



PEMROGRAMAN MOBILE DENGAN **FLUTTER**

Bayu Angga Wijaya, M.Kom
Juliansyah Putra Tanjung, S.T., M.Kom
N. Priya Dharshinni, M.Kom
Insidini Fawwaz, M.Kom
Batara Surya Perdana Girsang, S.Kom

PEMOGRAMAN MOBILE DENGAN FLUTTER

Penulis

Bayu Angga Wijaya, M.Kom
Juliansyah Putra, S.T., M.Kom
N. Priya Dharshinni, M.Kom
Batara Surya Perdana Girsang, S.Kom
Insidini Fawwaz

Editor

Batara Surya Perdana Girsang, S.Kom

Penerbit

UNPRI Press

ISBN :

978-623-8299-11-9

Redaksi

Jl. Sampul, Medan

Hak Cipta dilindungi Undang-Undang
Dilarang memperbanyak karya tulis ini dalam
bentuk dan dengan cara apapun tanpa ijin dari penerbit

KATA PENGANTAR

Puji dan syukur atas Kehadiran Tuhan Yang Maha Esa atas berkat dan rahmat-Nya sehingga dapat menyelesaikan buku ajar “Pemrograman Mobile dengan Flutter”. Buku ini menyajikan pengetahuan dan wawasan yang kami miliki dalam pembuatan program berbasis mobile dengan framework Flutter.

Buku ini terdiri atas penjelasan mengenai dasar pemrograman mobile, android, flutter, teknik pemrograman, bagaimana cara menjalankan program, contoh program serta latihan. Selain itu, buku ini juga memberikan landasan kuat dalam pengembangan aplikasi mobile, yang melibatkan banyak aspek, termasuk pemrograman, desain antarmuka pengguna, dan pengelolaan proyek.

Kami ingin berterima kasih kepada semua yang terlibat dalam pembuatan buku ini, mulai dari para penulis hingga tim penyunting, serta semua pihak yang memberikan dukungan dan inspirasi.

Selamat membaca, belajar, dan menjelajahi dunia pemrograman mobile. Semoga buku ini dapat memberikan manfaat bagi semua pembaca.

Medan, November 2023

Penulis

DAFTAR ISI

KATA PENGANTAR	i
DAFTAR ISI.....	ii
BAB I PEMROGRAMAN MOBILE	1
1. Pengenalan Pemrograman Mobile	1
1.1. Aplikasi Native.....	1
1.2. Aplikasi Hybrid.....	2
BAB II FLUTTER.....	4
2. Defenisi Flutter	4
2.1 Fitur unggulan flutter	4
BAB III BAHASA PEMROGRAMAN DART	7
3. Defenisi Bahasa Pemrograman Dart	7
3.1 Tipe Data.....	7
3.2 Operator	9
3.3 Function	10
3.4 Control Flow / Percabangan	11
3.5 Iterasi / Looping.....	13
BAB IV PEMROGRAMAN BERORIENTASI OBJEK (BASIC).....	15
4. Sekilas Tentang Pemrograman Berorientasi Objek	15
4.1 Kelas / class	15
4.2 Objek / object	16
4.3 Constructor.....	16
BAB V ANDROID STUDIO.....	18
5. Defenisi Android Studio	18
5.1 Android SDK Tools.....	18
5.2 Android versions and API levels	19
5.3 Memulai Project Baru	20
BAB VI RUNNING PROJECT	24
6.1 Android Studio.....	24
6.2 Visual Studio Code.....	25
BAB VII FLUTTER WIDGET	30
7.1 Sekilas Tentang Flutter Widget.....	30
7.2 The Widget Tree.....	32
7.3 Jenis Widget.....	33
7.4 Widget Lifecycle Event	38

7.5	Composing (Use Case)	39
7.6	Row – Horizontal View	42
7.8	Column – Vertical View	45
7.8	Penyesuaian - Customization	48
7.9	Composing and Customization View	49
7.10	Implementasi ListView	57
7.11	Navigation dan Routing	70
7.12	Input View	76
7.13	Basic State Management	96
BAB VIII PENUTUP		112
DAFTAR PUSTAKA		113

BAB I PEMROGRAMAN MOBILE

1. Pengenalan Pemrograman Mobile

Secara umum, membuat aplikasi seluler adalah tugas yang sangat kompleks dan menantang. Ada banyak framework yang tersedia, yang menyediakan fitur unggulan untuk mengembangkan aplikasi seluler. Namun sebelum melakukan pengembangan, perlu dilakukan analisis kebutuhan untuk menentukan target aplikasi yang ingin dibangun, adapun target aplikasi mobile terdiri dari:

1.1. Aplikasi Native

Aplikasi native adalah aplikasi mobile khusus yang hanya dapat digunakan di perangkat tertentu, baik itu Android ataupun iOS. Karena aplikasi native bekerja di sistem perangkat yang berbeda, bahasa pemrograman yang membentuknya tentu juga berbeda.

1. Android

Pembangunan aplikasi native untuk android menggunakan bahasa pemrograman java atau kotlin sebagai bahasa pemrograman dan Android Studio sebagai IDE atau code editor. Pengembangan aplikasi android sendiri bisa dilakukan menggunakan komputer windows, linux, ataupun Mac OS.

2. iOS

Pengembangan aplikasi native iOS menggunakan bahasa pemrograman objective-C atau swift dengan xcode sebagai IDE atau code editor. Hanya saja pengembangan aplikasi iOS hanya bisa dilakukan menggunakan komputer Mac OS, karena xcode dari hanya tersedia pada sistem operasi tersebut.

Dalam proses pengembangan maupun selama proses publikasi aplikasi native memiliki kelebihan dan kekurangan yaitu:

- Kelebihan
 1. Didukung sepenuhnya oleh penyedia perangkat, contoh native android akan didukung sepenuhnya oleh google dan native iOS akan sepenuhnya didukung oleh apple.
 2. Kemudahan dalam mengakses hardware ataupun memodifikasi kinerja dari hardware tersebut, seperti kamera, bluetooth, gps dll.
 3. Memiliki performa yang sangat baik.

- Kelemahan:
 1. Tidak flexible senggga tidak aplikasi native tidak akan bisa berjalan pada platform lain.
 2. Perbedaan teknologi membuat proses pengembangan menjadi lambat dan bisa jadi memakan biaya.

3. Update secara berkala, hal ini terjadi ketika platform seperti android dan iOS merilis versi barunya, dalam aplikasi native beberapa fitur ada kemungkinan berubah, dimodifikasi atau bahkan bisa tidak tersedia lagi, sehingga kita harus memperbaiki aplikasi kita dengan aturan terbaru.

1.2. Aplikasi Hybrid

Aplikasi hybrid adalah aplikasi yang bisa digunakan di berbagai platform. Baik itu Android, iOS, maupun Windows, hanya dengan menggunakan satu framework atau bahasa program. Beberapa tools terkenal yang menawarkan fitur pengembangan aplikasi hybrid adalah:

1. Flutter

Flutter adalah framework multi platform yang dikembangkan oleh google, untuk saat ini platform yang didukung oleh flutter adalah Android, iOS, Linux, Mac OS, Windows dan web. Flutter sendiri menggunakan bahasa program dart dalam proses pengembangannya.

2. React Native dan ionic

React Native adalah framework multi platform menggunakan bahasa javascript / typescript, React Native sendiri dikembangkan oleh facebook dan saat ini hanya mendukung Android dan iOS.

3. Xamarin

Xamarin adalah framework multiplatform yang dikembangkan microsoft menggunakan bahasa pemrograman C#.

Dalam proses pengembangan maupun selama proses publikasi aplikasi native memiliki kelebihan dan kekurangan yaitu:

- Kelebihan:

1. Pengembang hanya membutuhkan 1 code base, atau skill bahasa pemrograman.
2. Proses pengembangan lebih cepat karena kebanyakan framework sudah menyediakan fitur-fitur yang akan mempermudah developer dalam mengembangkan aplikasi. Contoh: tidak adanya xml untuk melakukan desain tampilan aplikasi pada Android.

- Kekurangan:

1. Kesulitan dalam penggunaan fitur native, implementasi fitur native pada aplikasi hybrid tetap membutuhkan skill bahasa native seperti java / kotlin untuk android, swift untuk iOS dan mac, C untuk windows dan linux.
2. Akses terhadap hardware terbatas, contoh penggunaan kamera pada aplikasi hybrid tetap bisa dilakukan, tapi untuk melakukan modifikasi pada fitur kamera tersebut akan sulit dikembangkan (contoh: fitur filter pada aplikasi tiktok dan instagram).

Dengan adanya kebutuhan native dan hybrid dalam pengembangan aplikasi mobile, perlu dilakukan analisis kebutuhan sebelum memutuskan teknologi atau framework apa yang akan digunakan dalam mengembangkan aplikasi mobile:

Kapan harus menggunakan native?

1. Ketika aplikasi memerlukan fitur yang cukup rinci dan berkaitan langsung dengan hardware, contoh: Camera untuk membuat fitur filter pada instagram atau tiktok, Lokasi secara realtime pada aplikasi gojek, fitur upload dengan jadwal pada google drive dilengkapi dengan analisis kualitas jaringan (upload akan pause jika jaringan mobile atau sedang tidak baik).
2. Ketika aplikasi memerlukan performa yang baik sekalipun dengan menangani banyak data, contoh: aplikasi tokopedia, yang menampilkan banyak produk dan promosi dll.
3. Ketika aplikasi membutuhkan fitur yang berkaitan dengan sistem operasi, contoh: fitur upload dengan jadwal dan pengecekan kestabilan jaringan pada google drive.

Contoh aplikasi yang sebaiknya dikembangkan secara native:

- Aplikasi dengan fitur filter pada kamera.
- Aplikasi penjualan yang butuh koneksi bluetooth ke printer.
- Aplikasi yang membutuhkan lacak lokasi secara real time.

Kapan harus menggunakan hybrid?

1. Ketika anda benar-benar tidak membutuhkan akses dan fitur yang sangat kompleks dari hardware.
2. Ketika aplikasi hanya mementingkan tampilan dan transaksi.

Contoh aplikasi yang sebaiknya dikembangkan secara hybrid:

- Aplikasi reservasi hotel, lapangan dll.
- Aplikasi pembelajaran.
- Aplikasi berita.

BAB II FLUTTER

2. Defenisi Flutter

Flutter adalah kit pengembangan perangkat lunak UI sumber terbuka dan gratis yang diperkenalkan oleh Google. Flutter digunakan untuk membangun aplikasi untuk Android, iOS, Windows, dan web. Versi pertama Flutter diumumkan pada tahun 2015 di Dart Developer Summit. Awalnya dikenal dengan nama kode "Sky" dan dapat berjalan di OS Android. Setelah Flutter diumumkan, versi Flutter Alpha pertama (v-0.06) dirilis pada Mei 2017. Kemudian, selama keynote hari Pengembang Google di Shanghai, Google meluncurkan pratinjau Flutter kedua pada September 2018 yang merupakan rilis besar terakhir sebelum versi Flutter 1.0. Pada 4 Desember 2018, versi stabil pertama dari kerangka kerja Flutter dirilis di acara Flutter Live, yang menunjukkan Flutter 1.0. Rilis kerangka kerja stabil saat ini adalah Flutter v1.9.1 + hotfix.6 pada 24 Oktober 2019.

Flutter menggunakan bahasa pemrograman Dart untuk membuat aplikasi. Pemrograman Dart memiliki beberapa fitur yang sama dengan bahasa pemrograman lain, seperti Kotlin dan Swift, dan dapat di-trans-compile menjadi kode JavaScript. Flutter terutama dioptimalkan untuk aplikasi seluler 2D yang dapat berjalan di platform Android dan iOS. Kita juga dapat menggunakannya untuk membangun aplikasi berfitur lengkap, termasuk kamera, penyimpanan, geolokasi, jaringan, SDK pihak ketiga, dan banyak lagi.

Flutter berbeda dari framework lain karena tidak menggunakan WebView maupun widget OEM yang dikirimkan bersama perangkat. Sebaliknya, ia menggunakan mesin render berkinerja tinggi untuk menggambar widget.

Flutter juga mengimplementasikan sebagian besar sistemnya seperti animasi, gerakan, dan widget dalam bahasa pemrograman Dart yang memungkinkan pengembang membaca, mengubah, mengganti, atau menghapus sesuatu dengan mudah. Flutter memberikan kontrol yang sangat baik kepada pengembang atas sistem.

