Flutter 개념(코딩쉐프 영상 참조)

2020년 7월 1일 수요일 오후 10:46

Widget이란??

- 독립적으로 실행되는 작은 프로그램
- 주로 바탕화면 등에서 날씨나 뉴스, 생활정보 등을 보여줌
- 그래픽이나 데이터 요소를 처리하는 함수를 가지고 있음.

Flutter에서 Widget이란?####

- UI를 만들고 구성하는 모든 기본 단위 요소
- 눈에 보이지 않는 요소들까지 위젯
- **모든 것이 위젯이다.**

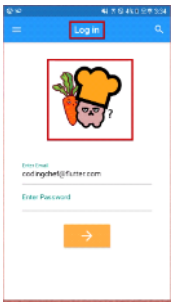
Flutter에서 중요한 위젯들

1) Stateless Widget
2) Stateful Widget
3) Inherited Widget

Stateless와 Stateful의 일반적인 의미

- **Stateful** : Value 값을 지속적으로 추적 & 보존 (**계속 움직임이나 변화가 있는 위젯**)
- **Stateless** : 이전 상호작용의 어떠한 값도 저장하지 않음.(**상태가 없는 정적인 위젯**)

Stateless Widget에 대해서



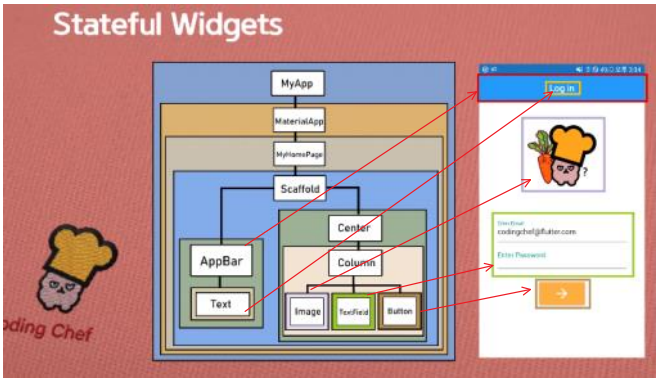
- 스크린상에 존재만 할 뿐 아무것도 하지 않음
- 어떠한 실시간 데이터도 저장하지 않음
- 어떤 변화(모양, 상태)를 발생시키는 value값을 가지지 않음.
- 그림에서 Log in 텍스트나 이미지는 그냥 화면 속에 존재만 할 뿐, 변화나 움직임은 없다.

Stateful Widget에 대해서

- 사용자의 interaction(상호작용)에 따라서 모양이 바뀜.
  (ex : 버튼을 누른다 던지, 라디오 버튼을 체크한다 던지 등)
- 데이터를 받게 되었을 때 모양이 바뀜.
- 옆 그림을 보면 사용자가 데이터를 입력(ID입력)을 할 때마다 화면에 변화가 있음.

Flutter Widget Tree

- **Widget들은 tree 구조**로 정리될 수 있다.
- **한 Widget내에 얼마든지 다른 Widget들이 포함**될 수 있다
- **Widget은 부모 위젯과 자식 위젯으로 구성**
- **Parent widget을 widget container(위젯을 내포한다는 의미)라고 부르기도 한다.**



- Log in 위젯 트리의 최상위 위젯은 MyApp 위젯이 존재한다. (반드시 최상위 위젯이 MyApp이라는 이름을 가질 필요는 없음.)
  MyApp위젯은 custom widget이다. **MyApp 위젯에서 MaterialApp 위젯이 build를 한다.**

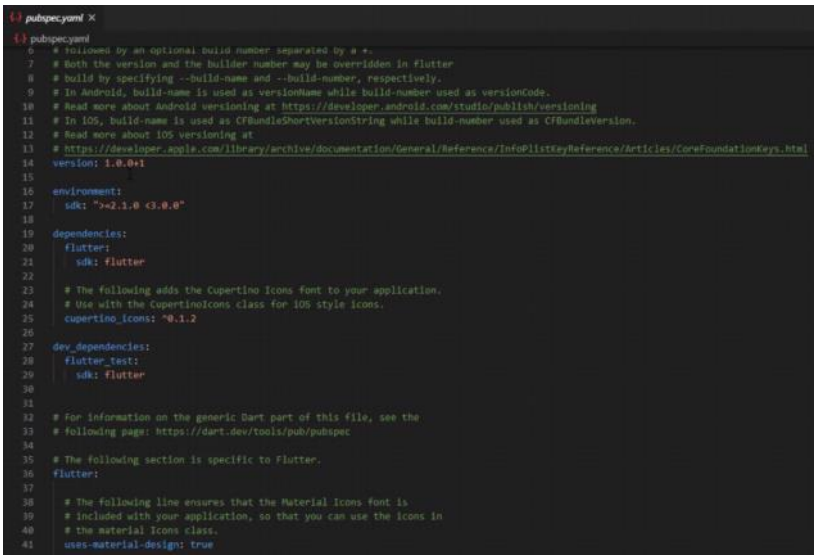
- **MaterialApp 위젯**은 **전체 앱을 감싸는 위젯**이라고 볼 수 있다. 그리고 MaterialApp 위젯을 통해서 Flutter SDK에서 제공하는 모든 위젯들을 사용할 수 있게 된다.

- MyHomePage위젯은 custom widget이다. **이 위젯부터 본격적으로 앱의 디자인과 기능들이 구현**된다. (물론 위젯 이름은 custom widget이니깐 개발자가 원하는 이름을 지정할 수 있다)

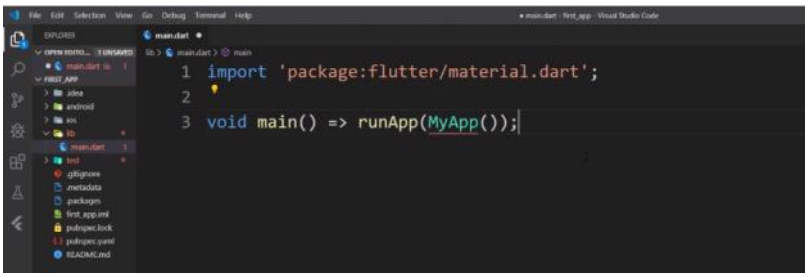
- **Scaffold 위젯(가장 중요한 위젯)**은 **앱 화면과 기능을 구성하기 위한 빈 페이지를 준비해**

주는 위젯이다. Scaffold 위젯이 생성되었으므로, 그 다음은 앱 화면의 최상단 부분을 차지하는 AppBar 위젯 생성이 가능하다.

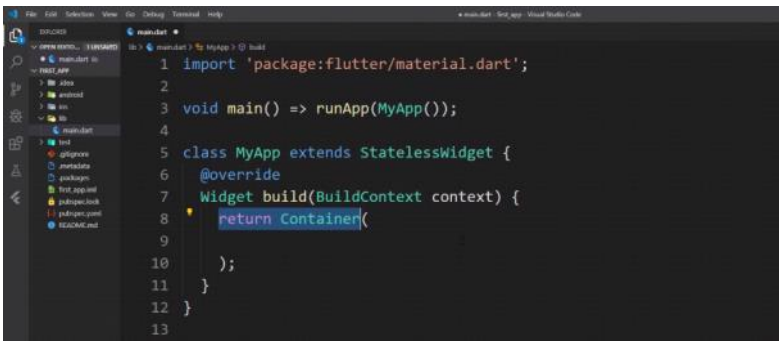
그 다음으로는 AppBar 위젯의 구성요소 중 하나인 Text 위젯이 위치한다.
그 다음으로는 Center 위젯, Column 위젯, Image, TextField, Button 위젯이 오도록 구성되어 있다.



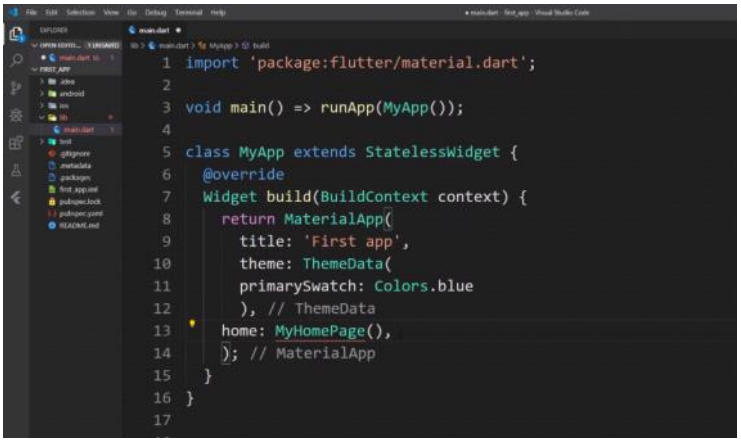
- Pubspec.yaml파일은 프로젝트의 메타데이터들을 정의하고 관리한다. 즉, 프로젝트의 버전, 사용환경, dart의 버전이나, 각종 dependancy와 써드파티 & 라이브러리 등을 이곳에서 정의한다.



