ERCİYES UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

MOBILE APPLICATION DEVELOPMENT

RESEARCH ASSIGNMENT

Layout and Design in Flutter

Instructor: Dr. Öğr. Üyesi FEHİM KÖYLÜ

Student ID: 1030510263

Student Name: Tugay COŞKUN

TABLE OF CONTENTS

Introduction

This report explores the layout and design capabilities of Flutter, a UI toolkit developed by Google for building natively compiled applications for mobile, web, and desktop from a single codebase. Flutter's unique approach to UI building provides developers with precise control over every pixel on the screen, enabling the creation of beautiful and highly customizable user interfaces.

Flutter's layout system is based on widgets that compose to form more complex UIs. Understanding how these widgets work together is essential for creating responsive and aesthetically pleasing applications. This report examines the fundamental concepts of Flutter's layout system, core layout widgets, responsive design techniques, Material Design implementation, and custom design possibilities.

Flutter Layout System

Basic Concepts

At the heart of Flutter's layout system is a constraint-based approach. Each widget receives constraints from its parent, determining the minimum and maximum width and height it can occupy. The widget then positions itself within these constraints and passes new constraints down to its children.

This constraint-based layout system follows these key principles:

1. **Constraints go down**: A parent widget passes constraints to its children.
2. **Sizes go up**: Children determine their sizes within the constraints and report back to the parent.
3. **Position is determined by the parent**: The parent decides where to place the child based on the child's size and the parent's layout logic.

Widget Tree

Flutter applications are built as a tree of widgets. Each widget in the tree may have child widgets, forming a hierarchical structure. The layout of the application is determined by how these widgets are arranged and configured.

```dart
Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      appBar: AppBar(
        title: Text('Flutter Layout Example'),
      ),
      body: Center(
        child: Text('Hello, Flutter!'),
      ),
    ),
  );
}
```

In this simple example, the widget tree consists of MaterialApp → Scaffold → (AppBar, Center) → Text. Each widget contributes to the overall layout according to its specific behavior.

Core Layout Widgets

Container

The Container widget is one of the most versatile layout widgets in Flutter. It combines common painting, positioning, and sizing functionality.

```
Container(
  margin: EdgeInsets.all(10.0),
  padding: EdgeInsets.symmetric(horizontal: 20.0, vertical: 15.0),
  decoration: BoxDecoration(
    color: Colors.blue,
    borderRadius: BorderRadius.circular(8.0),
    boxShadow: [
      BoxShadow(
        color: Colors.black26,
        offset: Offset(0, 2),
        blurRadius: 6.0,
      ),
    ],
  ),
  child: Text(
    'Flutter Container',
    style: TextStyle(color: Colors.white, fontSize: 18.0),
  ),
)
```

This Container creates a blue box with rounded corners, a drop shadow, margin, and padding, containing a text widget.

Row and Column

Row and Column are two fundamental layout widgets that arrange their children in horizontal and vertical arrays, respectively.

```
Column(
  mainAxisAlignment: MainAxisAlignment.center,
  crossAxisAlignment: CrossAxisAlignment.start,
  children: <Widget>[
    Text('First Item'),
    Text('Second Item'),
    Row(
      mainAxisAlignment: MainAxisAlignment.spaceEvenly,
      children: <Widget>[
        Icon(Icons.star),
        Icon(Icons.star),
        Icon(Icons.star),
      ],
    ),
  ],
)
```

This example creates a column of widgets with the first two being Text widgets, followed by a Row of Icon widgets. The `mainAxisAlignment` and `crossAxisAlignment` properties control how children are positioned along the main and cross axes.

Stack

The Stack widget allows children to be positioned on top of each other, relative to the edges of the box.

```
Stack(
  children: <Widget>[
    Container(
      width: 300,
      height: 200,
      color: Colors.amber,
    ),
    Positioned(
      top: 20,
      right: 20,
      child: Container(
        width: 100,
        height: 50,
        color: Colors.red,
      ),
    ),
    Positioned(
      bottom: 20,
      left: 20,
      child: Text('Stacked Text'),
    ),
  ],
)
```

This Stack creates a yellow background container with a red smaller container positioned at the top-right and text at the bottom-left.