2.1 Fitur unggulan flutter

Berikut merupakan beberapa fitur yang menjadi keunggulan flutter :

1. Open-Source : Flutter adalah framework dengan sumber terbuka dan gratis untuk mengembangkan aplikasi seluler.
2. Cross-platform : Fitur ini memungkinkan Flutter untuk menulis kode satu kali, memelihara (maintain), dan dapat berjalan di platform yang berbeda. Hal ini menghemat waktu, tenaga, dan uang para pengembang.
3. Hot Reload : Setiap kali pengembang membuat perubahan pada kode, maka perubahan ini dapat dilihat secara instan dengan Hot Reload. Artinya, perubahan langsung terlihat di aplikasi itu sendiri. Ini adalah fitur yang sangat

berguna, yang memungkinkan pengembang untuk memperbaiki bug secara instan.

4. Accessible Native Features and SDKs : Fitur ini memungkinkan proses pengembangan aplikasi dengan mudah dan menyenangkan melalui kode asli Flutter, integrasi pihak ketiga, dan API platform. Dengan demikian, kami dapat dengan mudah mengakses SDK di kedua platform.
5. Minimal Code : Aplikasi Flutter dikembangkan oleh bahasa pemrograman Dart, yang menggunakan kompilasi JIT dan AOT untuk meningkatkan waktu start-up secara keseluruhan, berfungsi, dan mempercepat kinerja. JIT meningkatkan sistem pengembangan dan menyegarkan UI tanpa perlu bersusah payah membangun yang baru.
6. Widget : Kerangka Flutter menawarkan widget, yang mampu mengembangkan desain khusus yang dapat disesuaikan. Yang terpenting, Flutter memiliki dua set widget: Desain Material dan widget Cupertino yang membantu memberikan pengalaman bebas gangguan di semua platform.

Flutter memiliki beberapa **kelebihan**, sebagai berikut :

1. Flutter membuat proses pengembangan aplikasi sangat cepat karena fitur hot-reload. Fitur ini memungkinkan kita untuk mengubah atau memperbarui kode yang direfleksikan segera setelah perubahan dilakukan.
2. Flutter memberikan pengalaman scrolling (gulir) yang lebih lancar dan mulus dalam menggunakan aplikasi tanpa banyak hang atau terputus, yang membuat aplikasi yang berjalan lebih cepat dibandingkan dengan framework aplikasi seluler lainnya.
3. Flutter membuat proses pengembangan aplikasi sangat cepat karena fitur hot-reload. Fitur ini memungkinkan kita untuk mengubah atau memperbarui kode yang direfleksikan segera setelah perubahan dilakukan.
4. Flutter memberikan pengalaman scrolling (gulir) yang lebih lancar dan mulus dalam menggunakan aplikasi tanpa banyak hang atau terputus, yang membuat aplikasi yang berjalan lebih cepat dibandingkan dengan framework aplikasi seluler lainnya.
5. Flutter mengurangi waktu dan upaya pengujian. Seperti yang kita ketahui, aplikasi flutter bersifat lintas platform sehingga penguji tidak selalu perlu menjalankan serangkaian pengujian yang sama pada platform berbeda untuk aplikasi yang sama.
6. Flutter memiliki antarmuka pengguna yang sangat baik karena menggunakan widget yang berpusat pada desain, alat pengembangan tinggi, API tingkat lanjut, dan banyak lagi fitur lainnya. Flutter mirip dengan kerangka kerja reaktif di mana pengembang tidak perlu memperbarui konten UI secara manual. Sangat cocok untuk aplikasi MVP (Minimum Viable Product) karena proses pengembangannya yang cepat dan sifatnya yang lintas platform.

Namun Flutter juga memiliki beberapa **kekurangan**, sebagai berikut :

1. Flutter adalah bahasa yang relatif baru yang membutuhkan dukungan integrasi berkelanjutan melalui pemeliharaan skrip.
2. Flutter memberikan akses yang sangat terbatas ke pustaka SDK. Artinya, developer tidak memiliki banyak fungsi untuk membuat aplikasi seluler. Jenis fungsi seperti itu perlu dikembangkan sendiri oleh pengembang Flutter.
3. Aplikasi Flutter tidak mendukung browser. Ini hanya mendukung platform Android dan iOS.
4. Flutter menggunakan pemrograman Dart untuk pengkodean, jadi pengembang perlu mempelajari teknologi baru. Namun, untuk pengembang itu mudah dipelajari.

BAB III

BAHASA PEMROGRAMAN DART

3. Defenisi Bahasa Pemrograman Dart

Dart adalah bahasa pemrograman open source, dan berorientasi objek dengan sintaks gaya C yang dikembangkan oleh Google pada tahun 2011. Tujuan pemrograman Dart adalah untuk membuat antarmuka pengguna frontend untuk web dan aplikasi seluler. Dart sedang dalam pengembangan aktif, dikompilasi ke kode mesin asli untuk membangun aplikasi seluler, terinspirasi oleh bahasa pemrograman lain seperti Java, JavaScript, C#. Karena Dart adalah bahasa yang dikompilasi sehingga Anda tidak dapat mengeksekusi kode Anda secara langsung; sebaliknya, kompilator mem-parsingnya dan mentransfernya ke dalam kode mesin.

Dart mendukung sebagian besar konsep umum bahasa pemrograman seperti class, interface, function, tidak seperti bahasa pemrograman lainnya. Bahasa Dart tidak mendukung array secara langsung. Dart mendukung collection, yang digunakan untuk mereplikasi struktur data seperti array, generik, dan pengetikan opsional.

3.1 Tipe Data

Dart adalah bahasa pemrograman yang diketik secara statis; ini berarti bahwa variabel selalu memiliki tipe yang sama, yang tidak dapat diubah.

Tipe data dalam pemrograman Dart diberikan di bawah ini:

1. Numbers

Di Dart, Number digunakan untuk mewakili literal numerik. Tipe data Number dikelompokkan menjadi dua jenis:

integer: mewakili bilangan bukan pecahan (bilangan bulat). Bilangan bulat dapat dideklarasikan dengan kata kunci `int`.

```
void main() {  
  int num = 1;  
  print(num);  
  // program akan mencetak 1  
}
```

double: mewakili bilangan pecahan (bilangan floating-point). Ganda dapat dideklarasikan dengan kata kunci `double`.

```
void main() {  
  double num = 1.5;  
  print(num);  
  // program akan mencetak 1.5  
}
```

2. Strings

String mewakili literal string dan merupakan urutan karakter. Sebuah string dideklarasikan dengan kata kunci String.

```
void main() {
    String str = 'Hello Dart';
    print(str);
    // program mencetak Hello Dart
}
```

3. Booleans

Boolean mewakili nilai benar dan salah. Itu dideklarasikan dengan kata kunci bool.

```
void main() {
    bool val = true;
    print(val);
    // program mencetak true
}
```

4. Lists

List digunakan untuk mewakili kumpulan objek. Ini mirip dengan konsep array dalam bahasa pemrograman lain.

```
void main() {
    List<String> greets = ['Hello, ', 'Welcome', 'to', 'Dart'];
    print(greets);
    // program akan mencetak [Hello, , Welcome, to, Dart]

    print(greets[1]);
    // program akan mencetak Welcome
}
```

5. Map

Map adalah kumpulan dinamis yang mewakili sekumpulan nilai sebagai pasangan nilai kunci. Kunci dan nilai pada Map dapat berupa jenis apa pun.

```
void main()
{
    Map shot = {
        'greet': 'Hello, Selamat datang di Dart',
        'meet': 'Wow, Apa kabar?',
        'leave': 'Sampai Jumpa'
    }
```

```

};
//print(shot);
// program akan mencetak {greet: Hello, Selamat datang di Dart, meet:
Wow, Apa kabar?, leave: Sampai Jumpa}

print(shot['greet']);
// program akan mencetak Hello, Selamat datang di Dart

print(shot['leave']);
// program akan mencetak Sampai Jumpa

print(shot['gone']);
// program akan mencetak null
}

```

3.2 Operator

Dart memiliki banyak operator, namun pada implementasinya hanya beberapa operator yang sering digunakan yaitu:

1. Operator aritmatika

Symbol	Name	Description
+	Penjumlahan	Gunakan untuk menambahkan dua operan
-	Pengurangan	Gunakan untuk mengurangi dua operan
-expr	Minus	Ini Digunakan untuk membalikkan tanda ekspresi
*	Perkalian	Gunakan untuk mengalikan dua operan
/	Pembagian	Gunakan untuk membagi dua operan
~/	Pembagian	Gunakan dua operan bagi dua tetapi berikan output dalam bilangan bulat
%	Modulus	Gunakan untuk memberikan sisa dua operan

2. Operator Relasional

Symbol	Name	Description
>	Lebih besar	Periksa operan mana yang lebih besar dan berikan hasil sebagai ekspresi boolean.

<	Lebih kecil	Periksa operan mana yang lebih kecil dan berikan hasil sebagai ekspresi boolean.
>=	Lebih besar atau sama dengan	Periksa operan mana yang lebih besar atau sama satu sama lain dan berikan hasil sebagai ekspresi boolean.
<=	Lebih kecil atau sama dengan	Periksa operan mana yang lebih kecil atau sama dengan satu sama lain dan berikan hasil sebagai ekspresi boolean.
==	Sama dengan	Periksa apakah operan sama satu sama lain atau tidak dan berikan hasil sebagai ekspresi boolean.
!=	Tidak sama dengan	Not Equal untuk Memeriksa apakah operand tidak sama satu sama lain atau tidak dan memberikan hasil sebagai ekspresi boolean.

3.3 Function

Function adalah jenis prosedur atau rutin. Beberapa bahasa pemrograman membuat perbedaan antara Function (mengembalikan nilai), dan Prosedur, yang melakukan beberapa operasi tetapi tidak mengembalikan nilai.

Secara umum function pada dart dibagi menjadi menjadi 3 bagian dan bisa digunakan gunakan sesuai dengan kebutuhan dan kapan saja, yakni:

1. Fungsi tanpa nilai kembali

Fungsi tanpa kembalian biasanya diawali dengan kata void atau tidak ada awalnya/tipe datanya sama sekali, biasanya fungsi ini dipakai saat kita tidak membutuhkan nilai kembali dari hasil proses yang dilakukan. contoh :

```
void penjumlahan(int a, int b) {
    int total = a + b;
    print(total);
}

void main() {
    penjumlahan(5, 12);
}
```

2. Fungsi dengan nilai kembali

Fungsi dengan nilai kembali adalah fungsi yang dapat diambil nilainya dan dapat langsung dijadikan sebuah nilai. biasanya fungsi jenis ini memiliki tipe data di depannya sebelum nama fungsinya ditulis. contoh:

```

int penjumlahan(int a, int b) {
    int total = a + b;
    return total;
}

void main() {
    int total = penjumlahan(5, 12);
    print(total);
}

```

3. Fungsi 1 baris

fungsi satu baris ini atau yang bisa disebut *single line function*. Fungsi ini hanya bisa diterapkan pada fungsi yang memiliki nilai kembali dan memiliki block function yang pendek, contoh:

```

int penjumlahan(int a, int b) => a + b;

void main() {
    int total = penjumlahan(5, 12);
    print(total);
}

```

3.4 Control Flow / Percabangan

Percabangan adalah suatu pilihan atau opsi dimana terdapat kondisi tertentu yang harus dipenuhi oleh program untuk menjalankan perintah, jika pilihan yang menjadi syarat terpenuhi, maka pilihan dijalankan, jika tidak maka program tidak akan menjalankan perintah atau melewatinya dan melihat kondisi lainnya untuk dijalankan atau berhenti sama sekali.