- Main은 타 언어들과 똑같이 진입구 역할을 하는 함수임.
- Flutter에서의 widget들은 전부 트리구조이다.
- **=> 는 기존 함수형태에서 중괄호를 대신하여 간략하게 쓰는 표현**이다.
- **runApp은 flutter에서 최상위 함수**이며, **인자 값은 widget**을 가져야 한다. **최초 한번만 호출**하면 된다.
- MyApp은 flutter framework에서 제공하는 widget이 아니라, custom widget이기 때문에 사용자가 직접 만들어야 하는 widget이다.(widget이름을 굳이 MyApp으로 할 필요가 없다는 뜻이기도 하다.)
- 클래스 명은 가독성을 위해 대문자를 사용하자.(ex : MyApp)

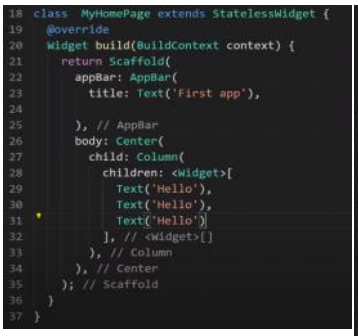


- **최상위 위젯인 MyApp 위젯은 App의 Layout을 build 하는 역할. 즉, 뼈대를 만드는 역할만** 하므로 **어떤 변화나 움직임이 없는 정적인 widget**이다. => **즉, stateless widget으로 지정을 해야 한다.**
- **모든 custom 위젯**은 **또 다른 위젯을 return 하는 build라는 함수를 가지고 widget을 return 한다.**(위 그림에서는 Container widget을 return 하게 되는 꼴임)

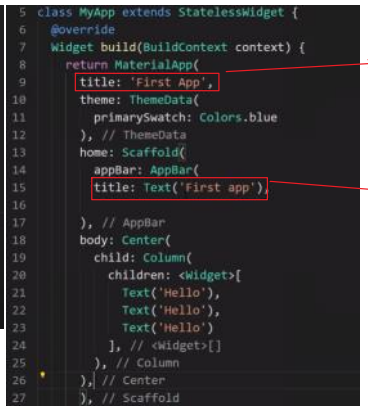


```dart
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'First app',
      theme: ThemeData(
      primarySwatch: Colors.blue
      ), // ThemeData
    home: MyHomePage(),
    ); // MaterialApp
  }
}
```

- 그러나 우리가 필요한 것은 앱을 만들기 위해서 import한 flutter material library를 사용할 수 있는 기능을 가진 MaterialApp widget이다. 그래서 Container를 MaterialApp으로 바꾸도록 한다.
- **MaterialApp 위젯은 widget tree 중에 2번째에 위치하는 위젯**이며, **실질적으로 모든 위젯을 감싸고 있는 widget**이다. 그리고 모든 위젯이 그렇듯이 인자 값을 가져야 함.
- MaterialApp 위젯은 title 이라는 문자열을 인자로 가지며, 여러 인자 값(theme 등)을 가질 수 있다.
- theme 인자 값인 ThemeData 위젯은 기본적인 디자인 테마를 지정.
- **primarySwatch는 견본을 의미**한다. **특정한 색상을 default로 사용할 색상으로 지정해서 앱에서 사용하겠다는 의미**를 가지는 인자.
- **Home 위젯은 앱이 정상적으로 실행되었을 때, 가장 먼저 화면에 보여주는 경로**.(이 위젯이 없으면 앱이 실행되어도 화면에 아무것도 나타나지 않음. **반드시 필요한 위젯**.)
  (위 코드에서는 custom widget인 MyHomePage위젯을 앱이 실행되었을 때, 가장 먼저 화면에 보여주도록 해놓음)



```dart
class  MyHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('First app'),

      ), // AppBar
      body: Center(
        child: Column(
          children: <Widget>[
            Text('Hello'),
            Text('Hello'),
            Text('Hello')
          ], // <Widget>[]
        ), // Column
      ), // Center
    ); // Scaffold
  }
}
```



```dart
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'First App',
      theme: ThemeData(
        primarySwatch: Colors.blue
      ), // ThemeData
      home: Scaffold(
        appBar: AppBar(
        title: Text('First app'),

      ), // AppBar
      body: Center(
        child: Column(
          children: <Widget>[
            Text('Hello'),
            Text('Hello'),
            Text('Hello')
          ], // <Widget>[]
        ), // Column
      ), // Center
      ), // Scaffold
```

앱 자체의 제목(작업 표시줄에 뜨는 이름)

화면 최상단의 AppBar 내의 텍스트

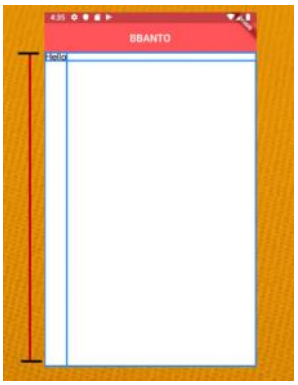
- MyHomePage 위젯은 custom 위젯이므로 **class 형태로 위젯을 만들어서 반환**해줘야 한다.
- **Scaffold는 앱 화면에 다양한 요소를 배치하도록 도와주는 역할**을 함.
- **appBar**는 Scaffold의 인자 값(위젯)이며, **앱 화면의 상단 바를 표현**한다.
- Title은 AppBar의 인자 값이며, 상단 바 안에 문자열을 표현한다.
- Scaffold 의 인자 값(위젯)인 **body는 본격적으로 앱 화면을 구성하는 시작점**이며, **Scaffold 위젯 내에서 가장 중요한 위젯**이다.
- **Center 위젯은 body 위젯의 자식 위젯**이다. Center 위젯을 사용하여 **화면에 표현할 요소를 가로축 기준으로 정중앙에 위치**시킨다.
- **Child(자식을 하나만 가질 경우)** 또는 **Children(자식 위젯을 여러개 가질 경우)**에는 **Center위젯의 자식 위젯으로 어떤 위젯을 사용할 건지 명시**한다.
- **Column 위젯**은 Center 위젯의 자식 위젯이며, **요소를 세로로 나열할 때 사용**한다.
- **Children: <Widget>[]의 의미**는 **여러 개의 자식 위젯을 가질 것**이며, 그 **자식 위젯을 [ ] 안에 배열처럼 나열하라는 의미**이다.
- **MaterialApp 위젯을 사용한다는 것은 flutter 프레임워크가 제공하는 모든 기본 위젯들을 사용할 수 있다는 의미**이기 때문에 **home 위젯에 MyHomePage대신 Scaffold 위젯을 사용할 수 있으며**, 그 결과는 위 코드상에서는 같다.

  즉, 따로 MyHomePage클래스를 생성해서 위젯을 꾸며 MyHomePage 위젯 내에서 Scaffold를 return 할 필요 없이 바로 상위 위젯인 MaterialApp 위젯에서 Scaffold 위젯을 꾸며 줄 수 도 있다는 의미이다.

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

void main() =>  runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Charactor card',
      home: MyCard(),
    ); // MaterialApp
  }
}
```

```dart
class MyCard extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('BBANTO'),
        centerTitle: true,
        backgroundColor: Colors.redAccent,
        elevation: 0.0,
      ), // AppBar
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text('Hello'),
            Text('Hello'),
            Text('Hello'),
          ], // <Widget>[]
        ), // Column
      ) // Center

    ); // Scaffold
  }
}
```

- **centerTitle는 boolean 값**을 가지며, true를 값으로 주면 **appBar 내의 문자열이 가운데로 정렬**된다.
- **backgroundColor**는 **위젯 영역의 배경색**을 설정.
- **Elevation은 높이 위치를 지정**하며, **소수점 까지 수치로 지정 가능**하다.
- **Padding**은 html에서 padding과 똑같다. **화면 상하좌우 기준으로 얼만큼 떨어져서 위치할 건지를 설정**한다. **Insets는 어떤 것을 삽입한다는 의미**이며, **LTRB는 Left, Top, Right, Bottom**을 의미.
- **mainAxisAlignment.center** 속성은 **위젯을 화면에서 세로축을 기준(Column 위젯 내에서)**으로 **상단, 중단, 하단으로 나눈 후, 중단의 가운데에 정렬**할 때 사용.
  Row 위젯 내에서는 가로축 기준으로 가운데 정렬을 수행함.



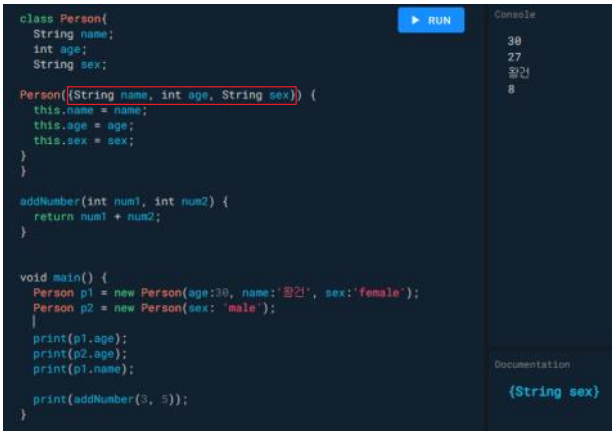
- Column 위젯 내에서 child 위젯으로 text 위젯을 만들고 실행시킨 모습인데, **column 위젯은 자식 위젯들(위 코드에서 Text위젯들이 해당)에게 세로축은 높이에 대한 제한을 주지 않으며, 가로축은 넓이에 대한 확실한 제약을 준다.**

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

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      title: 'Character Card',
      home: Grade(),
    ); // MaterialApp
  }
}

class Grade extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.amber[800],
      appBar: AppBar(
        title: Text('BBANTO'),
        centerTitle: true,
        backgroundColor: Colors.amber[700],
        elevation: 0.0,
      ), // AppBar
      body: Padding(
        padding: EdgeInsets.fromLTRB(30.0, 40.0, 0.0, 0.0),
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: <Widget>[
            Center(
              child: CircleAvatar(
                backgroundImage: AssetImage('assets/icon1.jpg'),
                radius: 60.0,
              ), // CircleAvatar
            ), // Center
            Divider(
              height: 60.0,
              color: Colors.grey[850],
              thickness: 0.5,
              endIndent: 30.0,
            ), // Divider
            Text(
              'NAME',
              style: TextStyle(
                color: Colors.white,
                letterSpacing: 2.0,
              ), // TextStyle
            ), // Text
            SizedBox(
              height: 30.0,
            ), // SizedBox
            Row(
              children: <Widget>[
                Icon(Icons.check_circle_outline),
                SizedBox(
                  width: 10.0,
                ), // SizedBox
                Text(
                  'using lightsaber',
                  style: TextStyle(fontSize: 16.0, letterSpacing: 1.0),
                ), // Text
              ], // <Widget>[]
            ), // Row
            Row(
              children: <Widget>[
                Icon(Icons.check_circle_outline),
                SizedBox(
                  width: 10.0,
                ), // SizedBox
                Text(
                  'face hero tattoo',
                  style: TextStyle(fontSize: 16.0, letterSpacing: 1.0),
                ), // Text
              ], // <Widget>[]
            ), // Row
            Row(
              children: <Widget>[
                Icon(Icons.check_circle_outline),
                SizedBox(
                  width: 10.0,
                ), // SizedBox
                Text(
                  'fire flames',
                  style: TextStyle(fontSize: 16.0, letterSpacing: 1.0),
                ), // Text
```

```
), // Divider
Text(
  'NAME',
  style: TextStyle(
    color: Colors.white,
    letterSpacing: 2.0,
  ), // TextStyle
), // Text
SizedBox(
  height: 10.0,
), // SizedBox
Text(
  'BBANTO',
  style: TextStyle(
    color: Colors.white,
    letterSpacing: 2.0,
    fontSize: 28.0,
    fontWeight: FontWeight.bold), // TextStyle
), // Text
SizedBox(
  height: 30.0,
), // SizedBox
Text(
  'BBANTO POWER LEVEL',
  style: TextStyle(
    color: Colors.white,
    letterSpacing: 2.0,
  ), // TextStyle
), // Text
SizedBox(
  height: 10.0,
), // SizedBox
```

```
), // SizedBox
Text(
  'fire flames',
  style: TextStyle(fontSize: 16.0, letterSpacing: 1.0),
), // Text
], // <Widget>[]
), // Row
Center(
  child: CircleAvatar(
    backgroundImage: AssetImage('assets/icon2.png'),
    radius: 40.0,
    backgroundColor: Colors.amber[800],
  ), // CircleAvatar
), // Center
], // <Widget>[]
)), // Column, Padding
); // Scaffold
}
}
```

