertinoTabScaffold(
  tabBar: CupertinoTabBar(
    items: [
      BottomNavigationBarItem(icon: Icon(Icons.home), title: Text("Home")),
    ],
  ),
  tabBuilder: (context, index) => CupertinoTabView(...),
)
```

Features:

- Automatic caching of inactive tabs
- No swipe gesture (tap-only navigation)

Cupertino Widgets

Common iOS Components:

- `CupertinoAlertDialog` : iOS-style alert
- `CupertinoButton` : Flat iOS button
- `CupertinoDialogAction` : Specialized dialog button

Example:

```
CupertinoAlertDialog(
  actions: [
    CupertinoDialogAction(
      isDestructiveAction: true,
      child: Text("Delete"),
      onPressed: () {},
    ),
  ],
)
```

Platform-Specific Considerations

Single OS vs. Cross-Platform:

- **Single OS**: Use Material/Cupertino exclusively if design matches guidelines
- **Cross-Platform**: Create unified UI that works on both

Wrong Approach:

```
if (Platform.isAndroid) runApp(AndroidVersion());
else runApp(iOSVersion()); // Maintains two codebases
```

Right Approach:

- Single codebase with adaptive design
- Use platform detection for minor adjustments only

Responsive Design Need

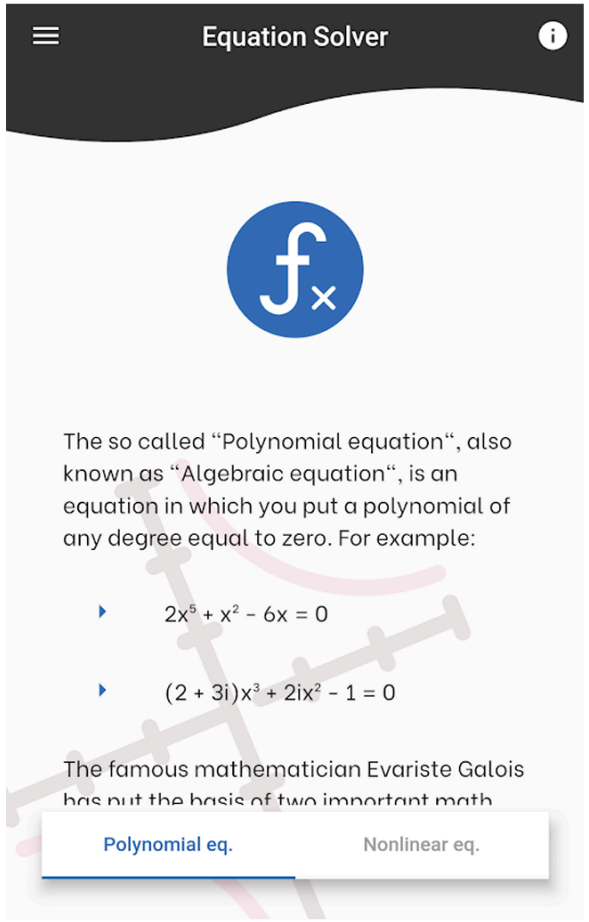
Why Responsive?:

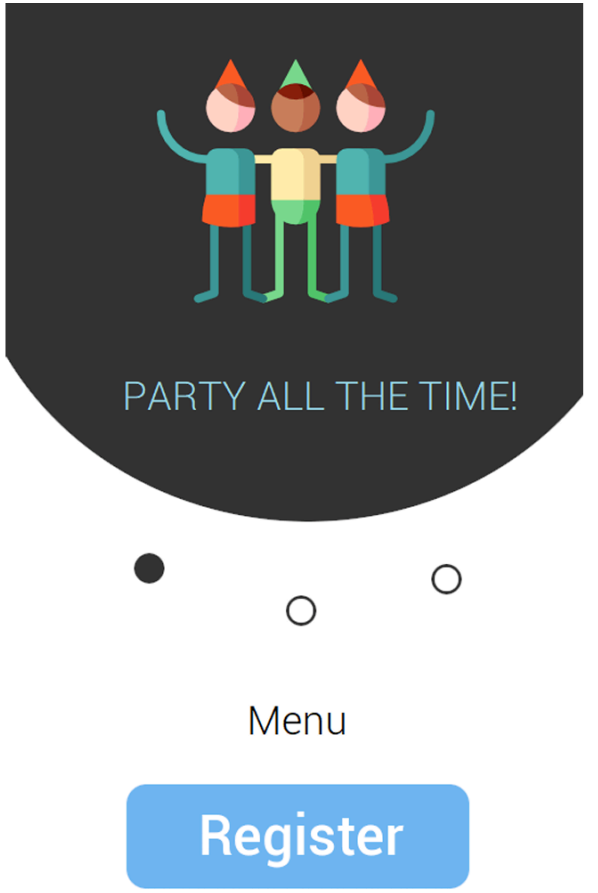
- Screen rotation (portrait ↔ landscape)
- Different device sizes (phone ↔ tablet ↔ desktop)
- Web & desktop support coming

Problem Example:

- List looks good in portrait
- Wasted space in landscape
- Need to adapt layout based on available space

Responsive UI





Adapt Layout:

In flutter, use `LayoutBuilder` to get parent constraints and adapt layout accordingly. It builds a widget tree that can depend on the parent widget's size.

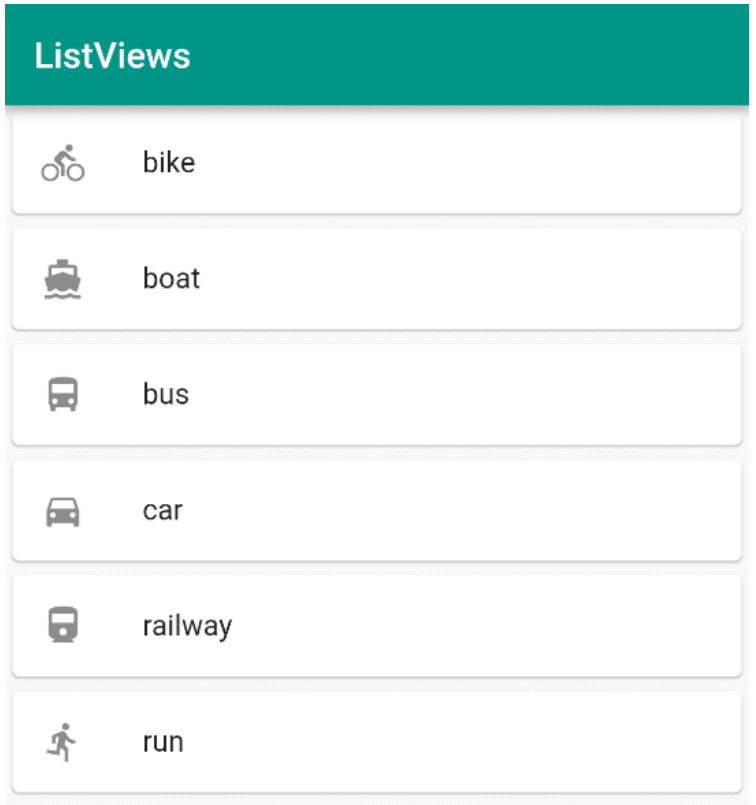
Type of Layout Builder

- `ListView` for narrow widths
- `GridView` for wider widths

Benefits:

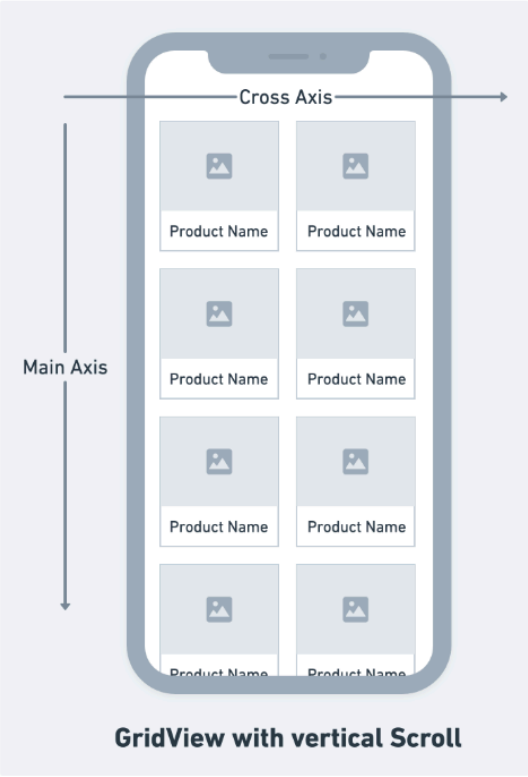
- Reacts to parent container size
- Not just screen size
- Handles padding/margins correctly

Example of ListView



Note: A `ListView` inside a `Column` has infinite height but is constrained in width by its parent. This mismatch causes layout errors if not handled properly.

Example of GridView



Note: A `GridView` inside a `Column` also has infinite height but is constrained in width by its parent. Proper handling of constraints is essential to avoid layout errors.

MediaQuery

MediaQuery is used to get information about the device's screen size and orientation.

- Returns device screen information
- `MediaQuery.of(context).size.width`
- Doesn't consider parent constraints

Example

`LayoutBuilder` based on screen width:

```
if (MediaQuery.of(context).size.width < 600) {
    return ListView(...);
} else {
    return GridView(...);
}
```

Orientation Detection

Device Orientation:

