**Responsives Design in Flutter**

1. **Pixeldichte des Displays:**

Displays haben nicht nur unterschiedliche Größen, die Pixeldichte kann auch variieren. Wenn die Größe eines Elements festgelegt werden soll, z.B. Breite 100px, dann würde dies dazu führen, dass das Element auf einem Display mit einer höheren Pixeldichte kleiner ist als auf einem Display mit einer geringeren Pixeldichte. Um dieses Problem zu umgehen, gibt es in Flutter logische Pixel. Wird die Anzahl der logischen Pixel mit der devicePixelRatio(Anzahl Display Pixel pro logischer Pixel) multipliziert, erhält man die Anzahl der Display Pixel.

1. **Relative Positioning and Sizing:**

Die Größe des Displays variiert je nach Gerät.Um trotzdem ein responsives Design gewährleisten zu können, ist es notwendig, die Positionierung und Größenangaben eines Elements relativ zu einer Bezugsgröße anzugeben. Es folgen einige Widgets und Strategien, die dazu genutzt werden können:

* **Align (Relative Positioning)**

(<https://api.flutter.dev/flutter/widgets/Align-class.html>)

Das Align Widget erlaubt es das Child innerhalb sich selbst relativ zu positionieren. Es stehen verschiedene Werte für das Alignment zur Verfügung, wie bspw. Alignment.topRight oder Alignment.center. Alternativ kann, wenn das Child mittig platziert werden soll, das Center Widget verwendet werden.

 child: Align(

            alignment: Alignment.bottomLeft,

            child: Container(

              color: Colors.red,

              width: 150,

              height: 150,

              child: Center(

                child: Text(

                  "BottomLeft",

                  style: TextStyle(fontSize: 20),

               ),

              )

            ),

          ),

* **Row & Column with Flexible/Expanded and Flex Factor (Relative Sizing)**

(<https://api.flutter.dev/flutter/widgets/Flexible-class.html>

https://api.flutter.dev/flutter/widgets/Flexible/flex.html)Das Flexible/Expanded Widget ermöglicht es dem Child, den verbleibenden Platz entlang der Hauptachse zu füllen. Das Flexible/Expanded Widget muss Child eines der folgenden Elemente sein: Row, Column, Flex.

Es ist möglich ein Flex Value anzugeben. Ein Flex Value ermöglicht es, den Platz der einzelnen Elemente entlang der Achse relativ zu einander festzulegen. Haben drei Elemente innerhalb einer Spalte jeweils ein Flex Value 1, nimmt jedes Elemente 1/3 des verfügbaren Platzes ein. (Vergleichbar mit CSS Flexbox)

Expanded(

           child: Container(

             color: Colors.red,

             child: Center(

                 child: Text(

                  "1/3",

                ),

              ),

            ),

            flex: 1,

          ),

          Flexible(

            child: Container(

fit: FlexFit.tight,

              color: Colors.yellow,

              child: Center(

                child: Text(

                  "2/3",

                ),

              ),

            ),

            flex: 2,

          ),

* **FractionallySizedBox (Relative Sizing)**

(<https://api.flutter.dev/flutter/widgets/FractionallySizedBox-class.html>)

Die FractionallySizedBox ermöglicht es die Größe und Breite des Child Widgets als Bruchteil des verfügbaren Platzes anzugeben. In folgendem Beispiel nimmt der Container immer 2/3 des gesamten Displays ein.

        child: FractionallySizedBox(

          heightFactor: 0.66,

          // widthFactor: 0.4;

          child: Container(

            color: Colors.red,

          ),

        )

      )

-**MediaQuery**

(<https://api.flutter.dev/flutter/widgets/MediaQuery-class.html>)

Mit Hilfe von MediaQueries kann die Display Breite und Höhe abgefragt werden. Soll die Größe eines Elements relativ zur Display Größe sein, z.B. das Widget soll 1/3 der Höhe des Displays haben, kann die ermittelte Höhe (oder auch Breite) mit dem gewünschten Faktor 1/3 multipliziert werden und als Höhe (oder Breite) des Widgets gesetzt werden.