- **SizedBox 위젯**은 height / width 속성을 사용해서 **위젯과 위젯 사이의 공간을 조절**한다.
- **CrossAxisAlignment.start**는 **화면 왼쪽 시작점에 맞춰 정렬.**
- Text 위젯의 style : **Textstyle 위젯**은 텍스트를 꾸며줌.
- **letterSpacing** 속성은 **각 글자 간의 폭을 조절**.
- **Row**는 **아이콘과 텍스트를 가로로 나열**할 때 사용.
- **.(점)**은 **위젯이 가지고 있는 여러가지 속성 or 기능 or 아이템들을 사용**하고 싶을 때 사용한다. (ex : Colors.blue 또는 Icons.check 등)
- **CircleAvatar** : **불러올 이미지를 원 형태**로 불러온다.
- **AssetImage()** : **페이지 내에 이미지를 삽입**할 때, 주로 사용하는 위젯이다.
- Radius : 원 형태의 이미지 크기 조절
- **Divider** 위젯 : **화면에 가로축 선을 하나 생성**함.
- **Endindent** 속성 : Divider 위젯으로 생성한 선이 화면 끝에서 얼마나 떨어져서 표현될지 결정
- **debugShowCheckedModeBanner** : 화면 우상단 끝에 debug라고 써있는 것을 없앰.




```
class Person{
  String name;
  int age;
  String sex;

Person({String name, int age, String sex}) {
  this.name = name;
  this.age = age;
  this.sex = sex;
}
}

addNumber(int num1, int num2) {
  return num1 + num2;
}


void main() {
  Person p1 = new Person(age:30, name:'황건', sex:'female');
  Person p2 = new Person(sex: 'male');

  print(p1.age);
  print(p2.age);
  print(p1.name);

  print(addNumber(3, 5));
}
```

```
Widget build(BuildContext context) {
  return new MaterialApp(
    title: 'First app',
    theme: ThemeData(
      primarySwatch: Colors.blue
    ), // ThemeData
    home: MyHomePage(),
  ); // MaterialApp
}
}
```

```
class MyHomePage extends StatelessWidget {
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: new AppBar(
      title: new Text('First app'),
    ), // AppBar
```

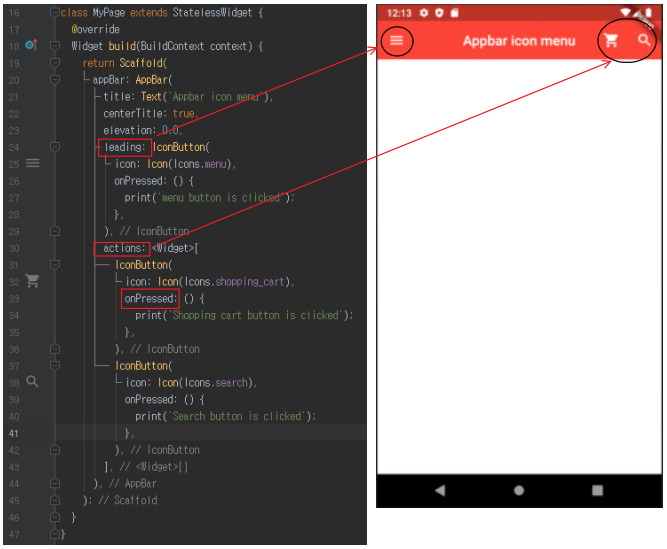
- **Person 생성자의 매개변수에 { } 중괄호를 해주는 부분을 named Argument**라고 한다. 인스턴스 생성 시, 정의한 생성자 파라미터에 대한 무조건 모든 값을 대입하지 않아도 되며, 선택적으로 생성자 파라미터를 초기화 할 수 있다. 물론 기존처럼 한번에 초기화 해도 된다.
  Named Argument는 플러터에서 중요한 개념인데 그 이유는 **각 위젯의 속성 값에 위젯을 생성할 때, 이 Named Argument 과정을 거쳐서 초기화 후, 위젯으로 반환해주는 것**이기 때문임. 그럼 왜 이런 개념을 도입했을까?

  기존의 타 언어 C++이나 JAVA 같은 경우, 인스턴스 생성 시, 생성자를 통해 클래스에서 정의한 멤버 값을 한번에 모두 초기화(안해주면 default 생성자가 호출되어 초기화를 수행) 해줘야만 했는데, Named Argument 방식에서는 선택적으로 원하는 생성자만 초기화 할 수 있다.

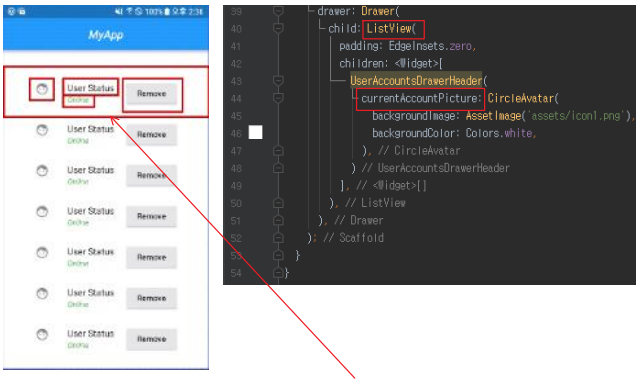
  즉, 여태까지 사용했던 모든 위젯들이 결국 클래스를 통해 생성된 인스턴스 였던 것임. 그래서 위 코드를 보면 **return new MaterialApp에서 new를 붙이는 이유도 결국 인스턴스를 생성해서 반환해주기 위함(new 생략가능)**이다. **즉, 위젯은 인스턴스와 똑같다.**

  옆 코드로 설명하자면, **Text 위젯**은 **Text 생성자를 통해서 문자열('First app')을 입력 받아 초기화 되어 인스턴스(위젯)이 생성된 후, 위젯이 반환된 것**이다.

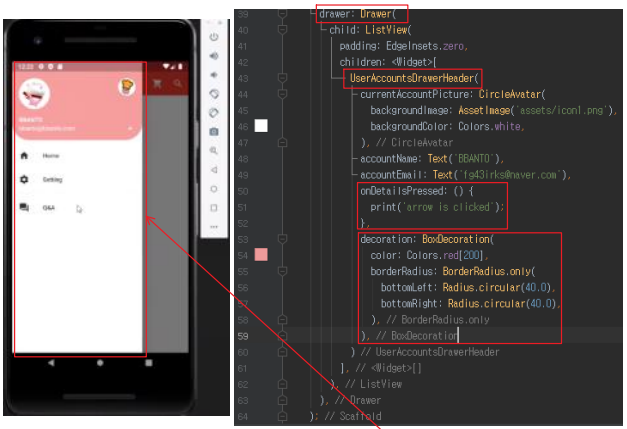
  AppBar 위젯은 AppBar 생성자를 통해서 여러 개의 argument들을 named Argument 형태로 필요한만큼 선택적으로 사용해서 생성된 인스턴스(위젯)인 것임.
- Tip : ctrl키 누른 상태에서 각 위젯들을 클릭하면, 각 위젯들이 가지고 있는 속성 및 소스코드를 볼 수 있음.



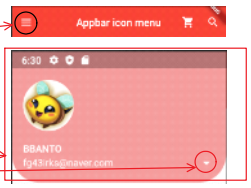
- **leading** 속성은 **위젯이나 아이콘을 왼쪽에 위치**시키는 역할을 한다.
- **actions**는 **복수의 아이콘 버튼 등을 오른쪽에 배치**할 때 사용한다.
- **onPressed**는 함수의 형태로 **일반 버튼이나 아이콘 버튼을 터치했을 때 일어나는 이벤트를 정의**함.



- Flutter는 margin이나 padding 값에 신경 쓸 필요 없이 **ListTile 위젯을 통해 복수의 위젯들을 나열**할 수 있게 해준다. (**각 하나의 List를 Flutter에서는 ListTile이라고 부른다**.)
- **currentAccountPicture** 속성은 현재 사용자의 이미지를 가져오는 역할을 수행.