```
final orientation = MediaQuery.of(context).orientation;
if (orientation == Orientation.portrait) { ... }
```

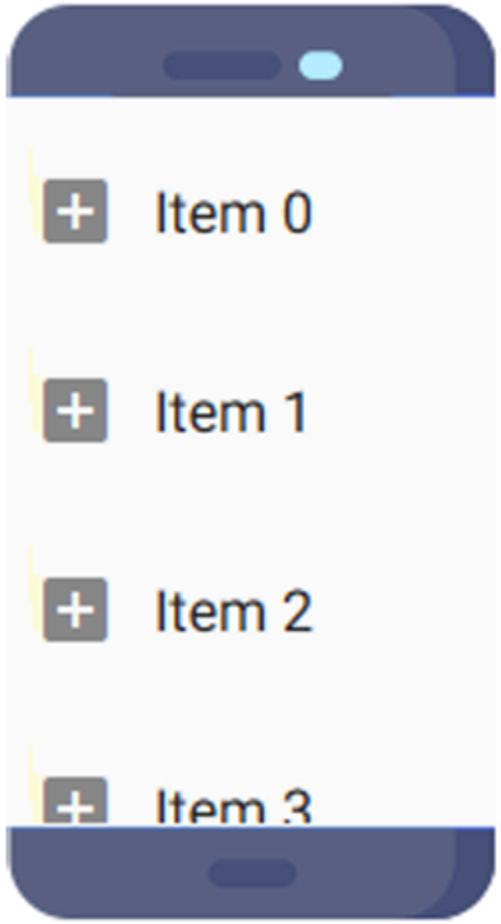
Widget Orientation:

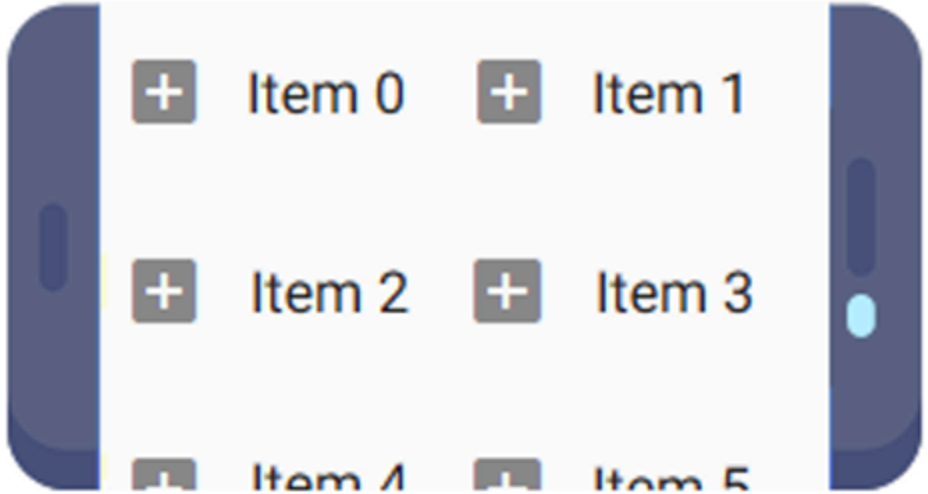
```
OrientationBuilder(
  builder: (context, orientation) {
    // Based on parent widget dimensions
  },
)
```

Key Difference:

- `MediaQuery`: Physical device orientation
- `OrientationBuilder`: Parent widget orientation

Example of OrientationBuilder





Theme

Theme is a way to define consistent colors, fonts, and styles across the entire app.

Example

```
MaterialApp(
  theme: ThemeData(
    primaryColor: Colors.blue,
    fontFamily: "Roboto",
  ),
)
```

Benefits:

- Consistent design throughout app
- Easy to change globally
- Automatic inheritance by widgets

Theme Customization

Predefined Themes:

```
ThemeData.dark()  // Dark theme
ThemeData.light() // Light theme
```

Custom Themes:

```
ThemeData(
  primaryColor: Colors.green,
  accentColor: Colors.orange,
  textTheme: TextTheme(
    bodyText1: TextStyle(fontSize: 18, color: Colors.black),
  ),
)
```

Cross-Platform Best Practices

Do:

- Create single, unified UI for all platforms
- Use responsive design with `LayoutBuilder`
- Test on multiple screen sizes & orientations
- Use themes for consistent styling

Don't:

- Maintain separate codebases per platform
- Hardcode widget sizes
- Assume portrait-only usage
- Ignore tablet/desktop layouts

Key Takeaways

1. **Material** for Android-style apps, **Cupertino** for iOS-style
2. **Scaffold** provides ready-made Material layout structure
3. **LayoutBuilder** is essential for responsive design
4. **Themes** ensure consistency and easy customization
5. **Handle constraints** properly (use `Expanded`, `shrinkWrap`)
6. **Design for all screen sizes** from the start

END