Percabangan if

Digunakan untuk mencari hasil sesuai dengan kondisi tertentu; percabangan akan melakukan pengecekan pertama ke block if kemudian ke else if sampai kondisi yang dibutuhkan menghasilkan true, jika tidak akan menggunakan block else.

```

void main() {
    var nilai = 65;

    if (nilai >= 80) {
        print("A");
    } else if (nilai >= 70) {
        print("B");
    } else if (nilai >= 60) {
        print("C");
    } else if (nilai >= 50) {

```



```
    print("D");
} else {
    print("E");
}
}
```

Percabangan ternary

Ternary expression digunakan untuk mempersingkat if, namun hanya memiliki 2 kondisi;

```
void main() {
    String password = '123';

    String hasil = password == '123' ? 'benar' : 'salah';

    print(hasil);
}
```

Percabangan switch - case

Digunakan untuk mencari hasil sesuai dengan kriteria yang tertentu; perbedaan dengan if adalah switch akan langsung mencari apakah kondisi yang diminta ada dalam case, jika tidak maka akan langsung dilempar ke block default.

```
void main() {
    String kodeNilai = "A";

    switch(kodeNilai) {
        case "A" : {
            print("80 - 100");
            break;
        }
        case "B": {
            print("70 - 79");
            break;
        }
        case "C": {
            print("60 - 69");
            break;
        }
        case "D": {
            print("50 - 59");
            break;
        }
    }
}
```

```
default: {
    print("0 - 49");
    break;
}
}
```

3.5 Iterasi / Looping

Perulangan adalah sebuah kondisi dimana satu atau beberapa baris kode program dieksekusi secara berulang-ulang. Loop digunakan untuk mengeksekusi blok kode yang sama berulang kali, blok kode yang sama dijalankan berulang-ulang beberapa kali selama kondisi tertentu benar. Secara umum perulangan dibagi menjadi 2 bagian:

Counted loop: perulangan yang jumlah perulangannya tentu dan sudah diketahui jumlah perulangannya. Kata kunci untuk perulangan counted loop **for, for..in**

Uncounted loop: perulangan yang jumlah pengulangannya tidak tentu. Kata kunci untuk perulangan uncounted loop **while, do while**

for loop

Struktur perulangan for bisa digunakan untuk mengulang proses yang sudah diketahui jumlah perulangannya. Contoh kita ingin melakukan perulangan dari 0 sampai 5, dan mencetak pesan tiap kali dilakukan perulangan:

```
void main() {
    for (int i = 0; i <= 5; i++){
        print('perulangan ke -$i');
    }
}
```

for in loop

Struktur perulangan for in digunakan untuk menampilkan isi yang ada dalam array atau list, perulangan for in sebenarnya sama dengan perulangan for yang bedakan cuma kata kunci in hanya saja for in digunakan untuk menampilkan isi dalam array atau list.

```
void main() {
    var apps = ["facebook", "youtube", "instagram", "tiktok"];

    for(var app in apps) {
        print(app);
    }
}
```

```
}
```

while Loop

Sama seperti statement for, cara kerjanya perulangan seperti percabangan, ia akan melakukan perulangan selama kondisinya dengan syarat bernilai benar atau true jadi bedanya adalah diharuskan membuat nilai awal dari variabel kemudian membuat batas akhir berupa kondisi dan operasi increment dan decrement.

```
void main() {  
    var index = 1;  
    while(index < 5) {  
        print("perulangan ke $index");  
        index++;  
    }  
}
```

do while loop

Cara kerja do while hampir mirip dengan while. Perbedaannya, jika do while hanya melakukan satu kali perulangan dulu, kemudian mengecek kondisinya. Sedangkan while kondisi di cek dulu baru kemudian statement perulangan dijalankan. Jadi akibat dari itu do while minimal terdapat 1x perulangan. Sedangkan while dimungkinkan perulangan tidak pernah terjadi, ketika kondisinya bernilai false.

```
void main() {  
    var index = 1;  
    do{  
        print("perulangan ke-$index");  
        index++;  
    } while(index < 5);  
}
```

BAB IV

PEMROGRAMAN BERORIENTASI OBJEK (BASIC)

4. Sekilas Tentang Pemrograman Berorientasi Objek

Pemrograman berorientasi objek (Inggris: object-oriented programming disingkat OOP) merupakan paradigma pemrograman berdasarkan konsep "objek", yang dapat berisi data, dalam bentuk field atau dikenal juga sebagai atribut; serta kode, dalam bentuk fungsi/prosedur atau dikenal juga sebagai method. Semua data dan fungsi di dalam paradigma ini dibungkus dalam kelas-kelas atau objek-objek. Bandingkan dengan logika pemrograman terstruktur. Setiap objek dapat menerima pesan, memproses data, dan mengirim pesan ke objek lainnya.

Model data berorientasi objek dikatakan dapat memberi fleksibilitas yang lebih, kemudahan mengubah program, dan digunakan luas dalam teknik peranti lunak skala besar. Lebih jauh lagi, pendukung OOP mengklaim bahwa OOP lebih mudah dipelajari bagi pemula dibanding dengan pendekatan sebelumnya, dan pendekatan OOP lebih mudah dikembangkan dan dirawat.

Pemrograman berorientasi objek hanya didasari oleh class dan object, walaupun masih ada beberapa istilah dalam OOP seperti Abstraksi, Interface, Polimorfisme dll, namun pada materi kali ini, hanya akan membahas beberapa istilah dasar dalam OOP.

4.1 Kelas / class

Kumpulan atas definisi data dan fungsi-fungsi dalam suatu unit untuk suatu tujuan tertentu. Sebagai contoh 'class of **phone**' adalah suatu unit yang terdiri atas definisi-definisi data (attribute) dan fungsi-fungsi yang menunjuk pada berbagai macam perilaku/turunan dari **phone**.

Deklarasi class memiliki beberapa aturan yaitu sebuah class harus memiliki nama unik dan huruf pertama harus menggunakan huruf kapital (pascal case).

```
void main() {
    final phone=Phone(name: 'NOKIA A71', brand: 'NOKIA');
    phone.call();
}
```

```
class Phone {
    /// name dan brand adalah attributes dari class.
    /// attributes bisa ditambah sesuai dengan kebutuhan.
    String name;
    String brand;
```

```

/// constructor dari class (penjelasan pada part berikut)
Phone(this.name, this.brand);

/// deklarasi function / fungsi yang nantinya digunakan
/// mengolah data attributes.
void call() {
    print('HP saya $name');
}
}

```

4.2 Objek / object

Secara sederhana object adalah representasi atau bentuk nyata dari sebuah class, sebagai contoh: **samsung** adalah representasi atau bentuk nyata dari sebuah **phone**.

4.3 Constructor

Constructor adalah method khusus yang akan dieksekusi pada saat pembuatan objek dari sebuah class, constructor adalah properti yang wajib dimiliki oleh sebuah class. Dart mendukung beberapa constructor, namun yang umum digunakan adalah:

- Traditional Constructor: pembuatan objek yang mengikuti urutan deklarasi variabel yang ada pada class. Constructor ini baik digunakan untuk class dengan attributes yang sedikit.

```

class Phone {
    String name;
    String brand;

    Phone(this.name, this.brand);

    void call() {
        print('HP saya $name');
    }
}

void main() {
    Phone phone = Phone('Samsung S22', 'Samsung');
    phone.call();
}

```

- Named Constructor: pembuatan objek yang mengikuti nama-nama variabel yang ada pada class. Constructor ini baik digunakan untuk class dengan

attributes yang banyak.

```
class Phone {
    String name;
    String brand;

    Phone({required this.name, required this.brand});

    void call() {
        print('Calling $name');
    }
}

void main() {
    Phone phone = Phone(name: 'Samsung S22', brand:
'Samsung');
    phone.call();
}
```

BAB V

ANDROID STUDIO

5. Defenisi Android Studio

Android Studio adalah Integrated Development Environment (IDE) untuk sistem operasi Android, yang dibangun di atas perangkat lunak JetBrains IntelliJ IDEA dan didesain khusus untuk pengembangan Android. IDE ini merupakan pengganti dari Eclipse Android Development Tools (ADT) yang sebelumnya merupakan IDE utama untuk pengembangan aplikasi android. Adapun yang dimaksud dengan IDE untuk android adalah, android studio akan digunakan sebagai tools untuk mendukung pengembangan aplikasi android, mulai dari tahap development hingga ke tahap production (publish), banyak fitur yang bisa digunakan untuk mempercepat pengerjaan aplikasi dan sudah tergabung secara default dengan paket instalasi Android Studio seperti:

- Intellisense, fitur yang memberikan autocompletion pada saat pengetikan code, dengan code suggestion yang cukup akurat.
- XML designer, fitur yang digunakan untuk mendesain tampilan aplikasi yang dibangun, tanpa melakukan coding xml secara manual, hanya melakukan drag and drop, namun pada beberapa kasus kustomisasi akan tetap dibutuhkan coding manual pada code xml.
- Git integration untuk mendukung project kolaborasi.
- Profiler untuk menguji dan menganalisis performa aplikasi kita.
- Dan masih banyak lagi.

Android Studio bisa di-download dari <https://developer.android.com/studio>, silahkan untuk menyesuaikan platform yang anda gunakan, untuk saat ini Android Studio tersedia untuk sistem operasi:

- MacOS,
- Linux,
- Windows,
- Dan chrome OS.

5.1 Android SDK Tools

Android Software Development Kit (SDK) merupakan kit yang bisa digunakan oleh para developer untuk mengembangkan aplikasi berbasis Android. Di dalamnya, terdapat beberapa tools seperti debugger, software libraries, emulator, dokumentasi, sample code dan tutorial. Ada beberapa Android SDK yang akan sering digunakan pada materi pembelajaran ini yaitu:

- Android Emulator, untuk melakukan pengetesan aplikasi yang sedang dibangun.
- SDK Platforms, untuk memastikan bahwa Android API yang kita gunakan sudah benar (Android API akan dijelaskan selanjutnya).

- Android SDK Build-Tools, tools yang akan digunakan untuk membangun file binary dari aplikasi kita, pada kasus ini Android menggunakan binary apk.

5.2 Android versions and API levels

Saat platform Android berkembang dan versi Android baru dirilis, setiap versi Android diberi pengenal bilangan bulat unik, yang disebut API Level. Oleh karena itu, setiap versi Android sesuai dengan satu Android API Level. Karena pengguna memasang aplikasi di Android versi lama dan terbaru, aplikasi Android dunia nyata harus dirancang untuk bekerja dengan beberapa level API Android. Adapun beberapa perubahan dari update API level pada umumnya berkaitan dengan fitur terbaru yang dimiliki android, perubahan kebijakan permission pada perangkat, atau adanya penghapusan fitur (deprecation).

API Level pada umumnya bersifat backward compatibility, yang artinya API terbaru akan tetap mendukung fitur yang dimiliki API Level sebelumnya (kecuali jika ada deprecation).

Sedangkan untuk versi, setiap rilis Android memiliki beberapa nama:

- Versi Android, seperti Android 9.0,
- Nama kode (atau makanan penutup), seperti Pie,
- Level API yang sesuai, seperti API level 28.

Adapun level API yang sudah tersedia saat ini bisa dilihat pada gambar berikut, selengkapnya kunjungi <https://apilevels.com/>.

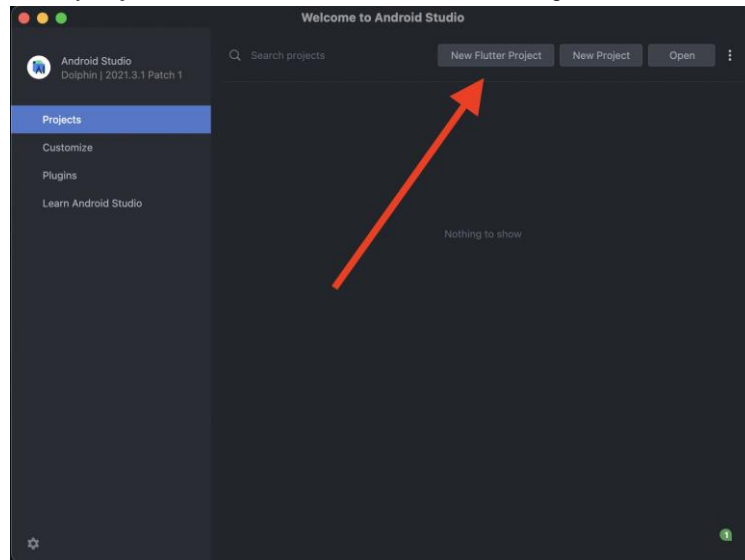
Version	SDK / API level	Version code	Codename	Cumulative usage ¹	Year
Android 13	Level 33	TIRAMISU	Tiramisu ²	No data	2022
Android 12	Level 32 <small>Android 12L</small>	S_V2	Snow Cone ²	20.7%	2021
	Level 31 <small>Android 12</small>	S			
<ul style="list-style-type: none"> ▪ targetSdk must be 31+ for new apps. ▪ targetSdk will need to be 31+ for app updates by Nov 2022 and all existing apps by Nov 2023. ³ 					
Android 11	Level 30	R	Red Velvet Cake ²	50.3%	2020
	<ul style="list-style-type: none"> ▪ targetSdk must be 30+ for app updates, and new WearOS apps. ▪ targetSdk will need to be 30+ for all existing apps by November 2022. ³ 				
Android 10	Level 29	Q	Quince Tart ²	72.1%	2019
Android 9	Level 28	P	Pie	82.9%	2018
	<ul style="list-style-type: none"> ▪ targetSdk must be 28+ for Wear OS app updates. 				
Android 8	Level 27 <small>Android 8.1</small>	O_MR1	Oreo	88.4%	2017
	Level 26 <small>Android 8.0</small>	O			
Android 7	Level 25 <small>Android 7.1</small>	N_MR1	Nougat	92.5%	2016
	Level 24 <small>Android 7.0</small>	N			
Android 6	Level 23	M	Marshmallow	97.4%	2015
Android 5	Level 22 <small>Android 5.1</small>	LOLLIPOP_MR1	Lollipop	98.8%	2015
	Level 21 <small>Android 5.0</small>	LOLLIPOP, L			
<ul style="list-style-type: none"> ▪ Jetpack Compose requires a minSdk of 21 or higher. 				No data	2014

5.3 Memulai Project Baru

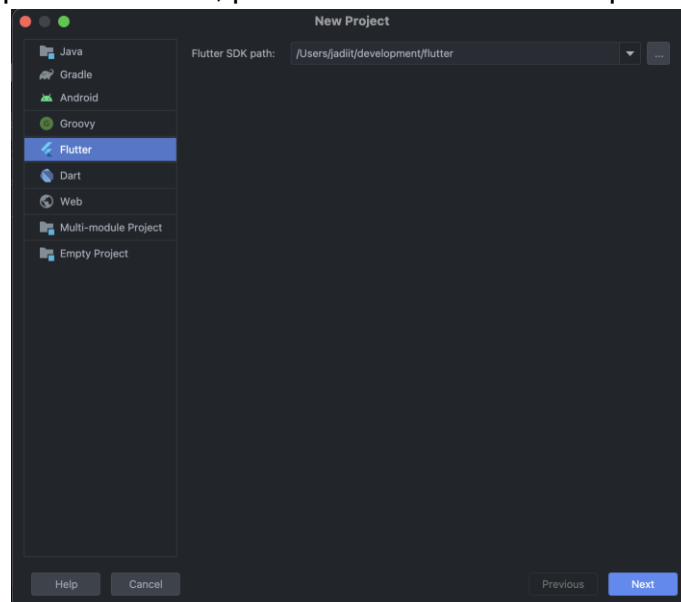
Untuk membuat project dengan android studio, pastikan plugin flutter sudah terinstall pada android studio atau extension flutter pada visual studio code.