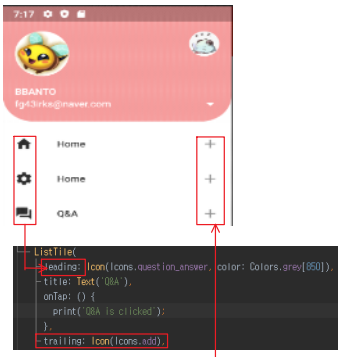


- **위 이미지와 같이 사이드에 부메뉴로서 사용되는 위젯이 Drawer 위젯**이다. BBANTO라는 텍스트와 이미지가 있는 분홍색으로 색칠된 부분이 **Drawer Header** 부분임.
- 그리고 이 **Drawer 위젯**을 사용하면 자동으로 **햄버거 메뉴**도 생성한다.
- **drawer 위젯**은 우리가 구글 앱스토어 사용 시, 좌 상단 햄버거 메뉴 버튼을 누르면 **사용자 계정정보가 나오는 것처럼 똑같은 형태를 그려주는 위젯**이다.
  UserAccountDrawerHeader 위젯은 drawer 위젯으로 생성된 창에서 분홍색으로 색칠된 부분. 즉, **헤더 영역을 꾸며주는 위젯**이다.
- **onDetailedPressed** 속성은 클릭했을 때, 부메뉴를 보여주는 역할을 수행.
- **Decoration** 속성은 **원하는 위젯 영역을 꾸며주는 역할**을 수행.
  위 코드에서는 UserAccountsDrawerHeader 영역을 꾸며주게 됨. **boderRadius** 속성은 사각형이였던 분홍색 영역 **모서리 부분을 둥글게** 만들어 줌.



```dart
class UserAccountsDrawerHeader extends StatefulWidget {
  /// Creates a material design drawer header.
  ///
  /// Requires one of its ancestors to be a [Material] widget.
  const UserAccountsDrawerHeader({
    Key key,
    this.decoration,
    this.margin = const EdgeInsets.only(bottom: 8.0),
    this.currentAccountPicture,
    this.otherAccountsPictures,
    @required this.accountName,
    @required this.accountEmail,
    this.onDetailsPressed,
    this.arrowColor = Colors.white,
  }) : super(key: key);

  /// The header's background. If decoration is null then a [BoxDecoration]
  /// with its background color set to the current theme's primaryColor is used.
  final Decoration decoration;

  /// The margin around the drawer header.
  final EdgeInsetsGeometry margin;
```

- **Required**가 붙은 argument는 **위젯 내에서 반드시 사용되어야 하는 argument**이다.



- **onTap과 onPressed의 차이**는 사용자의 동작의 차이에 있다. **onTap은 길게 누르기 or 두 번 탭하기 등 과 관련된 이벤트에 반응하는 기능을 담당**하며, **onPressed는 주로 버튼에 사용되며, 버튼을 누르면 그에 대한 반응**을 보여준다. 즉, 둘의 기능은 거의 동일하다. 다만 어떤 이벤트를 주느냐에 따라 반응을 할지 안 하는지에 차이가 있을 뿐이다.
- 그럼 **ListTile 위젯에서 onTap을 사용하는 이유**는? => 일반 버튼을 클릭하는 액션이 아닌 **tap 하거나 길게 누르기와 같은 이벤트를 감지할 수 있는 기능이 built in 되어있기 때문**이다.
- **Trailing 속성**은 **ListTile 영역 내에서 맨 우측에 위젯을 위치**시키는 기능을 수행.


```dart
Widget build(BuildContext context) {
  return Scaffold(
```

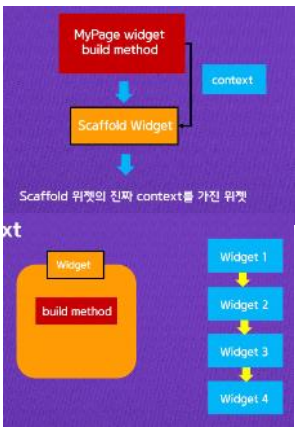
```dart
class MyPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
```

정리하자면, 해당 위젯을 만드는 class에서 build메소드를 통해 인자 값으로 context를 생성했다면, 그 context는 해당 클래스의 context다.

Ex) Class **MyApp** extends StatelessWidget {
    @override
    Widget build(BuildContext **context**) {
        return Scaffold(
            .....
    }
}
위 코드에서, context는 MyApp에 대한 context이다.

1) **BuildContext**는 **widget tree에서 현재 widget의 위치를 알 수 있는 정보**이다.
Flutter에서는 모든 widget은 Build method(함수)를 가질 수 있음. 그리고 이러한 위젯들이 계층구조를 이룬다.
Build 메소드는 Scaffold라는 위젯을 리턴 하는데 **widget tree 상에서 어디에 위치하는가에 대한 정보를 가지고있는 context라는 것을 Scaffold 위젯에 넣어서 return을 해준다는 의미**임.

2) **BuildContext**는 **statelss위젯**이나 **state 빌드 메서드에 의해서 return된 위젯의 부모가 된다.**
예를 들면, StatelessWidget으로 MyPage라는 커스텀위젯을 생성했는데, 이 커스텀 위젯도 자신만의 BuildContext 타입의 context를 가지고 있다. 그리고 build 메서드를 통해서 Scaffold가 return된 모습이다.
그리고 **return된 Scaffold위젯은 부모 위젯인 MyPage위젯의 context를 그대로 물려받게 된다.**(상속)
정리하자면, 자식 위젯은 부모 위젯의 context를 상속받는다.

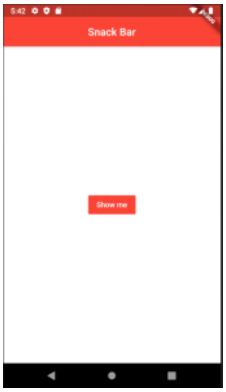


- (특징 1) **BuildContext에 대한 이해와 사용이 까다로운 이유** : Scaffold 위젯이 widget tree 상에서 자신의 widget-tree 상에서의 위치 정보인 context를 가지고 있어야하는게 상식적으로 당연하다고 생각되는데, **실제로는 Scaffold 위젯의 context는 부모 위젯의 context를 상속받으니 헷갈리는 것**임.
- **그럼 어떻게 Scaffold의 context(위치정보)를 얻을 수 있는가?** => (특징 1)의 의미는 결국 자식 위젯의 context는 부모 위젯의 context를 물려받기 때문에 **Scaffold Widget의 context를 알아내기 위해서는 Scaffold의 자식 위젯을 build method를 통해 생성 후, return** 한다면, 그 자식 위젯은 Scaffold 위젯의 context를 물려받게 된다.

Snack Bar & BuildContext

— 배울 내용 —

- Snack Bar를 사용하면서 BuildContext 활용에 대한 이해도를 향상 시키고, 플러터에서 자주 쓰이는 **Scaffold.of메소드**에 대해서 알아본다.
- **스낵바란 ?** : **스크린 하단에 간단한 메시지를 띄우는 기능**이다.




```
Widget build(BuildContext context) {
```

- **BuildContext와 context의 차이**는 무엇인가?
  => **context는 BuildContext 클래스의 instance**라고 생각하면 된다.
  그리고 **context의 이름은 사용자가 원하는 대로 바꿔도 무방**하다.

- **FlatButton은 앱의 body부분에서 자주 사용**되는데, **기능상으로 동일한 Raised button과 Floating action button이 있다. 버튼 디자인 차이**만 있다.

- Flutter 공식 문서를 보면 **Scaffold.of(context)메소드를 통해서 Scaffold 위치를 참조한 후, showSnackBar()메소드를 통해 snackbar를 구현**해야 한다.

  왜 이렇게 해야 하는가? => **Snackbar는 Scaffold내에서 구현**되어야 하기 때문에 **Flutter가 Scaffold의 context를 참조해서 snackbar를 그리기 때문**이다.

  즉, 앱 스크린에 표현되는 위젯들은 widget-tree에 맞춰서 계층적으로 표현되고, 그러기 위해선 **각 위젯들의 위치정보(context)를 알아야 widget-tree 계층 순서에 맞게 위젯을 화면에 그릴 수 있기 때문에 표현하려는 위젯의 개수가 많을수록 context에 의존적**일 수 밖에 없다.

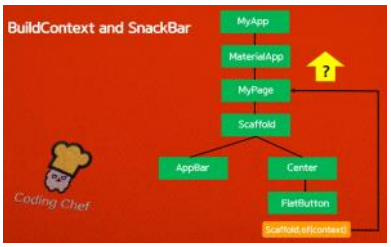
  즉, Scaffold.of(context) method는 "현재 주어진 context에서 위로 올라가면서 가장 가까운 Scaffold를 찾아서 반환하라."라는 의미이다. **예를 들면, Scaffold.of(ctx1) 이면, widget-tree상에서 ctx1의 위치부터 찾아 올라가라는 의미**.

- Scaffold.of 메소드 외에 여러 of 메소드가 존재한다. 그렇기에 Something.of(context)라고 표현하기도 한다. (Theme.of 등. 등 존재)

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

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Snack Bar',
      theme: ThemeData(primarySwatch: Colors.red),
      home: MyPage(),
    ); // MaterialApp
  }
}
```

```dart
class MyPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Snack Bar'),
        centerTitle: true,
      ), // AppBar
      body: Center(
        child: FlatButton(
          child: Text(
            'Show me',
            style: TextStyle(color: Colors.white),
          ), // Text
          color: Colors.red,
          onPressed: () {
            Scaffold.of(context).showSnackBar(
              SnackBar(
                content: Text('Hellow'),
              ), // SnackBar
            );
          },
        ), // FlatButton
      )); // Center, Scaffold
  }
}
```

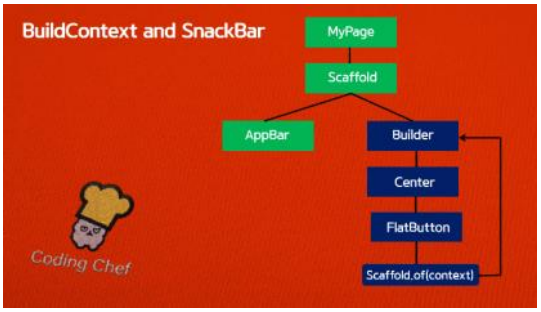
```
Exception caught by gesture
The following assertion was thrown while handling a gesture:
Scaffold.of() called with a context that does not contain a Scaffold.

No Scaffold ancestor could be found starting from the context that was passed to Scaffold.of(). This usually happens when the context provided is
 from the same StatefulWidget as that whose build function actually creates the Scaffold widget being sought.

There are several ways to avoid this problem. The simplest is to use a Builder to get a context that is "under" the Scaffold. For an example of
 this, please see the documentation for Scaffold.of():
  https://api.flutter.dev/flutter/material/Scaffold/of.html
A more efficient solution is to split your build function into several widgets. This introduces a new context from which you can obtain the
 Scaffold. In this solution, you would have an outer widget that creates the Scaffold populated by instances of your new inner widgets, and then
 in these inner widgets you would use Scaffold.of().
A less elegant but more expedient solution is assign a GlobalKey to the Scaffold, then use the key.currentState property to obtain the
 ScaffoldState rather than using the Scaffold.of() function.