Expanded and Flexible

Expanded and Flexible widgets allow children in a Row or Column to expand to fill available space.

```
Row(
  children: <Widget>[
    Container(
      width: 50,
      color: Colors.red,
      child: Text('Fixed'),
    ),
    Expanded(
      flex: 2,
      child: Container(
        color: Colors.green,
        child: Text('Takes 2/3 of remaining space'),
      ),
    ),
    Expanded(
      flex: 1,
      child: Container(
        color: Colors.blue,
        child: Text('Takes 1/3 of remaining space'),
      ),
    ),
  ],
)
```

In this example, the first container has a fixed width of 50 pixels, while the remaining space is divided between the green and blue containers in a 2:1 ratio.

Responsive Design in Flutter

MediaQuery

MediaQuery allows access to the size, orientation, and other properties of the current device screen, enabling responsive layouts.

```
Widget build(BuildContext context) {
  final screenSize = MediaQuery.of(context).size;

  return Container(
    width: screenSize.width * 0.8, // 80% of screen width
    height: screenSize.height * 0.3, // 30% of screen height
    color: Colors.purple,
    child: Center(
      child: Text(
        'Responsive Container',
        style: TextStyle(color: Colors.white),
      ),
    ),
  );
}
```

This example creates a container that always occupies 80% of the screen width and 30% of the screen height, regardless of device size.

LayoutBuilder

LayoutBuilder provides the parent's constraints to the builder function, allowing for even more fine-grained control over responsiveness.

```
LayoutBuilder(
  builder: (context, constraints) {
    if (constraints.maxWidth > 600) {
      // Wide layout
      return Row(
        children: [
          Expanded(child: LeftColumn()),
          Expanded(child: RightColumn()),
        ],
      );
    } else {
      // Narrow layout
      return Column(
        children: [
          LeftColumn(),
          RightColumn(),
        ],
      );
    }
  },
)
```

This example demonstrates a common pattern: showing content side-by-side on wide screens and stacked on narrow screens.

Material Design Implementation

Scaffold

Scaffold implements the basic Material Design visual layout structure, providing APIs for showing drawers, snack bars, and bottom sheets.

```
Scaffold(
  appBar: AppBar(
    title: Text('Material App'),
    actions: [
      IconButton(
        icon: Icon(Icons.search),
        onPressed: () {},
      ),
    ],
  ),
  drawer: Drawer(
    child: ListView(
      children: [
        DrawerHeader(
          decoration: BoxDecoration(color: Colors.blue),
          child: Text('Drawer Header'),
        ),
        ListTile(
          title: Text('Item 1'),
          onTap: () {},
        ),
      ],
    ),
  ),
  body: Center(child: Text('Body Content')),
  floatingActionButton: FloatingActionButton(
    onPressed: () {},
    child: Icon(Icons.add),
  ),
  bottomNavigationBar: BottomNavigationBar(
    items: [
      BottomNavigationBarItem(icon: Icon(Icons.home), label: 'Home'),
      BottomNavigationBarItem(icon: Icon(Icons.business), label: 'Business'),
    ],
    currentIndex: 0,
    onTap: (index) {},
  ),
)
```

This comprehensive example showcases a typical Material Design application layout with an app bar, drawer, body content, floating action button, and bottom navigation bar.

AppBar

AppBar is a horizontal bar typically displayed at the top of an application following Material Design guidelines.

```
AppBar(
  leading: IconButton(
    icon: Icon(Icons.menu),
    onPressed: () {},
  ),
  title: Text('Custom AppBar'),
  backgroundColor: Colors.teal,
  elevation: 4.0,
  actions: [
    IconButton(icon: Icon(Icons.favorite), onPressed: () {}),
    IconButton(icon: Icon(Icons.more_vert), onPressed: () {}),
  ],
  bottom: TabBar(
    tabs: [
      Tab(icon: Icon(Icons.directions_car)),
      Tab(icon: Icon(Icons.directions_transit)),
      Tab(icon: Icon(Icons.directions_bike)),
    ],
    controller: TabController(length: 3, vsync: this),
  ),
)
```

This AppBar features a leading menu icon, title, custom background color, elevation, action buttons, and a tab bar.