 child: Container(

            color: Colors.red,

            width: MediaQuery.of(context).size.width \* 0.33,

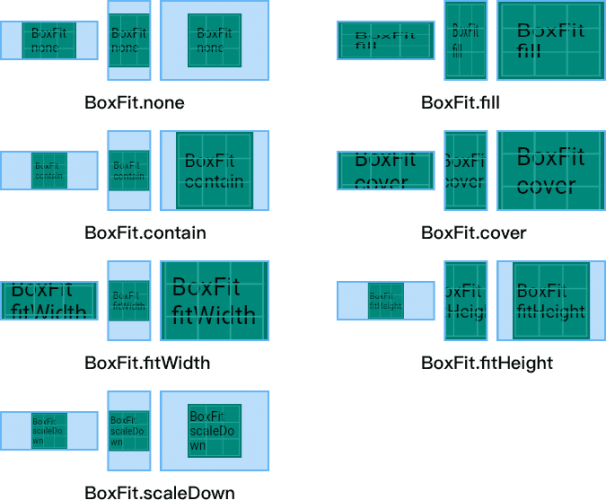
            height: MediaQuery.of(context).size.height \* 0.33,

        )

-**FittedBox (Auto Scaling)**

(https://api.flutter.dev/flutter/widgets/FittedBox-class.html)

FittedBox ermöglicht es die Größe des Childs automatisch an die Größe des Parent Widget anzupassen. Es kann zwischen verschiedenen BoxFit Typen gewählt werden. BoxFit.fill sorgt bspw dafür, dass das Child Widget immer das gesamte Parent Widget ausfüllt.



1. **Handling Orientation Change**

Der OrientationBuilder kann genutzt werden, um auf eine Änderung der Display Orientierung zu reagieren.

(<https://api.flutter.dev/flutter/widgets/OrientationBuilder-class.html>)

1. **Restrukturierung des Layouts**

Je nach Displaygröße oder Orientierung kann das Layout anders strukturiert werden, um den Platz optimal auszunutzen. Dazu könne die folgenden Klassen in Flutter verwendet werden: LayoutBuilder, OrientationBuilder, MediaQuery.

Wie diese Klassen genutzt werden könne wird im Folgenden an zwei Usecases demonstriert.

* **GridView**

(https://api.flutter.dev/flutter/widgets/GridView-class.html)

Die Anzahl der Spalten soll sich der verfügbaren Breite anpassen, um den Platz optimal auszunutzen. Dazu gibt es drei Möglichkeiten:

1. Die erste Möglichkeit besteht darin, die MediaQuery Klasse zu nutzen, um die Display Breite zu ermitteln. Anschließend kann je nach breite eine spezifische Anzahl an Spalten angeben werden. Diese Möglichkeit ist je noch nur bedingt responsiv, d.h. es wird nur eine neue Spalte hinzugefügt, wenn ein entsprechender Breakpoint existiert.

Widget build(BuildContext context) {

    return GridView.count(

        crossAxisCount: \_displayWidth < 600 ? 2 : 4,

        children: List.generate(100, (index){

          return Card(

            child: Container(

              alignment: Alignment.center,

              color: Colors.red[100 \* (index % 9)],

              child: new Text("Item $index"),

            ),

          );

        })

    );

  }

1. Die zweite Möglichkeit legt die Anzahl der Spalten dynamisch fest. Sobald in der Reihe genug Platz für eine neue Box ist, wird eine neue Spalte hinzugefügt.

double \_displayWidth = MediaQuery.of(context).size.width;

int \_cardWidth = 200;

int \_axisCount = (\_displayWidth/\_cardWidth).round();

Widget build(BuildContext context) {

    return GridView.count(

          crossAxisCount: \_axisCount,

          children: List.generate(100, (index){

            return Card(

              child: Container(

                alignment: Alignment.center,

                color: Colors.red[100 \* (index % 9)],

                child: new Text("Item $index"),

              ),

            );

          })

        );