Android Studio

1. Buka Android Studio, tampilan awal Android Studio seperti gambar dibawah, untuk pembuatan project baru, klik **New Flutter Project** kemudian klik **next**.

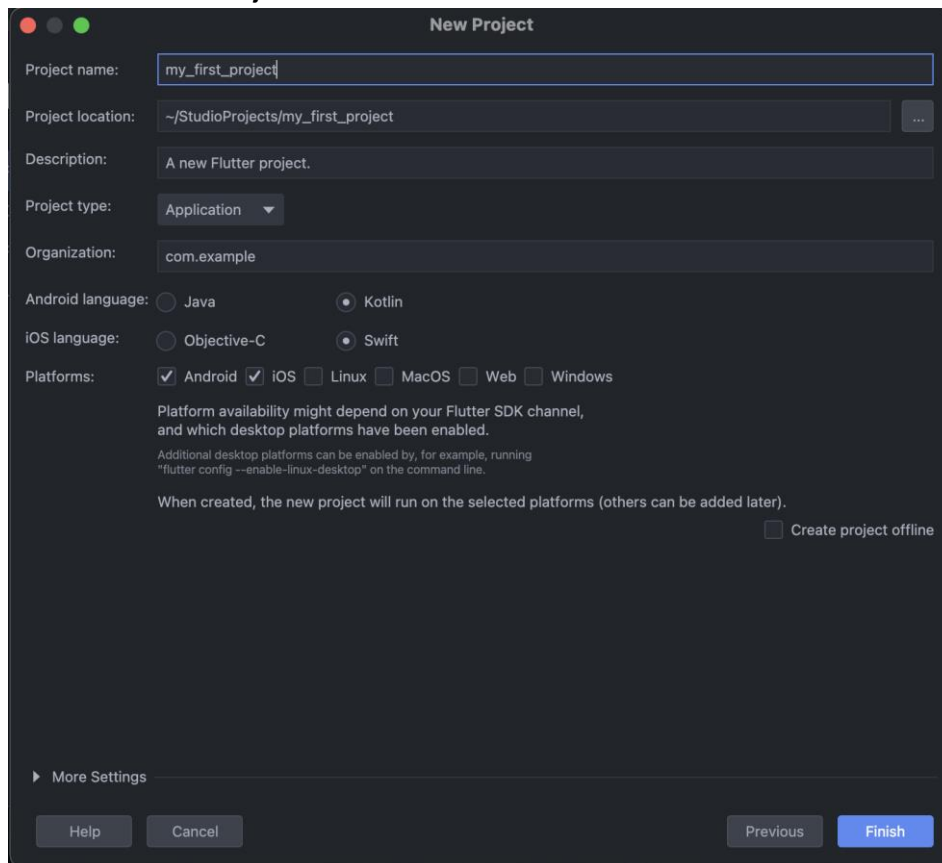


2. Pada halaman pemilihan SDK, pastikan kalau flutter SDK path sudah terisi



3. Pada halaman New Project ada beberapa komponen perlu untuk diperhatikan, yaitu:

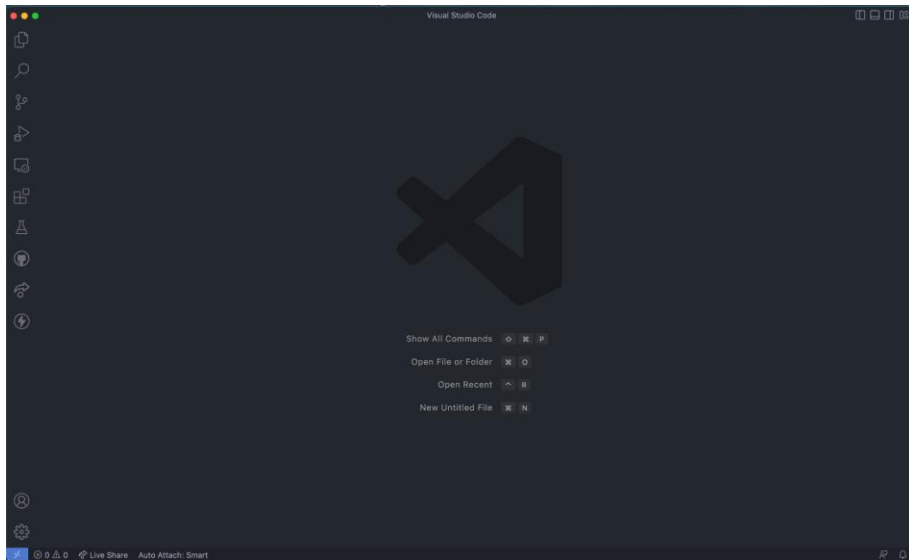
- **Project Name:** nama project yang menggunakan format penulisan tidak boleh menggunakan spasi, dengan pemisah menggunakan symbol ' _ ' (underscore), contoh: my_first_project
- **Project Location:** lokasi penyimpanan project pada komputer.
- **Description:** deskripsi atau detail mengenai project.
- **Project Type:** tipe dari project, karena module ini berfokus pada pengembangan aplikasi, maka pastikan project type yang terpilih adalah Application.
- **Organization:** organisasi dari pengembang (hanya untuk kebutuhan publikasi ke playstore atau appstore)
- **Android Language:** bahasa native untuk android (pastikan untuk memilih kotlin).
- **iOS Language:** bahasa native untuk iOS (pastikan untuk memilih swift).
- **Platform:** target platform dari aplikasi, untuk saat ini pastikan hanya memilih Android dan iOS saja.



Selanjutnya klik **Finish**, dan tunggu android studio selesai memproses project baru tersebut.

Visual Studio Code

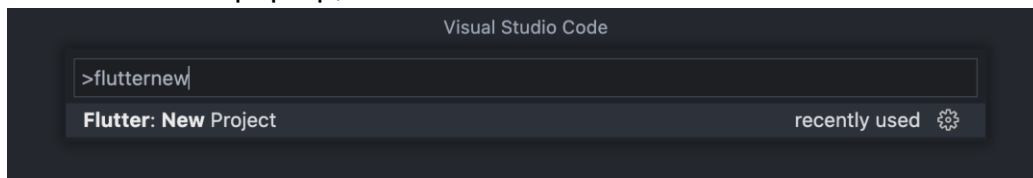
1. Buka visual studio code dan pastikan visual studio code sedang tidak membuka project apapun.



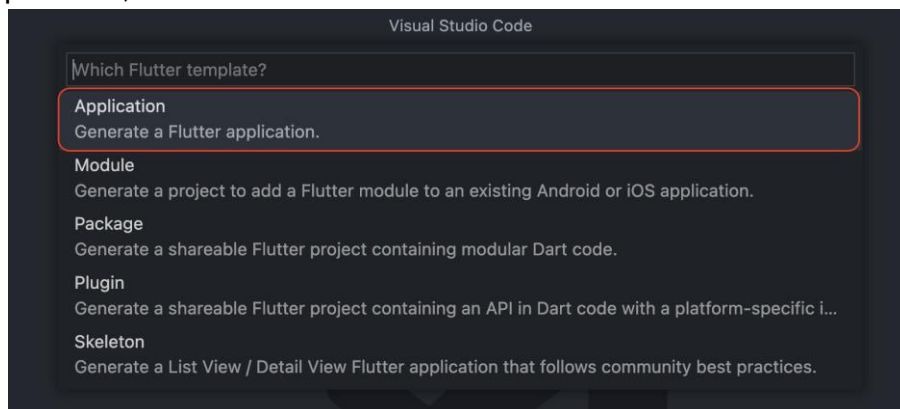
2. Gunakan shortcut untuk membuat project dengan menekan tombol secara bersamaan:

- Windows: **ctrl + shift + p**
- Mac: **command + shift + p**
- Linux: **ctrl + shift + p**

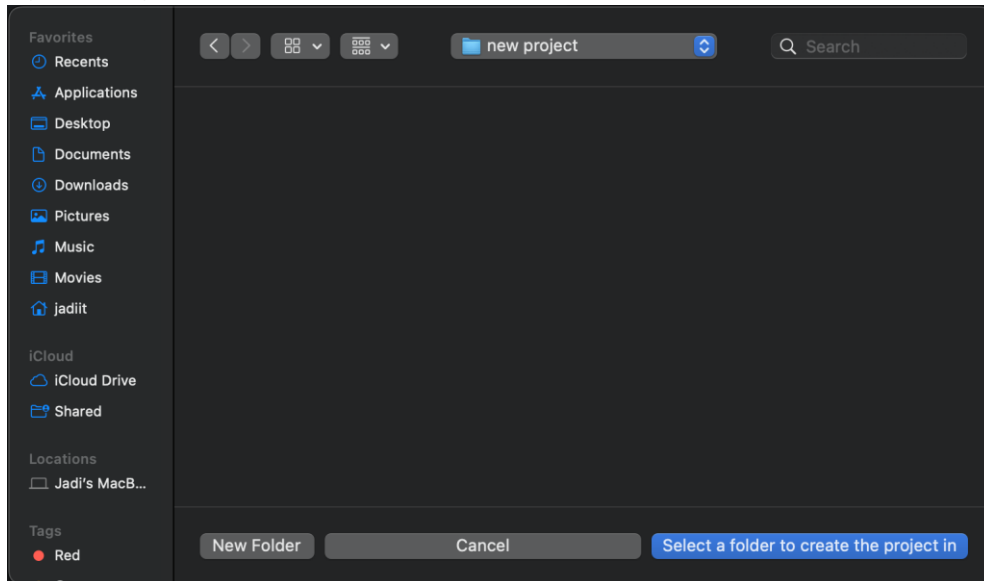
Jika sudah muncul pop-up, kemudian ketik **flutter new** dan tekan tombol enter



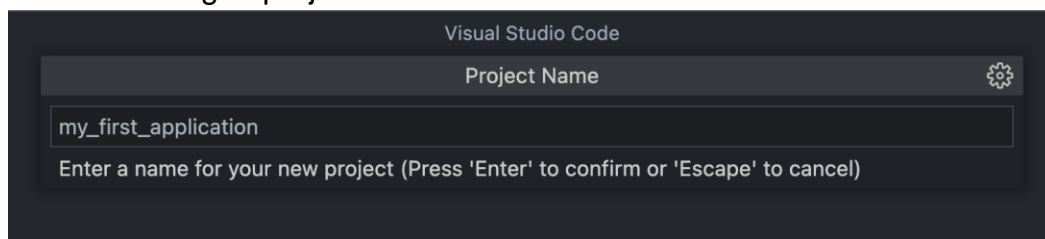
3. Pilih Application, kemudian tekan enter



4. Selanjutnya pilih lokasi penyimpanan project, sesuaikan dengan komputer masing-masing.



5. Kemudian isi nama project yang menggunakan format penulisan tidak boleh menggunakan spasi, dengan pemisah menggunakan symbol '_' (underscore), contoh: my_first_project. Tekan tombol enter dan tunggu visual studio code selesai membangun project.