Card

Card is a Material Design panel with slightly rounded corners and an elevation shadow.

```
Card(
  elevation: 4.0,
  shape: RoundedRectangleBorder(
    borderRadius: BorderRadius.circular(10.0),
  ),
  child: Column(
    mainAxisSize: MainAxisSize.min,
    children: <Widget>[
      ListTile(
        leading: Icon(Icons.album),
        title: Text('The Enchanted Nightingale'),
        subtitle: Text('Music by Julie Gable. Lyrics by Sidney Stein.'),
      ),
      ButtonBar(
        children: <Widget>[
          TextButton(
            child: const Text('BUY TICKETS'),
            onPressed: () {},
          ),
          TextButton(
            child: const Text('LISTEN'),
            onPressed: () {},
          ),
        ],
      ),
    ],
  ),
)
```

This Card contains a ListTile with an icon, title, and subtitle, followed by a ButtonBar with two action buttons.

Custom Designs

CustomPainter

CustomPainter allows for the creation of custom snapes and graphics by drawing directly on the canvas.

```
class CirclePainter extends CustomPainter {
  @override
  void paint(Canvas canvas, Size size) {
    final paint = Paint()
      ..color = Colors.blue
      ..strokeWidth = 5
      ..style = PaintingStyle.stroke;

    final center = Offset(size.width / 2, size.height / 2);
    final radius = min(size.width, size.height) / 2 - 10;

    canvas.drawCircle(center, radius, paint);
  }

  @override
  bool shouldRepaint(CustomPainter oldDelegate) => false;
}

// Usage
CustomPaint(
  painter: CirclePainter(),
  size: Size(200, 200),
)
```

This example creates a custom painter that draws a blue circle with a 5-pixel stroke width.

Animations

Flutter provides a rich set of animation capabilities. Here's a simple example of an animated container:

```
class AnimatedContainerExample extends StatefulWidget {
  @override
  _AnimatedContainerExampleState createState() => _AnimatedContainerExampleState();
}

class _AnimatedContainerExampleState extends State<AnimatedContainerExample> {
  bool _expanded = false;

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: () {
        setState(() {
          _expanded = !_expanded;
        });
      },
      child: AnimatedContainer(
        duration: Duration(milliseconds: 300),
        curve: Curves.easeInOut,
        width: _expanded ? 200.0 : 100.0,
        height: _expanded ? 200.0 : 100.0,
        color: _expanded ? Colors.blue : Colors.red,
        child: Center(
          child: Text(
            _expanded ? 'Tap to shrink' : 'Tap to expand',
            style: TextStyle(color: Colors.white),
          ),
        ),
      ),
    );
  }
}
```

This example creates a container that smoothly animates its size and color when tapped.

Best Practices

When working with layouts and designs in Flutter, consider the following best practices:

1. **Think in terms of constraints**: Understand how constraints flow down the widget tree and how sizes flow back up.
2. **Use the right widget for the job**: Flutter provides specialized widgets for common layout patterns; use them instead of building everything from scratch.
3. **Keep the widget tree shallow**: Deeply nested widget trees can impact performance. Use methods and classes to break up complex UIs.
4. **Test on multiple screen sizes**: Ensure your layouts work well on various device sizes and orientations.
5. **Be mindful of overflows**: Use widgets like SingleChildScrollView to handle content that might not fit on smaller screens.
6. **Extract reusable widgets**: Create custom widgets for UI elements that are used repeatedly throughout your app.
7. **Use const constructors**: Where possible, use the const constructor to improve performance.

```
// Example of extracting a reusable widget
class CustomButton extends StatelessWidget {
  final String text;
```

```
    .4
    final VoidCallback onPressed;

    const CustomButton({
      Key? key,
      required this.text,
      required this.onPressed,
    }) : super(key: key);

    @override
    Widget build(BuildContext context) {
      return ElevatedButton(
        style: ElevatedButton.styleFrom(
          padding: EdgeInsets.symmetric(horizontal: 20, vertical: 12),
          shape: RoundedRectangleBorder(
            borderRadius: BorderRadius.circular(8),
          ),
        ),
        onPressed: onPressed,
        child: Text(text),
      );
    }
  }

  // Usage
  CustomButton(
    text: 'Press Me',
    onPressed: () {
      // Action
    },
  )
```

Conclusion

Flutter's layout and design system provides a powerful and flexible way to create beautiful user interfaces. By combining the core layout widgets with responsive design techniques and Material Design components, developers can create applications that look great on any device.

The constraint-based layout model, while initially challenging to grasp, offers precise control over how UI elements are positioned and sized. Custom painting and animation capabilities further extend what's possible, allowing for truly unique user experiences.

As with any UI framework, mastering Flutter's layout system requires practice and experimentation. By following the best practices outlined in this report and studying the provided examples, developers can create polished, responsive, and visually appealing mobile applications.