The context used was: MyPage
When the exception was thrown, this was the stack:
#0      Scaffold.of (package:flutter/src/material/scaffold.dart:1316:5)
#1      MyPage.build.<anonymous closure> (package:practice_snackbar/main.dart:32:24)
#2      _InkResponseState._handleTap (package:flutter/src/material/ink_well.dart:706:14)
#3      _InkResponseState.build.<anonymous closure> (package:flutter/src/material/ink_well.dart:789:36)
#4      GestureRecognizer.invokeCallback (package:flutter/src/gestures/recognizer.dart:182:24)
...
Handler: "onTap"
Recognizer: TapGestureRecognizer#4b6d6
  debugOwner: GestureDetector
  state: possible
  won arena
  finalPosition: Offset(212.2, 374.1)
  finalLocalPosition: Offset(51.5, 10.3)
  button: 1
  sent tap down
```



- Scaffold 위젯의 자식 위젯인 FlatButton 위젯의 context를 사용해서 widget-tree 상의 Scaffold 위치를 찾아 거슬러 올라가려고 했는데, 정작 Scaffold.of( )메소드가 사용한 context는 MyPage의 context를 사용했기 때문에 **에러가 발생**.

- 여기서 **핵심**은, build메소드를 호출했을 때, 인자 값으로 전달되는 BuildContext 타입의 context는 return되는 Scaffold widget의 context가 아니라 , build 메소드를 호출한 MyPage위젯의 BuildContext라는 것임.

- 정리하자면, **위 코드의 경우 Scaffold.of( ) 메소드는 widget-tree 상에서 MyPage 위젯의 위치부터 거슬러 올라가면서 Scaffold를 탐색**하게 된다. **하지만 부모 위젯들인 MaterialApp 위젯과 MyApp 위젯에는 Scaffold 위젯이 존재하지 않는 상태**이고, 만약 있다고 해도 원하는 위젯 내의 Scaffold위젯이 아니므로 원하는 결과를 얻을 수 없다.



- 위와 같은 문제를 해결하기 위해 등장한 것이 **Builder 위젯**이다.
  이전과 같이 Scaffold.of( )메소드를 사용했는데, Scaffold의 context를 찾기 애매한 경우에 **지금까지 사용해왔던 context가 무엇이던지 간에 Builder 위젯을 사용해서 Builder 위젯의 context를 기반의 새로운 widget-tree를 따로 생성하는 것**이다.
  이렇게 되면 Scaffold.of( )메소드는 MyPage의 context를 사용하지 않고 Builder 위젯의 context를 사용할 수 있게 된다.

  정리하자면, Builder 위젯을 생성함으로써, 더 쉽게 원하는 context를 찾아갈 수 있다고 생각하면 된다.

```dart
class MyPage extends StatelessWidget {
  @override
```

- 기존과 다른 점은 body argument 에서 바로 Center 위젯을 불러와서 FlatButton 과 SnackBar를 구현했던 것과 달리,

정리하자면, Builder 위젯을 생성함으로써, 더 쉽게 원하는 context를 찾아갈 수 있다고 생각하면 된다.

```dart
class MyPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Snack Bar'),
        centerTitle: true,
      ), // AppBar
      body: Builder(
        builder: (BuildContext ctx) {
          return Center(
            child: FlatButton(
              child: Text(
                'Show me',
                style: TextStyle(color: Colors.white),
              ), // Text
              color: Colors.red,
              onPressed: () {
                Scaffold.of(ctx).showSnackBar(
                  SnackBar(
                    content: Text('Hellow'),
                    ), // SnackBar
                );
              },
            ), // FlatButton
          ); // Center
        },
      )); // Builder, Scaffold
  }
}
```

- 기존과 다른 점은 body argument 에서 바로 Center 위젯을 불러와서 FlatButton 과 SnackBar를 구현했던 것과 달리, **Builder 위젯 내에서 Center 위젯을 return 한 게 전부**이다.
- BuildContext의 context명을 ctx로 바꾼 이유는 MyPage 위젯의 Context명과 구분하기 위해서 바꿔준 것 뿐이다.
- 그럼 기존과 달리 어떻게 동작하는가?
  => **Scaffold.of ( ) 메소드가 Builder의 ctx를 찾아서 거슬러 올라가 widget-tree 상에서 Scaffold 위젯을 찾게된다.**

```dart
    body: Center(
      child: FlatButton(
        child: Text(
          'Show me',
          style: TextStyle(color: Colors.white),
        ), // Text
        color: Colors.red,
        onPressed: () {
          Scaffold.of(context).showSnackBar(
            SnackBar(
              content: Text('Hellow'),
              ), // SnackBar
          );
        },
      ), // FlatButton
    )); // Center, Scaffold
  }
}
```

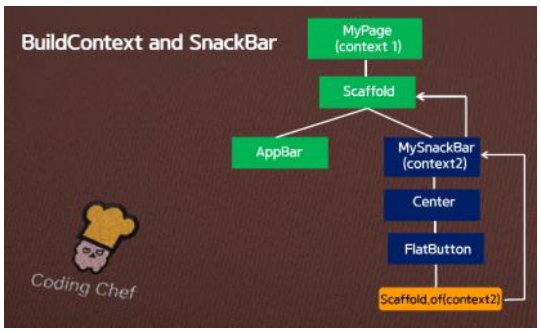
Snack Bar & Toast message

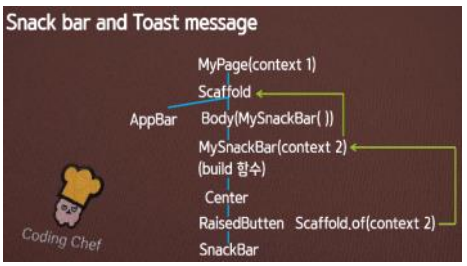
— 목 차 —

1) **빌더 위젯 없이 Snack Bar 만들기**
2) **Toast message 구현하기**

- ***Toast Message 란?*** : **사용자에게 짧은 메세지형식으로 정보를 전달하는 팝업**을 의미. 정말 많이 사용되는 기능이다.

1) Builder 위젯 없이 Snack Bar 구현하는 방법.





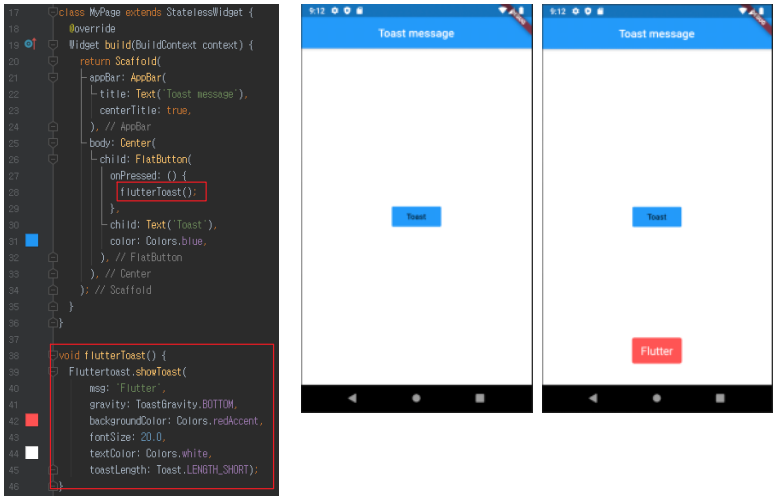
- 개념은 전에 사용했던 Builder 위젯과 동일하다. **차이점** 단지 Builder 위젯 생성 대신, 사용할 위젯(커스텀 위젯)인 MySnackBar 위젯을 widget-tree 상에 생성해서 Scaffold의 context를 찾는 것 뿐이다.


```dart
class MyPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Snack Bar'),
        centerTitle: true,
      ), // AppBar
      body: MySnackBar(),
    ); // Scaffold
  }
}