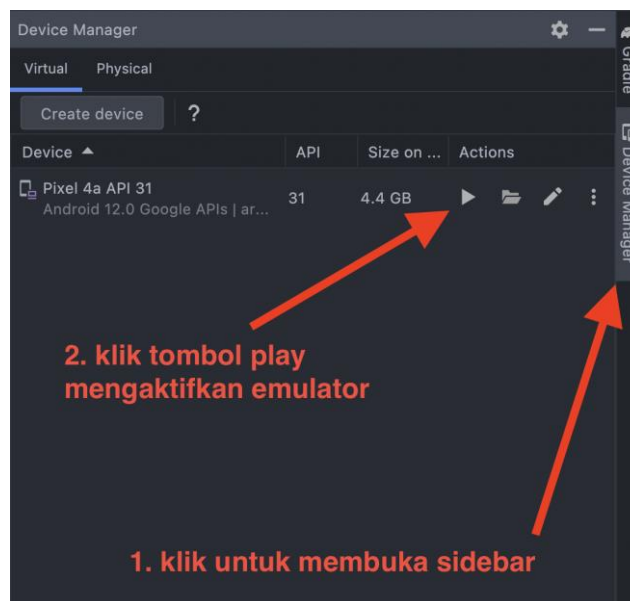
BAB VI RUNNING PROJECT

Running project adalah proses yang digunakan untuk mengeksekusi program yang sudah dikerjakan serta untuk menjalankan aplikasi. Proses ini membutuhkan sistem operasi android sebagai media eksekusinya, dalam kasus ini anda bisa menggunakan emulator dari android studio, atau ponsel fisik android yang anda miliki untuk menjalankan aplikasi, namun untuk menggunakan ponsel sebagai media eksekusi aplikasi android harus mengaktifkan opsi pengembang terlebih dahulu.

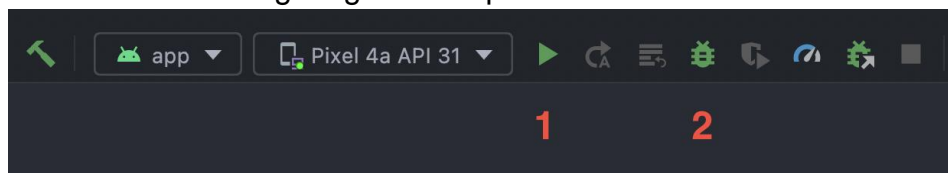
6.1 Android Studio

Adapun cara running project untuk Android studio yaitu :

1. Aktifkan emulator



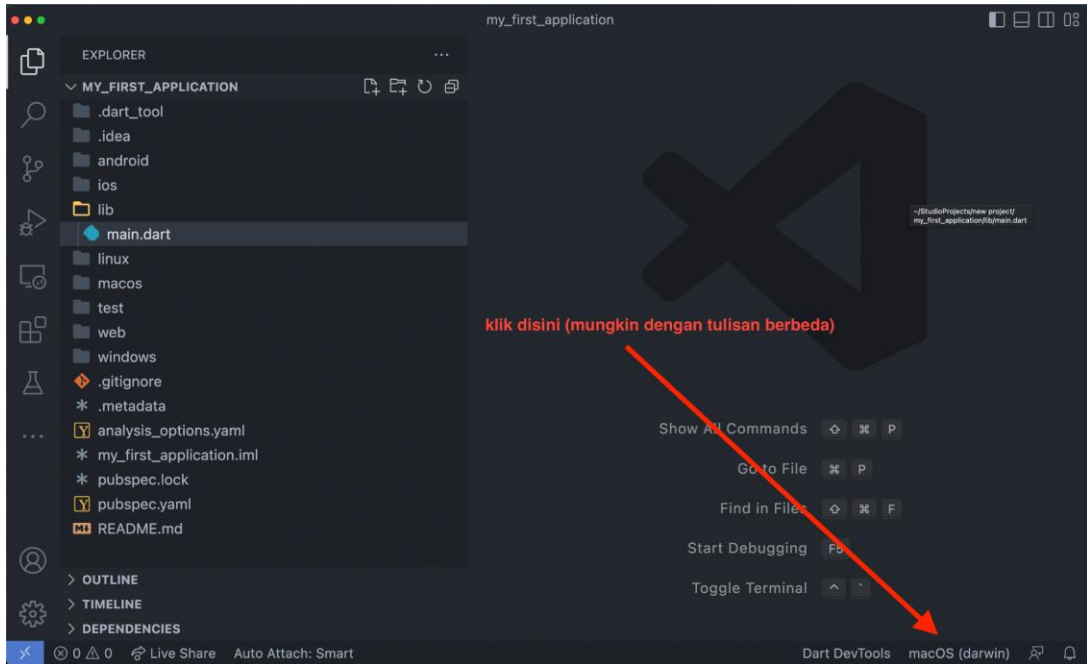
2. Jika emulator sudah berada pada mode standby, tekan **tombol nomor 1** untuk melakukan proses run, Tunggu proses Gradle Sync melakukan compile pada code anda, biasanya proses Gradle Sync untuk run pertama kali akan memakan waktu yang cukup lama. Jika Gradle Sync sudah selesai maka aplikasi anda akan langsung terbuka pada emulator.



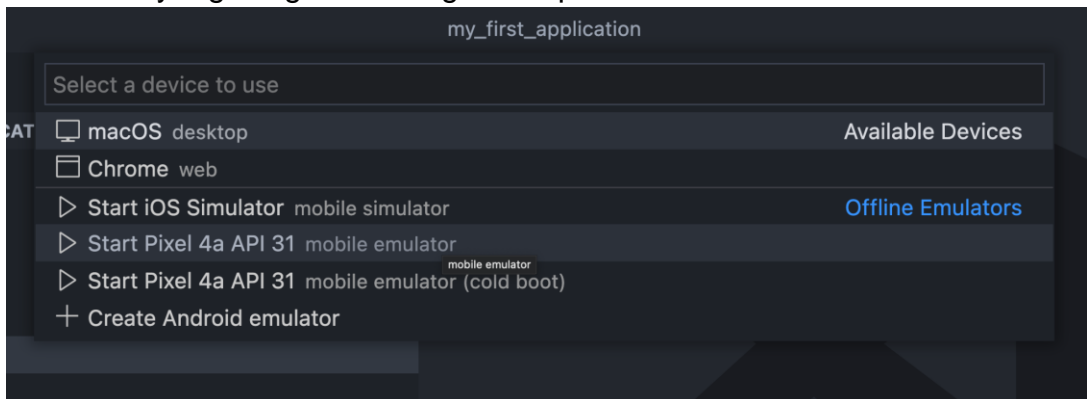
6.2 Visual Studio Code

Sedangkan untuk running project pada Visual Studio Code adalah :

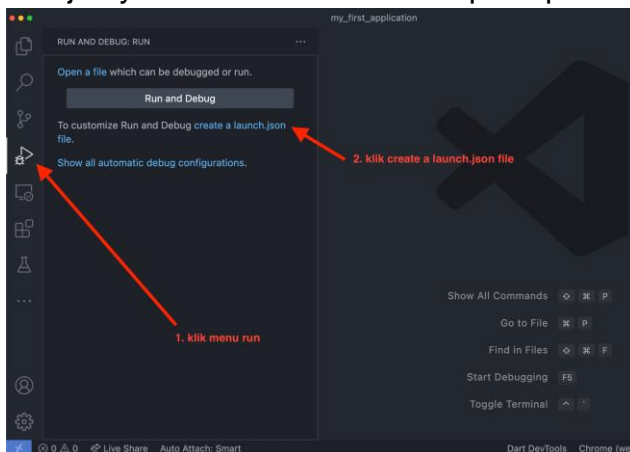
1. Pilih Device yang ingin digunakan dengan cara:



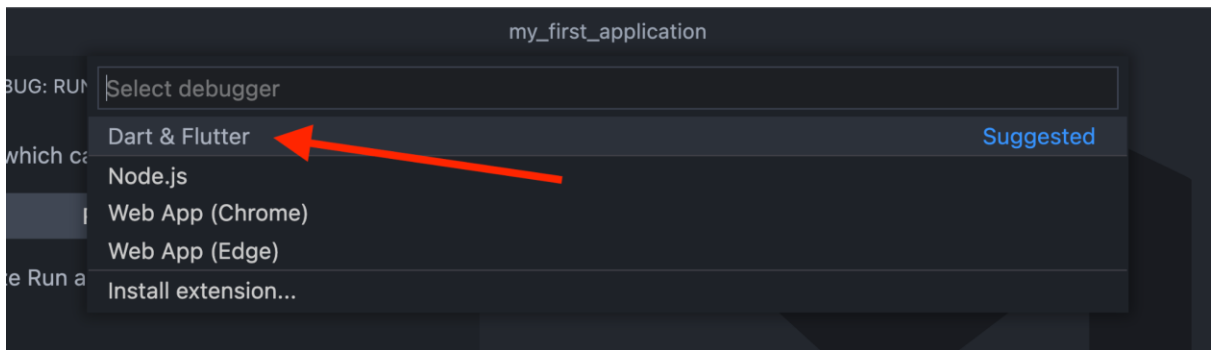
2. Maka akan muncul beberapa device yang bisa digunakan untuk run project. Pilih device yang diinginkan dengan klik pada device tersebut.



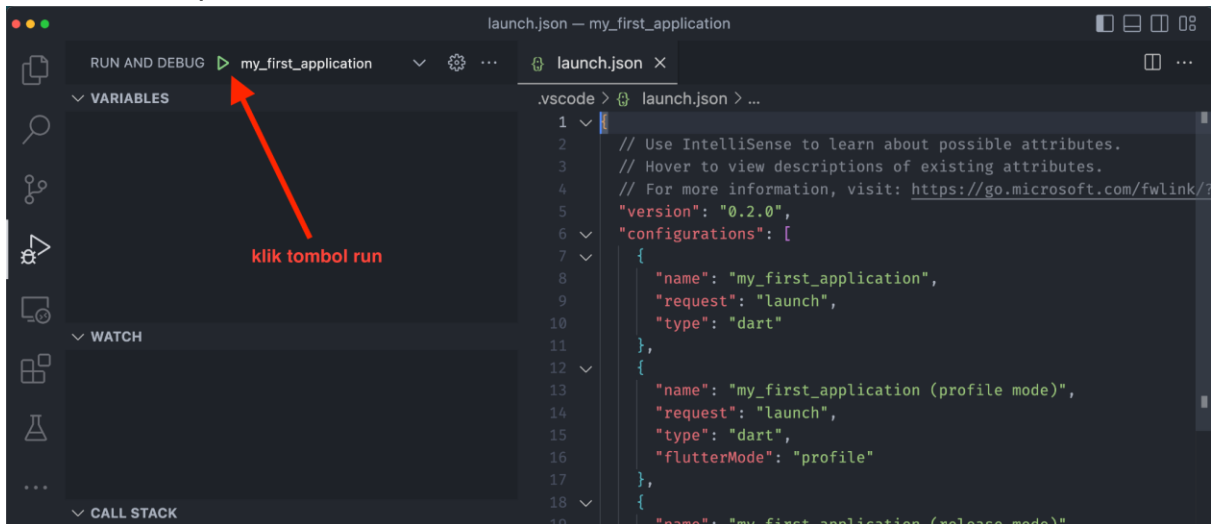
3. Selanjutnya adalah membuat setup script untuk run project.



4. Pastikan untuk memilih Dart & Flutter kemudian tekan enter



5. Maka run script akan otomatis dibuat



6.3. Debugging Project Android Studio

Debugging adalah proses untuk mencari dan menghapus suatu bug dalam sebuah program atau sistem. Yang dimaksud bug adalah suatu error atau cacat yang dapat menyebabkan program atau sistem tidak berjalan dengan sempurna. Biasanya suatu program dibuat dengan penulisan kode yang sangat rumit dan kompleks, sehingga satu kesalahan kecil saja akan berpengaruh besar pada keseluruhan program. Oleh karena itu, debugging sangat penting untuk dilakukan, baik sebelum maupun sesudah perilisan program.

Tujuan utama dari debugging sendiri adalah untuk menghilangkan bug atau masalah yang ada. Debugging ini juga memiliki manfaat lain, berikut ini adalah beberapa di antaranya:

- Mendeteksi error lebih cepat.
- Mempercepat proses perbaikan.
- Mengurangi resiko program disusupi malware.