  }

1. Die dritte Möglichkeit legt die Spalten Anzahl ebenfalls dynamisch fest. Der Unterschied zur zweiten Alternative ist, dass hier die Spaltenanzahl nicht manuell berechnet werden muss. Dies übernimmt das SliverGridDelegateWithMaxCrossAxisExtent Widget.

 return GridView.builder(

      itemCount: 100,

      gridDelegate: SliverGridDelegateWithMaxCrossAxisExtent(

maxCrossAxisExtent: \_cardWidth.toDouble(), mainAxisSpacing: 0.0,

crossAxisSpacing: 0.0, childAspectRatio: 1.0),

      itemBuilder: (context, index){

        return Card(

            child: Container(

              alignment: Alignment.center,

              color: Colors.red[100 \* (index % 9)],

              child: new Text("Item $index"),

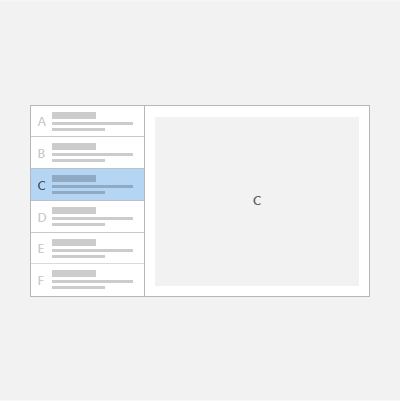
            ),

          );

      },

    );

* **Master Detail Layout**



Das Master Detail Layout ermöglicht es, auf größeren Geräten oder im horizontalen Modus, zwei Screens nebeneinander anzuzeigen.

Mit Hilfe der MediaQuery Klasse kann die kleinste Seite des Displays ermittelt werden. Anhand dieser Information kann nun entschieden werden, welches Layout geladen werden soll.

    Widget body;

    var shortestSide = MediaQuery.of(context).size.shortestSide;

    if (shortestSide < 600) {

      body = \_buildMobileLayout(list);

    } else {

      body = \_buildTabletLayout(list);

    }

buildMobileLayout() und buildTabletLayout() können nun jeweils ein Größen spezifisches Layout erstellen. Dieser Ansatz hat jedoch ein Problem. Wenn für beide Display Varianten ein komplett eigenständiges Layout, z.B. MobileLayout und TabletLayout erstellt wird, würde viel redundanter Code geschrieben werden. Ziel des responsiven Designs ist es jedoch, Größen spezifischen Code möglichst zu vermeiden.

Stattdessen sollte das Layout in mehrere kleine Fragmente unterteilt werden, die anschließend kombiniert werden können. In Anlehnung an die obere Grafik, könnte ein ListLayout und ein DetailLayout erstellt werden. Wenn nun das Mobile Layout erzeugt werden soll, wird entweder nur das ListLayout oder das DetailLayout angezeigt. Wenn hingegen das TabletLayout erzeugt werden soll, werden die beiden Layouts miteinander durch Nutzung des Row Widgets kombiniert. So können List- und Detail Layout in beiden Fällen wiederverwendet werden.

  Widget \_buildMobileLayout(var list) {

      return ListLayout(

…

itemSelectedCallback: (item) {

            Navigator.push(

                context,

                MaterialPageRoute(builder: (context) => DetailLayout(…)

              ),

            );

        }

);

  }

  Widget \_buildTabletLayout(var list) {

    return Row(

      children: <Widget>[

        Flexible(

          flex: 1,

          child: ListLayout(…),

        ),

        Flexible(

          flex: 3,

          child: DetailLayout(…),

        ),

      ],

    );

  }

}

1. **Weiter Widgets, die evtl in Frage kommen:**

* AxisAlignment zur relativen Positionierung innerhalb einer Row oder eines Columns (https://api.flutter.dev/flutter/rendering/CrossAxisAlignment-class.html)
* AspectRatio (https://api.flutter.dev/flutter/widgets/AspectRatio-class.html)
* CustomMultiChildLayout (https://api.flutter.dev/flutter/widgets/CustomMultiChildLayout-class.html)
* Wrap (https://api.flutter.dev/flutter/widgets/Wrap-class.html)