class MySnackBar extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Center(
      child: RaisedButton(
        child: Text('Show me'),
        onPressed: () {
          Scaffold.of(context).showSnackBar(
            SnackBar(
              content: Text(
                'Hellow',
                textAlign: TextAlign.center,
                style: TextStyle(color: Colors.white),
              ), // Text
              backgroundColor: Colors.teal,
              duration: Duration(milliseconds: 1000),
            ), // SnackBar
          );
        }), // RaisedButton
    ); // Center
  }
}
```

2) Toast Message 구현하기

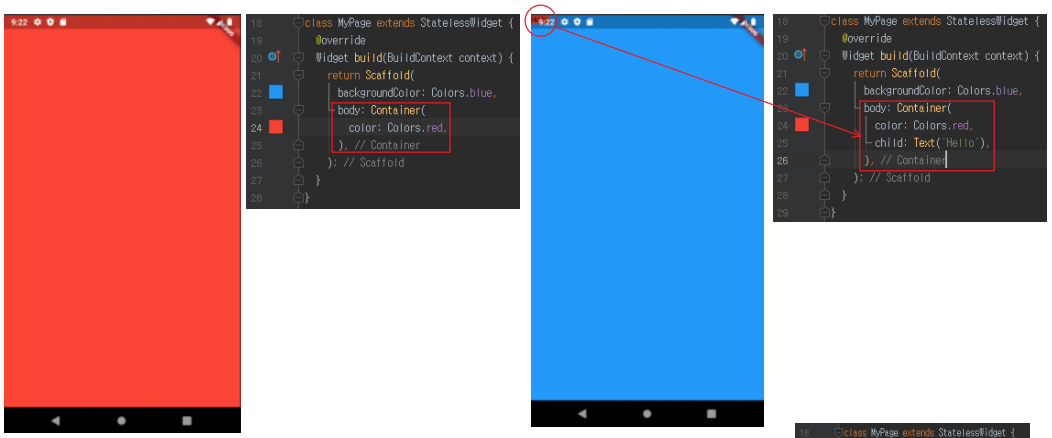
```dart
import 'package:fluttertoast/fluttertoast.dart';
```

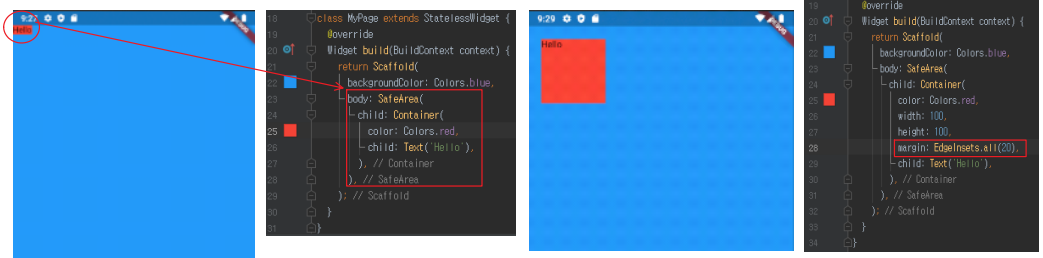
- **Toast Message를 사용**하기 위해선 **Fluttertoast라는 라이브러리를 import**해야 한다.
- **Toast Message는 widget-tree와는 상관없으므로, Toast message를 실행할 함수를 만들어서 onPressed 함수 내로 넣어주면 됨.**



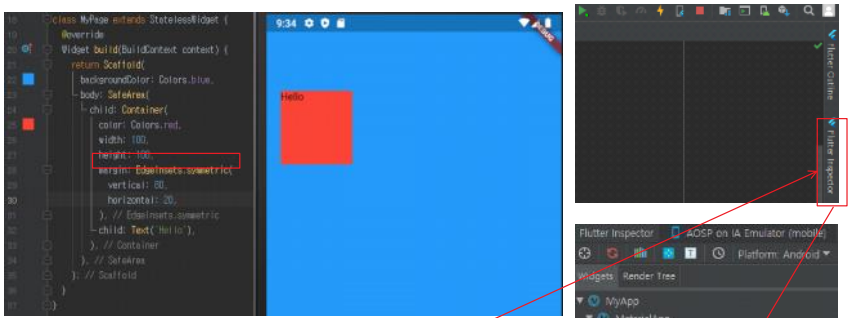
Container 위젯에 대해서

- **Container 위젯**은 무조건 페이지 내에서 **최대한의 공간을 차지하려는 특징**이 있다.
- **Container 위젯은 child(자식위젯)를 가지게 되면 그 child 크기로 줄어들게 된다.**
- **Container 위젯**은 **오직 한 개의 child위젯만**을 가질 수 있다.





- **SafeArea위젯**은 개발자가 **보여주기를 원하는 컨텐츠가 화면 밖으로 빠져나가지 않게 경계를 지정**해준다.
- **EdgeInsets.all** 은 위젯 위치를 인자 값 만큼 스크린 좌 상단 가장자리로부터 떨어진 위치에 위젯을 배치한다.

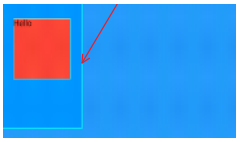


- **EdgeInsets.symmetric** 위젯을 사용해서 **수직/수평으로 80/20만큼 떨어진 위치에 배치**했다. 그런데 **이 위젯은 모든 방향 기준으로 위치를 변경**하는데, 화면 상으로 봤을 땐 우측과 하단은 80/20으로 보기엔 거리가 상당히 떨어져 있다.

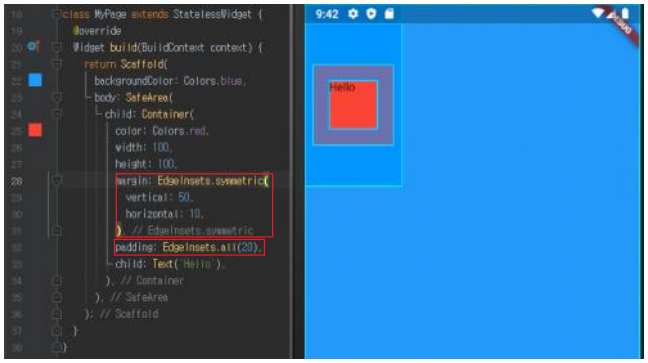
  => 오른쪽 그림을 보면 Flutter Inspector메뉴가 있는데 메뉴에 들어가면 확인할 수 있다. 아래와 같이 위젯 주변에 **형광색 테두리로 가이드라인**이 그려지는데 **그 영역을 기준으로 EdgeInsets가 동작**하는 것을 알 수 있다.

- 그럼 **Container 안에 있는 텍스트 'Hello'의 위치는 어떻게 조절**하는가?
  => **padding을 사용**하면 된다.

아래와 같이 위젯 주변에 **형광색 테두리로 가이드라인**이 그려지는데 **그 영역을 기준으로 EdgeInsets가 동작**하는 것을 알 수 있다.

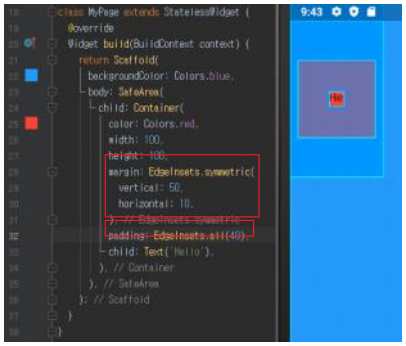


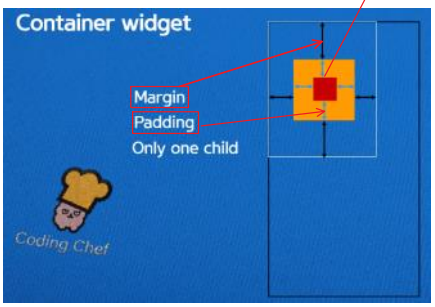
- 그럼 **Container 안에 있는 텍스트 'Hello'의 위치는 어떻게 조절**하는가?
  => **padding을 사용**하면 된다.

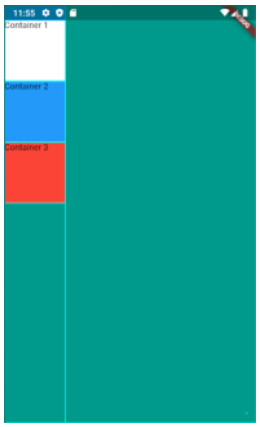


- **Container가 스크린의 가장자리에서 일정 간격을 가지게 하고 싶은 경우 : margin** 사용
- 그 Container가 포함하고 있는 요소(위 그림에선 텍스트인 'Hello')가 Container의 가장자리에서 일정간격을 가지게 하고 싶은 경우 => padding 사용.
- 정리하자면, **margin은 widget의 바깥쪽 간격**을, **padding은 widget의 안쪽 간격**을 조정한다.

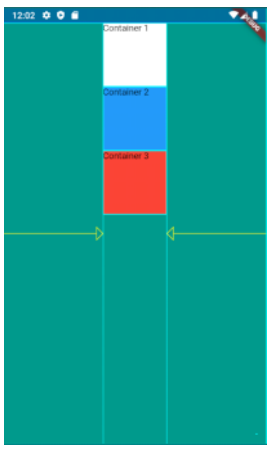
Container를 의미.

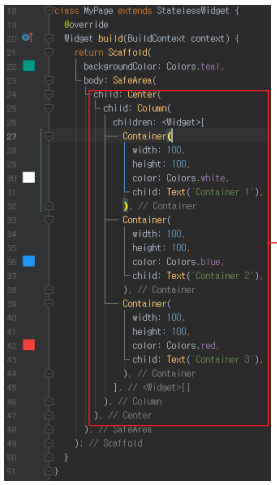


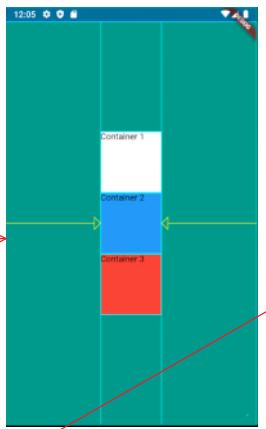


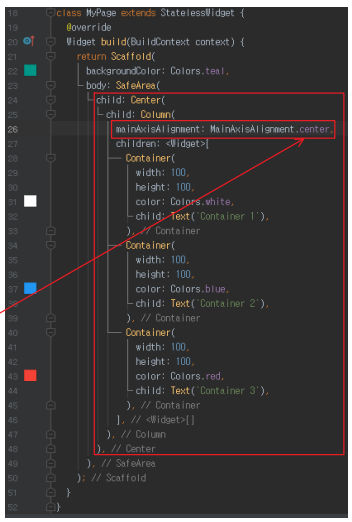


- **Column의 특징**은 **스크린의 세로축 방향으로 가능한 모든 공간을 차지**한다. **그러나 가로축은 위젯의 크기만큼 가진다**.
- 특히, **Center위젯의 child 위젯으로 Column위젯이 사용될 경우**, 이 개념을 잘 알고서 **mainAxisAlignment 속성을 사용**해야 **column 위젯을 화면 정중앙에 위치**시킬 수 있다.

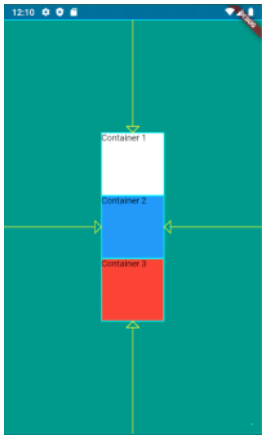








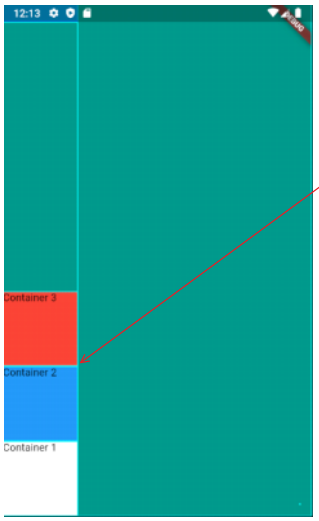
- 위 이미지를 보면 **Center 위젯의 자식 위젯으로 Column 위젯이 사용된 경우**, 행 기준의 가운데 정렬이 아닌 **열 기준의 가운데 정렬**을 하게 된다.
- 그럼 **Container 들을 화면 정가운데 위치시키려면 어떻게 해야 할까**?
  => **Column 위젯에 mainAxisAlignment속성의 center값을 주면 해결**된다.