Berikut ini langkah-langkah yang dapat kamu ikuti untuk melakukan debugging:

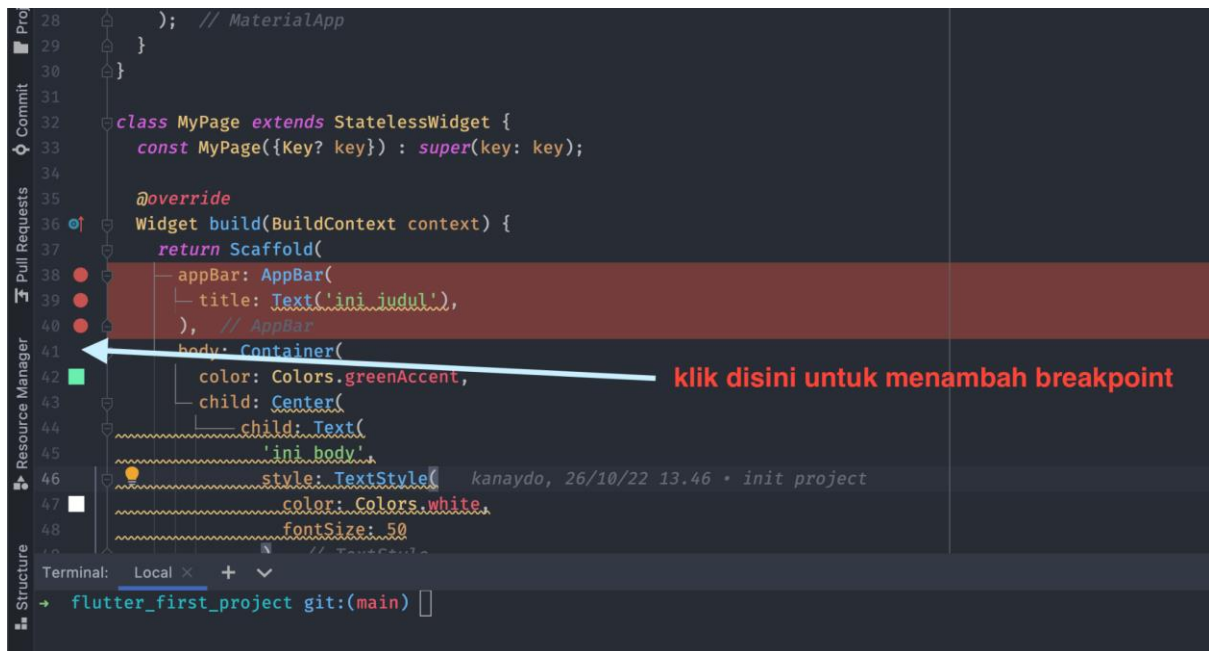
- Melakukan identifikasi error
Hal yang pertama dilakukan adalah mengidentifikasi kesalahan atau error yang terjadi pada program. Hal ini dilakukan agar tidak membuang banyak waktu dan perbaikan tepat sasaran.

- Menemukan sumber bug
Setelah identifikasi dilakukan dengan baik, selanjutnya adalah menemukan sumber atau lokasi dari kode yang error.
- Menganalisis bug
Pada tahap ini, debugger harus menganalisis baris kode yang menimbulkan error atau kesalahan. Hal tersebut dilakukan untuk memastikan apakah bug tersebut akan mempengaruhi fungsi yang lain. Selain itu, analisis ini diperlukan untuk mengantisipasi meningkatnya jumlah bug.
- Membuktikan hasil analisis
Setelah analisis selesai, seorang debugger perlu menemukan kemungkinan error yang lain pada program. Tahap ini dapat dilakukan secara otomatis menggunakan automated testing.
- Memperbaiki bug
Tahap terakhir adalah memperbaiki bug yang sudah ditemukan. Setelah perbaikan selesai, program akan diperiksa kembali untuk memastikan tidak ada error yang terjadi setelah perbaikan.

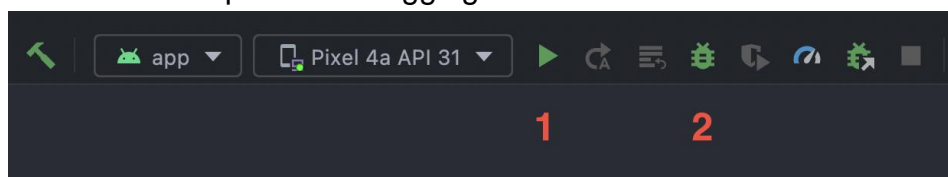
Notice:

Proses debugging ini pada umumnya digunakan ketika bekerja pada level data atau ketika tidak bisa melakukan identifikasi terhadap bug secara langsung, ataupun adanya bug yang kemungkinan dihasilkan oleh library dari pihak ketiga, maka dari itu fitur debugging ini cukup jarang digunakan.

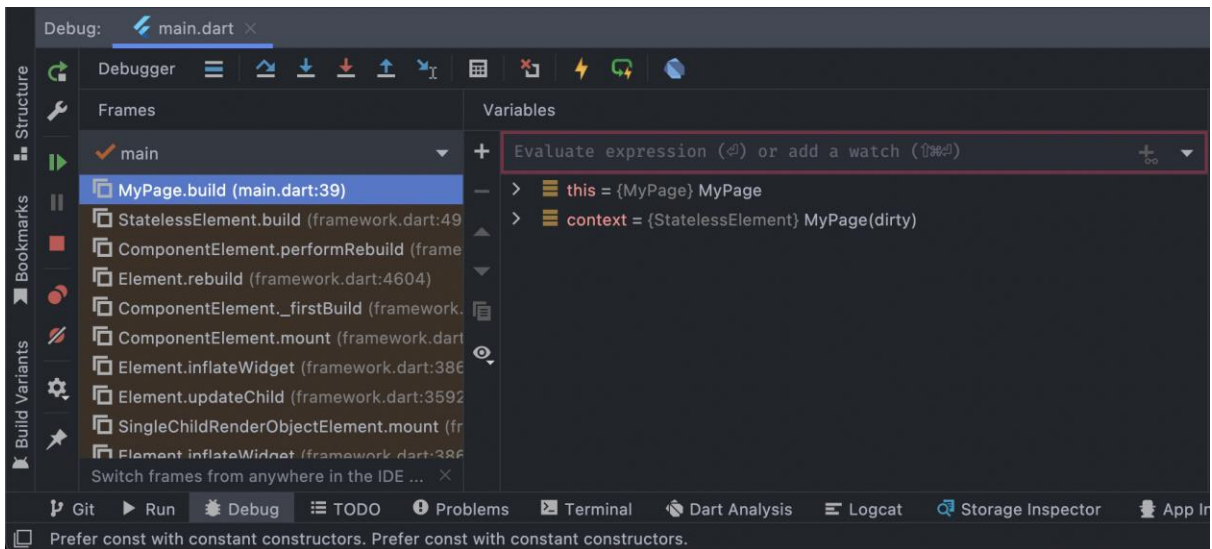
Proses debugging sendiri lebih disarankan menggunakan Android Studio dikarenakan Android Studio adalah sebuah IDE yang disarankan untuk development aplikasi, dan fitur debugging pada Android Studio sudah build-in dan tidak ada konfigurasi tambahan, untuk memulai proses debugging klik pada nomor baris yang ingin di-debug hingga tanda breakpoint (warna merah bulat) muncul di samping nomor baris:



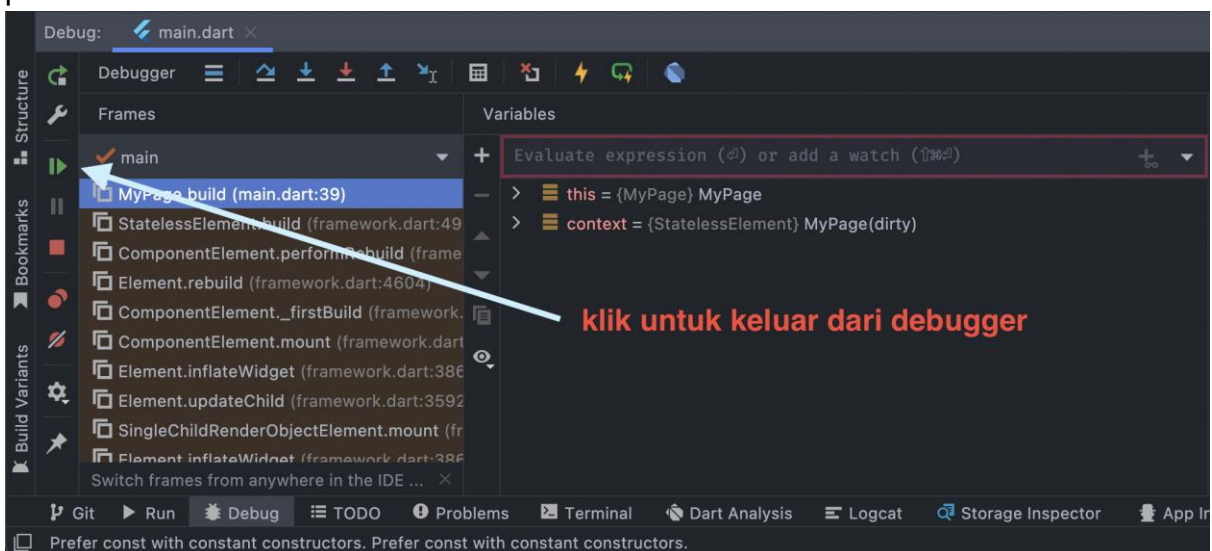
Jika sudah memberikan breakpoint pada kode yang ingin di-debug, klik pada icon nomor 2 untuk memulai proses debugging.



Proses debugging akan dimulai ketika program mencapai pada breakpoint yang telah ditentukan tadi. Pada umumnya akan ditandai dengan terbukanya section debug pada android studio secara otomatis seperti pada gambar berikut:



Selanjutnya dilakukan pengecekan dan proses debugging sesuai dengan masalah atau bug atau apapun yang perlu untuk di debug, atau pengecekan variabel atau class dan sebagainya, jika sudah maka bisa lepas dari debugger dengan cara klik pada menu berikut:



BAB VII FLUTTER WIDGET

7.1 Sekilas Tentang Flutter Widget

Flutter adalah framework bersifat open source yang dikembangkan oleh Google untuk membangun aplikasi multi-platform hanya dengan satu codebase. Hasil dari pengembangan aplikasi menggunakan Flutter bisa berupa aplikasi Android, iOS, Desktop, dan Website. Flutter memiliki dua komponen penting yang harus kamu ketahui, Software Development Kit (SDK) dan Framework User Interface.

Software Development Kit (SDK) merupakan tools-tools yang berfungsi untuk membuat aplikasi agar bisa dijalankan di berbagai platform. Sedangkan Framework User Interface adalah komponen seperti teks, tombol, dan lainnya yang dapat dikustomisasi sesuai kebutuhan kamu.

Desain antarmuka pengguna dalam Flutter melibatkan perakitan dan/atau pembuatan berbagai widget. Sebuah widget dalam Flutter mewakili deskripsi yang tetap dari bagian antarmuka pengguna; semua grafik, termasuk teks, bentuk, dan animasi dibuat menggunakan widget. Widget yang lebih kompleks dapat dibuat dengan memadukan widget-widget yang lebih sederhana. Untuk pengembangan mobile framework Flutter berisi dua set widget yang disesuaikan dengan bahasa desain tertentu. Widget **Material Design** menerapkan bahasa desain Google dengan nama yang sama, sedangkan widget **Cupertino** meniru desain iOS milik Apple.

Widget Flutter dibuat menggunakan framework modern yang mengambil inspirasi dari React. Ide utamanya adalah Anda membangun UI dari widget. Widget menjelaskan seperti apa tampilan mereka jika diberikan konfigurasi dan status saat ini. Saat status widget berubah, widget akan membuat ulang deskripsinya, yang mana kerangka kerja berbeda dengan deskripsi sebelumnya untuk menentukan perubahan minimal yang diperlukan dalam pohon render yang mendasari untuk transisi dari satu state ke state berikutnya.

Flutter sendiri memiliki semboyan ***“everything is a widget”*** atau semuanya adalah widget. Widget hanyalah potongan kecil UI yang dapat Anda gabungkan untuk membuat aplikasi yang lengkap. Widget bersarang di dalam satu sama lain untuk membangun aplikasi Anda. Bahkan root aplikasi Anda hanyalah sebuah widget.

Sebuah widget mungkin menampilkan sesuatu, mungkin membantu menentukan desain, mungkin membantu dengan tata letak, atau mungkin menangani interaksi.

- Widget sederhana yang menampilkan teks:

```
const Text('Hello World')
```

- Widget sederhana yang digunakan pengguna untuk berinteraksi:

```
const Button(onTap: ...callback...)
```

- Widget sederhana yang menambahkan warna latar belakang:

```
const BoxDecoration(latar belakang: Colors.blue)
```

Properti pada Widget

Setiap widget memiliki property, misalnya kita membuat sebuah tombol, lalu kita ingin agar warna text menjadi merah dengan text tebal. Untuk melakukan itu semua, maka tombol kita atur melalui propertinya.

```
const Text(  
  'You have pushed the button this many times:',  
  style: TextStyle(  
    color: Colors.red,  
    fontWeight: FontWeight.bold  
  ),  
)
```

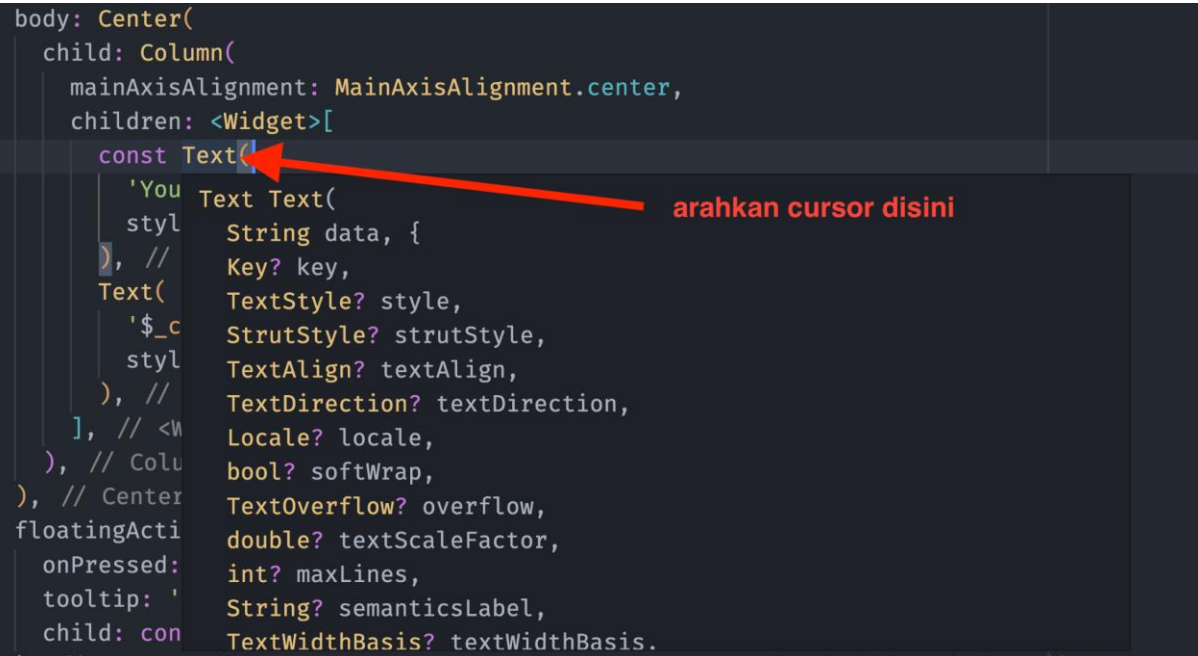
Masing-masing widget akan memiliki properti masing-masing dan memiliki tugas masing masing juga, widget bisa jadi memiliki banyak tapi hanya dibagi menjadi 2 bagian:

Required - properti yang wajib ada ketika kita menggunakan widget tersebut, contoh di atas teks 'You have pushed the button this many times:' adalah properti yang wajib pada widget Text, yang artinya widget Text tidak akan bisa digunakan / error jika tidak ada properti tersebut.

Optional - properti mungkin yang tidak dibutuhkan karena widget tersebut masih bisa berlajan tanpa properti tersebut, seperti contoh di atas properti style sebelumnya belum digunakan tapi widget tetap bisa digunakan. Jadi pada intinya properti digunakan ketika dibutuhkan perubahan pada widget tersebut.

Untuk melihat properti dari masing-masing widget bisa dilakukan dengan melakukan hover / tempatkan cursor pada widget yang diinginkan.

```
body: Center(  
  child: Column(  
    mainAxisAlignment: MainAxisAlignment.center,  
    children: <Widget>[  
      const Text(  
        'You have pushed the button this many times:',  
        style: Theme.of(context).textTheme.headline4,  
      ),  
      Text(  
        '$_counter',  
        style: Theme.of(context).textTheme.headline4,  
      ),  
    ],  
  ),  
),
```



Hover tersebut akan memunculkan properti yang dimiliki widget seperti pada gambar, untuk membedakan properti **required** dan **optional** adalah properti optional akan diberi tanda ?.

7.2 The Widget Tree

Untuk membuat suatu UI atau tampilan pada aplikasi dengan flutter, perlu dilakukan penyusunan widget yang nantinya akan membentuk suatu widget tree, sebagai contoh perhatikan code dari project yang telah dibuat sebelumnya.

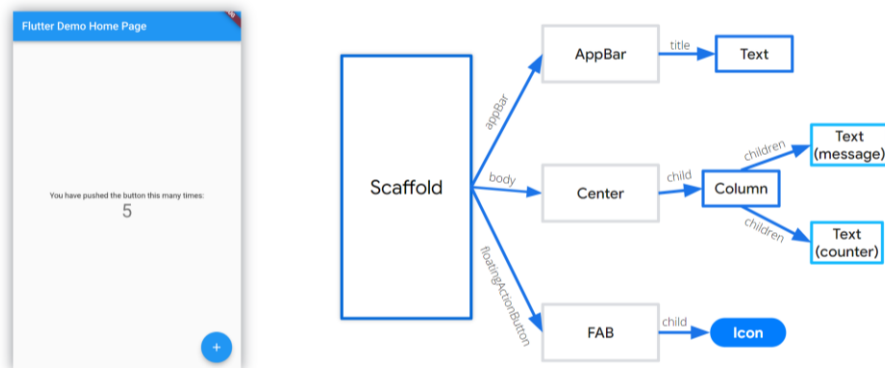
```
@override  
Widget build(BuildContext context) {  
  return Scaffold(  
    appBar: AppBar(  
      title: Text(widget.title),  
    ),  
    body: Center(  
      child: Column(  
        mainAxisAlignment: MainAxisAlignment.center,  
        children: <Widget>[  
          const Text(  
            'You have pushed the button this many times:',  
          ),  
          Text(  
            '$_counter',  
            style: Theme.of(context).textTheme.headline4,  
          ),  
        ],  
      ),  
    ),  
  ),  
),
```

```

    ),
    floatingActionButton: FloatingActionButton(
      onPressed: _incrementCounter,
      tooltip: 'Increment',
      child: const Icon(Icons.add),
    ),
  );
}

```

Scaffold sebagai widget induk memiliki parameter **AppBar**, **body**, dan **floatingActionButton**. kemudian, **AppBar** juga memiliki parameter **title** yang menggunakan widget **Text**. parameter **body** diisi dengan widget **Center** yang memiliki anak **Column**. **Column** yang memiliki dua **Text** sebagai anak. Widget **FloatingActionButton** memiliki callback **onPressed**, **tooltip** 'Increment', dan **Icon** sebagai anak. Maka secara grafis susunan widget tersebut bisa digambarkan seperti berikut:



7.3 Jenis Widget

Widget di flutter sangat banyak bahkan saat ini sudah mencapai ratusan widget (± 680 Widget), Namun pada praktik dan implementasi tidak semua widget ini digunakan, karena pada umumnya sebuah widget didesain untuk menyelesaikan suatu tugas spesifik, jadi jika kebutuhan aplikasi memang tidak membutuhkan widget tersebut, maka widget tersebut tidak akan digunakan. Beberapa widget akan bersifat umum dan akan sering digunakan, untuk mempermudah pemahaman widget akan dibagi menjadi 2 fungsi:

1. **View Widget** - Widget yang secara visual berfungsi sebagai tampilan atau akan muncul pada aplikasi. Beberapa widget view yang sering digunakan yaitu:
 - **Text**: widget untuk menampilkan tulisan.

Contoh: kita ingin menampilkan teks *'ini adalah stateless'*

```

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      home: Scaffold(
        body: Center(
          child: Text('ini adalah stateless'),
        ), // Center
      )); // Scaffold // MaterialApp
  }

```

- **Container:** widget yang digunakan untuk membuat dekorasi, seperti bentuk, latar belakang dll.
Contoh: teks sebelumnya ingin diberikan latar belakang merah, dan sedikit spacing

```

Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      body: Center(
        child: Container(
          color: Colors.red,
          padding: const EdgeInsets.all(8),
          child: const Text("haloo"),
        ), // Container
      ), // Center
    )); // Scaffold // MaterialApp
}

```

- **Image:** widget yang digunakan untuk menampilkan gambar.
Contoh: kita ingin menampilkan gambar dari internet

```

import 'package:flutter/material.dart';

class StatelessExample extends StatelessWidget {
  const StatelessExample({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: Image.network(
          'https://placeimg.com/640/480/nature'

```

```
    ),  
  ),  
);  
}  
}
```

- **Button:** widget yang berfungsi untuk menjalankan suatu aksi.

```
import 'package:flutter/material.dart';  
  
class StatelessExample extends StatelessWidget {  
  const StatelessExample({super.key});  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      body: Center(  
        child: ElevatedButton(  
          onPressed: () {  
            debugPrint('tombol ditekan');  
          },  
          child: const Text('press')),  
      ),  
    );  
  }  
}
```


2. **Layout Widget** - Widget yang berfungsi untuk menentukan tata letak dari View Widget, ada beberapa Layout Widget yang sering digunakan yaitu:

- **Center**: widget yang berfungsi meletakkan anaknya di tengah,

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

class StatelessExample extends StatelessWidget {
  const StatelessExample({super.key});

  @override
  Widget build(BuildContext context) {
    return const Scaffold(
      body: Center(
        child: Text('widget ini berada di tengah'),
      ),
    );
  }
}
```

- **Row**: widget yang berfungsi mengurutkan widget secara horizontal

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

class StatelessExample extends StatelessWidget {
  const StatelessExample({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: Row(
          mainAxisAlignment: MainAxisAlignment.center,
          children: const [
            FlutterLogo(),
            FlutterLogo(),
            FlutterLogo()
          ],
        ),
      ),
    );
  }
}
```

- **Column**: widget yang berfungsi mengurutkan widget secara vertikal.

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

class StatelessExample extends StatelessWidget {
  const StatelessExample({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: const [
            FlutterLogo(),
            FlutterLogo(),
            FlutterLogo()
          ],
        ),
      ),
    );
  }
}
```

- **ListView**: widget yang berfungsi mengurutkan widget secara vertikal dan dilengkapi dengan scroll.

Contoh: kita ingin menampilkan 100 logo flutter secara vertikal:

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

class StatelessExample extends StatelessWidget {
  const StatelessExample({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: ListView(
        children: List.generate(
          100,
          (index) => const FlutterLogo(),
        ).toList(),
      ),
    );
  }
}
```

```
}  
}
```

7.4 Widget Lifecycle Event

Selama aplikasi dijalankan mungkin akan memiliki event yang terjadi pada UI atau tampilannya dalam hal ini widget. Lifecycle Event ini hanya akan berlaku atau diimplementasikan pada custom widget yang kita buat sendiri bukan widget standard dari flutter yang dijelaskan sebelumnya, namun perubahan event tersebut bisa akan berdampak pada widget standard tersebut. Secara lifecycle widget dibagi menjadi 2 jenis, yaitu:

1. Stateless Widget

Widget yang tidak memiliki event apapun atau perubahan data apapun di dalamnya setelah ditampilkan. Sebagai contoh:

```
class StatelessExample extends StatelessWidget {  
  const StatelessExample({super.key});  
  
  @override  
  Widget build(BuildContext context) {  
    return const Scaffold(  
      body: Center(  
        child: Text('ini adalah stateless'),  
      ),  
    );  
  }  
}
```

2. Statefull Widget

Widget yang tidak memiliki event atau perubahan data apapun di dalamnya setelah ditampilkan. Sebagai contoh adalah aplikasi akan menampilkan apakah tombol sudah ditekan atau belum:

```
import 'package:flutter/material.dart';  
  
class StatefulExample extends StatefulWidget {  
  const StatefulExample({super.key});  
  
  @override  
  State<StatefulExample> createState() =>  
    _StatefulExampleState();  
}
```

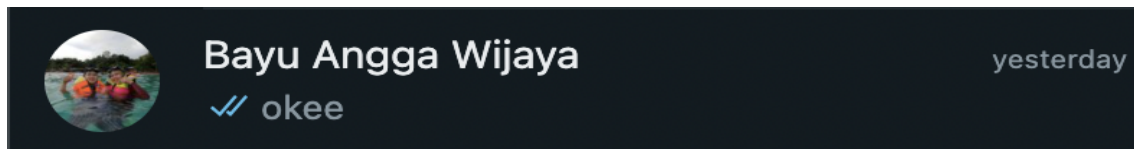
```

class _StatefulExampleState extends State<StatefulExample> {
  String text = 'tombol belum di-klik';
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Column(
        children: [
          Text(text),
          ElevatedButton(
            onPressed: () {
              setState(() {
                text = 'tombol sudah ditekan';
              });
            },
            child: const Text('tekan')
          )
        ],
      ),
    );
  }
}