```dart
class MyPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.teal,
      body: SafeArea(
        child: Center(
          child: Column(
            mainAxisSize: MainAxisSize.min,
            children: <Widget>[
              Container(
                width: 100,
                height: 100,
                color: Colors.white,
                child: Text('Container 1'),
              ), // Container
              Container(
                width: 100,
                height: 100,
                color: Colors.blue,
                child: Text('Container 2'),
              ), // Container
              Container(
                width: 100,
                height: 100,
                color: Colors.red,
                child: Text('Container 3'),
              ), // Container
            ], // <Widget>[]
          ), // Column
        ), // Center
      ), // SafeArea
    ); // Scaffold
  }
}
```

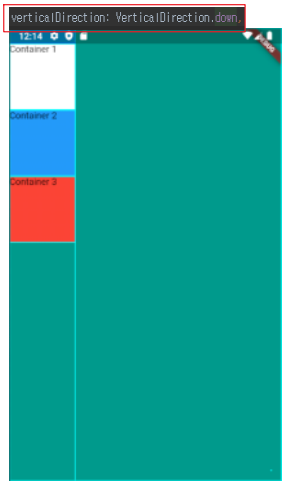
- 그런데 Column 위젯이 세로축의 모든 영역을 차지하고 있는데, 개발자는 그걸 원하지 않는 상황이다. 그럼 어떻게 해야 할까?

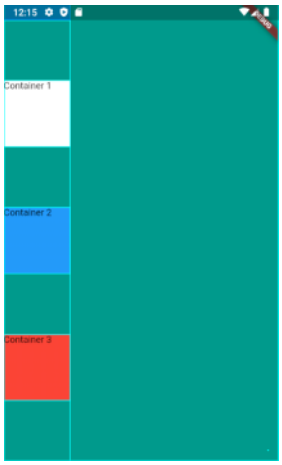
  => **mainAxisSize.min 를 사용**하면 된다. 그럼 **배치하는 위젯의 크기만큼의 세로축 영역을 가지게 된다.**




```dart
class MyPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.teal,
      body: SafeArea(
        child: Column(
          verticalDirection: VerticalDirection.up,
          children: <Widget>[
            Container(
              width: 100,
              height: 100,
              color: Colors.white,
              child: Text('Container 1'),
            ), // Container
            Container(
              width: 100,
              height: 100,
              color: Colors.blue,
              child: Text('Container 2'),
            ), // Container
            Container(
              width: 100,
              height: 100,
              color: Colors.red,
              child: Text('Container 3'),
            ), // Container
          ], // <Widget>[]
        ), // Column
      ), // SafeArea
    ); // Scaffold
  }
}
```

```
verticalDirection: VerticalDirection.down,
```



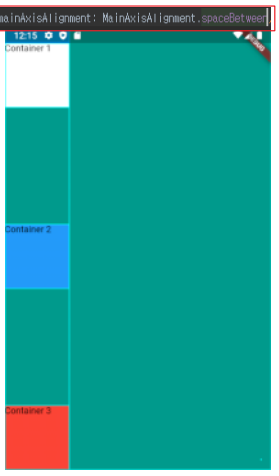


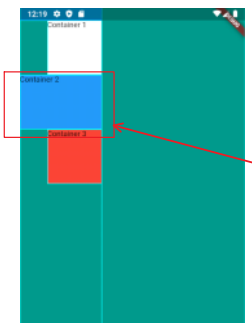

```dart
class MyPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.teal,
      body: SafeArea(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.spaceEvenly,
```

- 위와 같은 경우, Login with google 이라는 텍스트를 가운데에 정렬 시켜 주진 못한다.
  Why? 버튼 위젯 내에 이미지 1개와 텍스트 위젯 하나 뿐이므로, 텍스트를 가운데 정렬 시킬 수 없다.

- 하지만 위와 같이 구글 로고 이미지를 오른쪽에 하나 더 삽입하여 텍스트를 가운데로 정렬한 후, Opacity 위젯의 opacity 속성을 사용하여 이미지를 투명하게 해주면 가운데로 정렬이 가능하다.

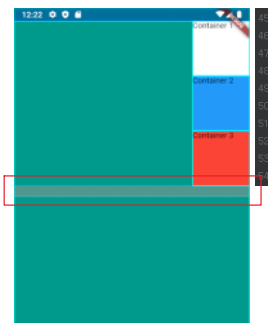
```
mainAxisAlignment: MainAxisAlignment.spaceBetween,
```






```dart
class MyPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.teal,
      body: SafeArea(
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.end,
```

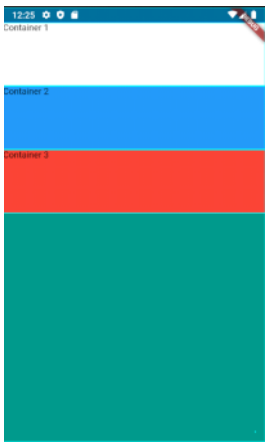
- **crossAxisAlignment**의 역할을 확실히 알기 위해
  Container2의 width 넓이를 다른 Container보다 50을 더 키운 상태.

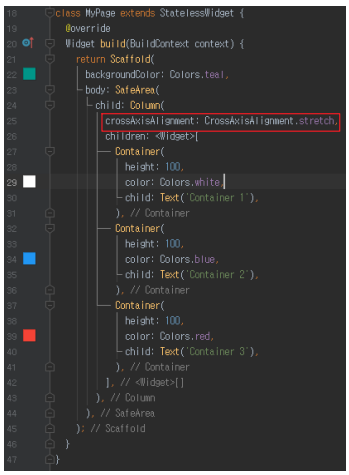



```dart
            Container(
              width: double.infinity,
              height: 20.0,
            ), // Container
          ], // <Widget>[]
        ), // Column
      ), // SafeArea
    ); // Scaffold
  }
}
```

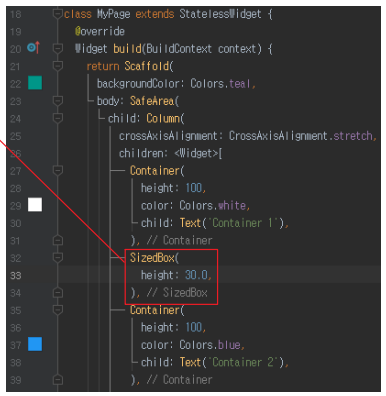
- **Container3 아래** 하나의 Container를 **width 속성에 infinity 값을 부여**하여 **새로운 Container를 생성한 상태** 이다.
  새롭게 생성한 이 Container는 invisible Container라고도 한다.
  그 이유는 **말 그대로 화면 상으로는 보이지 않지만 Container(공간)을 차지하는 역할**을 하기 때문이다.

새로운 Container를 생성한 상태 이다.
새롭게 생성한 이 Container는 invisible Container라고도 한다.
그 이유는 **말 그대로 화면 상으로는 보이지 않지만 Container(공간)을 차지하는 역할**을 하기 때문이다.





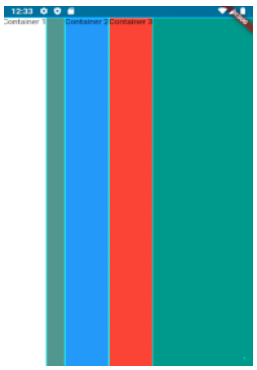
```
class MyPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.teal,
      body: SafeArea(
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.stretch,
          children: <Widget>[
            Container(
              height: 100,
              color: Colors.white,
              child: Text('Container 1'),
            ), // Container
            Container(
              height: 100,
              color: Colors.blue,
              child: Text('Container 2'),
            ), // Container
            Container(
              height: 100,
              color: Colors.red,
              child: Text('Container 3'),
            ), // Container
          ], // <Widget>[]
        ), // Column
      ), // SafeArea
    ); // Scaffold
  }
}
```

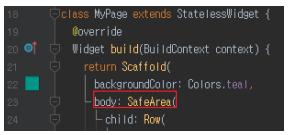


```
class MyPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.teal,
      body: SafeArea(
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.stretch,
          children: <Widget>[
            Container(
              height: 100,
              color: Colors.white,
              child: Text('Container 1'),
            ), // Container
            SizedBox(
              height: 30.0,
            ), // SizedBox
            Container(
              height: 100,
              color: Colors.blue,
              child: Text('Container 2'),
            ), // Container
```

- **Container 사이에 간격을 생성**하고 싶다면 Container와 Container 사이에 **SizedBox 위젯**을 생성하면 된다.

- **Strech 속성 값은 width 속성 값이 필요 없다.**
  그 이유는 **가로축으로 가로질러 뻗어 나가는 stretch 값을 사용**했기 때문에 **자식 위젯들인 Container에 width 값을 통해 속성값을 사용할 필요가 없기 때문**이다.





```
class MyPage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.teal,
      body: SafeArea(
        child: Row(
```

- Column을 Row로 바꿔준 후, 생성된 결과이다.
- **이러한 레이아웃을 연습할 수 있는 사이트**
  => https://medium.com/flutter-community/...

Navigator

— 목차 —

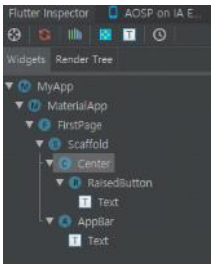
1) **Route의 개념**
2) **Navigator의 정의와 push, pop 함수, stack 자료구조**
3) **MaterialPageRoute위젯과 context**
4) **페이지 이동 가능 구현 완성**

- Flutter에서의 **Route란?** : **스마트폰 상에서 보여지는 하나의 페이지** or **화면** or **Activity**가 될 수 있다.(단어는 전부 같은 뜻이며, 개발환경에 따라 다르게 불림)

- **Navigator란?** : **모든 웹페이지를 관리하는 위젯**이며, **Stack이라는 자료구조 형식으로 route 객체들을 관리**하는 것. 그리고 **이러한 관리를 위해서 push와 pop 메소드를 제공**한다.

- **MaterialPageRoute**는 앱 상에서 페이지를 이동할 때, Android에서 기본적으로 제공하는 페이지 이동 시, 애니메이션 효과를 제공한다.

- **기타 설명 :**
  Android 나 IOS 플랫폼에서 기본적으로 제공되는 애니메이션 효과가 있는데 **Android는 페이지 이동 시, 위로 올라오면서 Fade-in** 되며, **페이지에서 나갈 때**는 아래로 내려가면서 **Fade-out**되는 기능이 내장되어 있다.
  IOS에서는 좌우로 화면이 움직이면서 페이지 이동이 되는 애니메이션 효과를 가지고 있다.




```dart
class FirstPage extends StatelessWidget {
  @override
  Widget build(BuildContext context2) {
    return Scaffold(
      appBar: AppBar(
        title: Text('First page'),
      ), // AppBar
      body: Center(
        child: RaisedButton(
          child: Text('Go to the Second page'),
          onPressed: () {
            Navigator.push(context2,
                MaterialPageRoute(builder: (BuildContext context) {
              return SecondPage();
            })); // 버튼을 눌렀을 때, 이동할 페이지 // MaterialPageRoute
          } //
          ), // RaisedButton
      ), // Center
    ); // Scaffold
  }
}
```

- First page에서 버튼을 누르지 않은 상태에서의 widget-tree 상태이다.
- 이 First page를 하나의 Route라고 보면 된다. 즉 Route는 하나의 앱 페이지를 의미.

  정리하자면, 앱 페이지를 구성하기 위한 Scaffold 위젯을 return 하는 각각의 Custom widget 들이 Route라고 생각하면 된다.

  그리고 이 Route들은 MaterialApp 위젯의 child 위젯이어야만 하는데, 그 이유는 Widget-tree 구조상 모든 위젯들은 MaterialApp 위젯이 감싸고 있는 구조이기 때문이다.

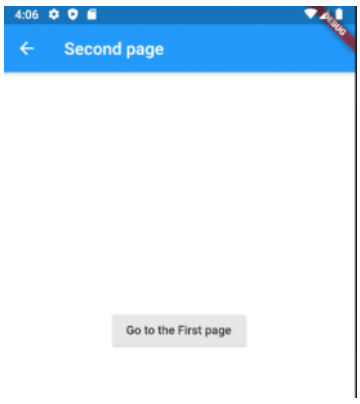
- MaterialPageRoute함수의 인자 값인 builder의 역할은 "어떤 위젯이 MaterialPageRoute의 도움을 받아 생성되어야 할지를 정의하는 역할을 함. 위에서는 Second page를 의미.
- 왜 push 함수를 호출할 때 마다 MaterialPageRoute를 사용하여 인자 값인 builder를 통해서 context를 Flutter에 의해서 할당 받아야 하는가?

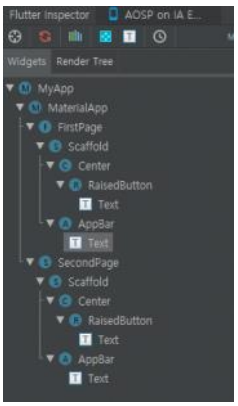
  => StatelessWidget을 생성할 때마다 매번 build 함수는 flutter가 자동으로 호출해주었고, BuildContext 또한 flutter가 자동으로 할당해 주었다.

  Route는 push 메소드를 통해서 다양한 곳에서 호출되며, 그때마다 생성되고, 필요에 따라 재 생성되는 과정을 반복하게 된다.

  물론 child 속성을 사용해서 child widget 형식으로 Route를 추가할 수 있지만 언제 Route 호출이 일어나는가에 따라서 Route가 build하는 과정 중에 다른 Context를 사용할 수 있게 될 수도 있고, 자칫 잘못된 context를 전달하여 Route를 호출해 에러를 발생시킬 수도 있기 때문이다.

  Builder를 사용하면 이런 에러를 미연에 방지할 수 있고, Builder에서 제공한 context를 사용하는 것은 개발자의 선택에 맡기기 때문에(사용을 해도 되고 안 해도 상관없다는 의미) MaterialApp에서 Builder의 사용은 잘못된 Context를 참조해서 원치 않는 Route로의 이동을 방지하는 안전 장치와도 같은 역할을 한다.






```dart
// 두 번째 페이지 작성 (버튼 눌르면 이동하게끔 설계)
class SecondPage extends StatelessWidget {
  @override
  Widget build(BuildContext ctx) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Second page'),
      ), // AppBar
      body: Center(
        child: RaisedButton(
          child: Text('Go to the First page'),
          // Second 페이지에서 FirstPage로 이동.
          onPressed: () {
            Navigator.pop(ctx);
          }), // RaisedButton
      ), // Center
    ); // Scaffold
  }
}
```

```dart
Navigator.push(context2,
    MaterialPageRoute(builder: (BuildContext context) {
  return SecondPage();
```

- First page에서 버튼을 눌러 Second page로 이동한 상태이다. 오른쪽 widget-tree를 보면 버튼을 누르기전 위치했던 First page 위젯이 아래로 내려갔고, 원래 First page 위젯이 있던 자리를 Second page 위젯이 위치한 상태이다. 즉, Stack 자료구조 형태의 widget-tree를 구성한 모습이다.
- Second page는 이미 자체 context인 ctx를 가지고 있고 pop 메소드는 ctx를 전달받아 실행되기 때문에 First page에서 push 메소드에서 사용한 인자 값 MaterialPageRoute를 통해 builder 위젯을 생성했기 때문에 BuildContext를 사용할 필요가 없는 것임.
- 그리고 Scaffold를 사용해서 appBar를 생성하면 뒤로 가기 화살표 버튼이 flutter에서 자동으로 생성해준다.

```dart
const MaterialApp({
  Key key,
  this.navigatorKey,
  this.home,
  this.routes = const <String, WidgetBuilder>{},
  this.initialRoute,
```

- MaterialApp의 아규먼트 중 에서 this.routes 와 this.initialRoute 는 멀티페이지 이동기능을 구현할 때, 반드시 필요하다.
- initialRoute는 멀티페이지 이동 시, 화면에 제일 처음 출력할 Route를 불러오는 역할을 한다. 즉, home argument와 똑같은 기능을 수행한다. 그래서 멀티 페이지 이동 기능 구현 시, home 대신 initialRoute를 사용한다. 주의할 점은 home과 initialRoute가 동시에 존재하면 에러가 발생한다.
- This.routes는 이동할 페이지들의 이름을 지정하고 생성하는 역할을 수행한다. 또한, Map 자료구조 형태로 구성 되어있다.
- Map 자료구조는 key 값을 사용하면 결과로 Value값의 결과를 return 한다.

```dart
void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      initialRoute: '/',
      routes: {
        '/' : (context) => ScreenA(),
      },
    ); // MaterialApp
  }
}
```

- Navigator.pushNamed 메소드는 각 페이지 별로
  붙여진 이름을 통해서 위젯을 build한 후, push한다.
- 옆 이미지의 코드에서 Navigator.pushNamed 메소드의 인자 값인
  Context는 ScreenA 위젯의 context이고, 2번째 인자 값은 이동할 Route의 Key값이다.

Collection 과 Generic

- **Collection** : 데이터들을 모아서 가지고 있는 자료구조
- **Generic** : Collection이 가지고 있는 데이터들의 **데이터 type을 지정**.
- Flutter에서 **가장 흔히 볼 수 있는 Collection은 List**이다.
- List에서는 fixed-length list 와 growable list가 있는데 **fixed-length list는 list 내의 데이터 개수가 지정한 개수만큼만 올 수 있는 것**.
  growable list는 데이터 개수가 제한이 없음.
- C++의 Templete과 유사한 개념.

```
void main()
{
  var number = new List(5);
}
```

- 아래와 같은 경우, **List 클래스에 생성자를 통해서 새로운 인스턴스를 생성한 것**이며, 값으로 5를 넣어줬는데 **이 경우는 fixed-length list**이다.
  => **즉, 데이터 개수가 5개로 제한**한 것.
- **Growable list**는 **List( )에 인자 값 없이 비워 놓으면 된다**.
- 위 코드에서는 자료형이 var인데 List 객체의 자료형에 맞게 정해진다.

```
void main()
{
  List<String> names = List();
  names.addAll(['James', 'John', 'Tom']);
  print(names);
}
```

```
[James, John, Tom]
```

- 여기서 **<String> 이 부분이 Generic**이라고 한다.
- **Column 위젯의 속성**에도 쓰이며, **route 속성으로도 쓰인다**.

```
List<Widget> children = const <Widget>[],
```

- **List안에 들어오는 데이터들이 반드시 widget이어야 한다는 의미**.
- 즉, < >안에 들어간 자료형에 해당되는 데이터만이 List에 저장될 수 있다는 의미이다.

```
final Map<String, WidgetBuilder> routes;
```

- **Map이라는 Collection 자료를 Generic 기법으로 구현한 것**임.

```
class Circle {}
class Square {}

class SquareSlot{
  insert (Square sqareSlot){
  }
}

class CircleSlot{
  insert (Circle circleSlot){
  }
}
void main() {
  var circleSlot = new CircleSlot();

  circleSlot.insert(new Circle());

  var squareSlot = new SquareSlot();

  squareSlot.insert(new Circle());
}
```

```
void main() {
  var circleSlot = new Slot<Circle>();
  circleSlot.insert(new Circle());

  var squareSlot = new Slot<Square>();
  squareSlot.insert(new Square());

}

class Circle {}
class Square {}

class Slot<T>{
  insert (T shape){
  }
}
```

- **왼쪽은 제네릭을 사용하지 않은 코드**이고, **오른쪽은 제네릭을 사용한 코드**이다
  한눈에 봐도 코드 량이 훨씬 줄어들었으며 간결하다.
  그리고 나중에 배울 **Stateful 위젯은 이 Generic을 사용한 위젯**이므로, 반드시 제네릭을 알아야 한다.

```
void main()
{
  String name = 'Sean';
  print(name);

  print("Hi, $name, what's up?");
}
```

▶ RUN

```
Console

Sean
Hi, Sean, what's up?
```

- **$연산자를 사용**하여 **출력할 변수명을 지정해주면 변수에 저장된 값이 출력**됨.

```
$a 님 반갑습니다 => 홍길동님 반갑습니다
$b 님 반갑습니다 => 코딩셰프님 반갑습니다
Interpolation : 보간법
당신의 점수는 $score 이며 당신의 레벨은 $level 입니다
```

- **String Interpolation**이란, **문자열 내에서 Interpolation을 사용**하여 **문자열 중간에 변수를 끼워 넣는 것**.