```

7.5 Composing (Use Case)

Composing pada flutter adalah penggabungan beberapa widget menjadi sebuah widget utuh dan memiliki fungsi dalam aplikasi, baik dari segi tata letak widget, content di dalam widget, maupun tujuan dari pembuatan widget itu sendiri. Composing pada umumnya bisa diimplementasikan dari desain aplikasi yang sudah ada ataupun desain terpisah. Sebagai studi kasus, dalam pembelajaran ini akan diimplementasikan chat item dari aplikasi pesan whatsapp. Seperti terlihat pada gambar dibawah:

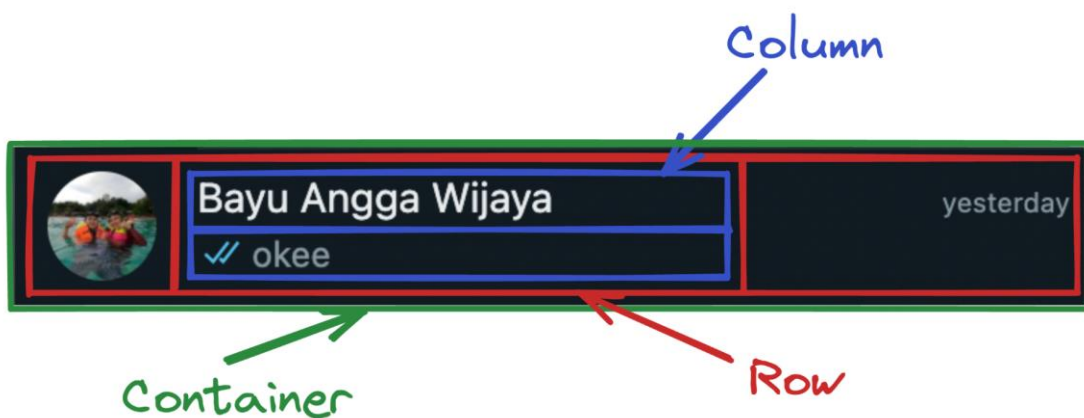


Sebelum dilakukan composing pada UI diatas, ada baiknya lakukan analisis terlebih dahulu widget-widget apa saja yang ada pada desain tersebut, untuk mempermudah analisis, ada baiknya lakukan pemisahan antara view widget dan layout widget

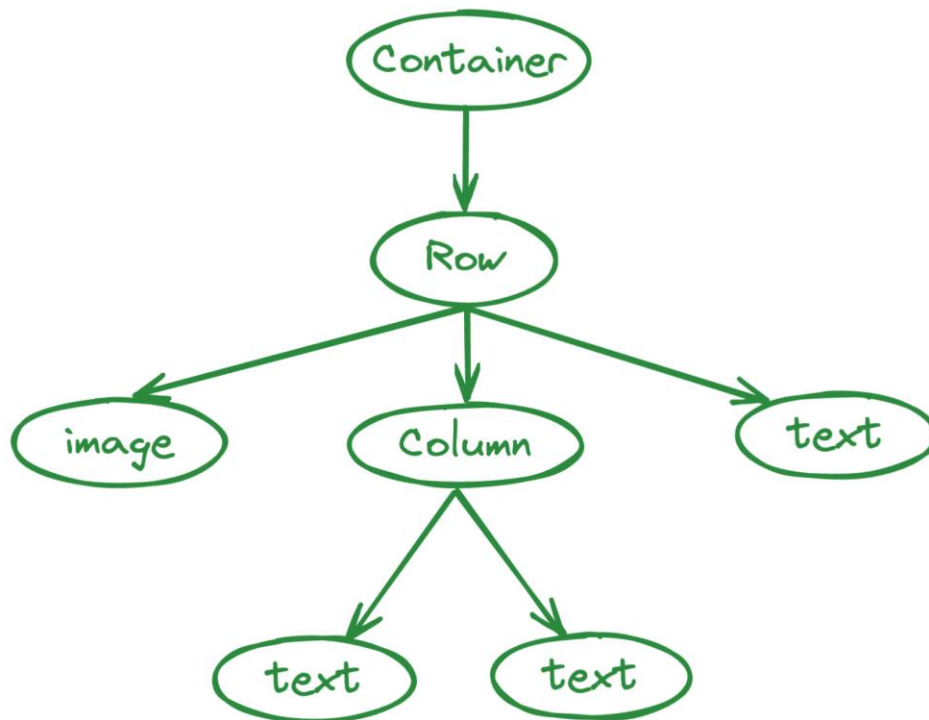
- View Widget atau widget-widget yang terlihat secara visual:



- **Layout Widget** atau widget yang menentukan tata letak dari **View Widget**



Jika analisis widget sudah diketahui maka, selanjutnya bisa dilakukan penyusunan widget tree, sehingga bisa ditentukan widget apa saja yang menjadi widget inti, dan mana widget content:



Jika semua analisis widget dan widget tree sudah bisa disimpulkan, maka composing UI sudah bisa dilakukan dengan cara mengikuti alur dari widget tree yang sudah dibuat, pembacaan widget tree dilakukan mulai dari atas, yang artinya composing akan dimulai dengan pengerjaan widget **Container**, kemudian **Row**, kemudian **Column**.

Container

Container adalah widget yang digunakan untuk memberikan dekorasi pada widget lain, contoh untuk memberikan latar belakang pada widget. Pada kasus kali ini yang dibutuhkan dari Container adalah background color dan adanya spacing atau jarak.

```
class MyPage extends StatelessWidget {
  const MyPage({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Container(
        padding: const EdgeInsets.all(8),
        color: Colors.blueGrey,
      )
    );
  }
}
```

Maka hasil dari kode diatas terlihat seperti berikut:



7.6 Row – Horizontal View

Selanjutnya adalah widget row, yang memberikan tata letak secara horizontal pada widget anaknya, pada kasus ini widget anaknya ada 3 yaitu, **Image**, **Column** dan **Text**.

beberapa dari widget anak ini perlu dicustom, yaitu:

- Widget **Image**, ada perubahan yaitu ukuran yang kecil dan berbentuk oval. Maka untuk itu akan lebih baik menggunakan widget yang memiliki fitur oval yaitu widget **CircleAvatar**

```
CircleAvatar(  
  backgroundImage: NetworkImage(  
    'https://placeimg.com/640/480/people'  
  )  
),
```

- Widget **Column** yang memakan semua space yang tersisa. Maka widget **Column** harus dibungkus ke dalam widget **Expanded**

```
Expanded(  
  Column(  
    Text(  
      'Hello World'  
    ),  
  ),  
),
```

```
        child: Column()  
      ),
```

Sehingga jika dilakukan penggabungan pada semua widget composing dari row akan terlihat seperti berikut:

```
class MyPage extends StatelessWidget {  
  const MyPage({Key? key}) : super(key: key);  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      body: Container(  
        padding: const EdgeInsets.all(8),  
        color: Colors.blueGrey,  
        child: Row(  
          children: [  
            CircleAvatar(  
              backgroundImage: NetworkImage(  
                'https://placeimg.com/640/480/people'  
              )  
            ),  
            Expanded(  
              child: Column()  
            ),  
            Text('yesterday')  
          ],  
        ),  
      ),  
    );  
  }  
}
```

Jika kode diatas dijalankan, hasil sementara dari composing UI terlihat seperti gambar dibawah. Pada gambar terlihat kalau widget di tengah masih belum memiliki konten, namun dari segi tata letak sudah terlihat bahwa penempatan widget sudah berada pada posisi yang benar.

12:17



yesterday

7.8 Column – Vertical View

Pada tahap sebelumnya kita sudah melakukan deklarasi pada widget column, maka tahap selanjutnya adalah untuk mengisi konten atau widget anak dari widget column tersebut, dari widget tree diatas, kita bisa melihat bahwa widget column memiliki 2 anak yaitu:

- Widget Text untuk menampilkan tulisan `Bayu Angga Wijaya`

```
Text('Bayu Angga Wijaya')
```

- Widget Row untuk menampilkan Icon Centang dan pesan terakhir

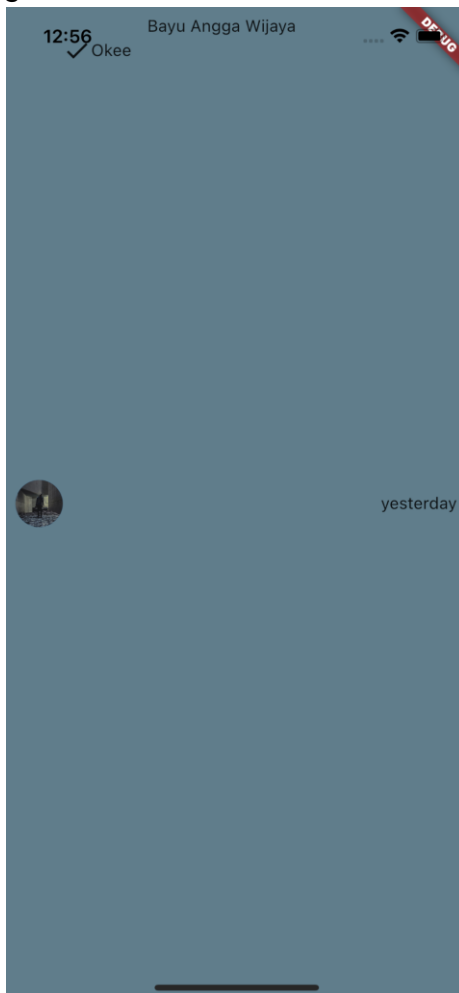
```
Row(  
  children: [  
    Icon(Icons.check),  
    Text('Okee')  
  ],  
)
```

Kedua widget tersebut bisa langsung diimplementasikan seperti kode berikut:

```
class MyPage extends StatelessWidget {  
  const MyPage({Key? key}) : super(key: key);  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      body: Container(  
        padding: const EdgeInsets.all(8),  
        color: Colors.blueGrey,  
        child: Row(  
          children: [  
            CircleAvatar(  
              backgroundImage: NetworkImage(  
                'https://placeimg.com/640/480/people'  
              )  
            ),  
            Expanded(  
              child: Column(  
                children: [  
                  Text('Bayu Angga Wijaya'),  
                  Row(  
                    children: [  
                      Icon(Icons.check),
```

```
Text('Okee')
],
),
],
),
),
Text('yesterday')
],
),
),
);
}
```

Namun ketika dijalankan, hasil dari kode diatas akan berantakan tata letaknya, seperti yang terlihat pada gambar berikut:



Hal ini disebabkan oleh properti default dari widget **Column** dimana konten di dalam **Column** akan diberikan efek rata tengah secara default, dan **Column** jika digunakan dalam **Row** akan diletakkan pada posisi start secara vertikal yaitu atas properti yang

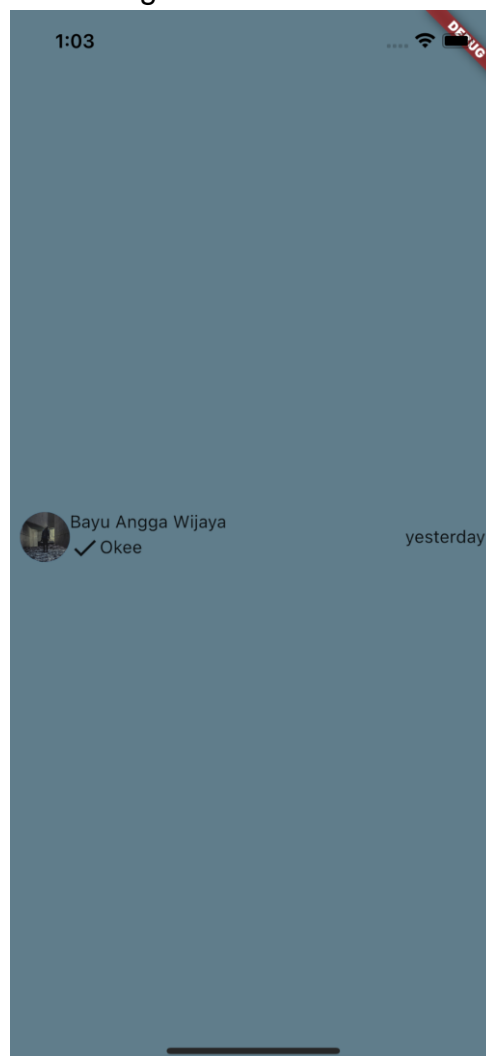
bertanggung jawab untuk hal ini adalah:

- **mainAxisAlignment:** properti untuk penyesuaian (alignment) widget **Column** terhadap widget induknya (pada kasus ini **Row**). widget **Column** yang kita gunakan harus berada di tengah dari widget **Row**, dimana secara default jika widget **Column** menjadi anak dari **row**, maka akan diletakkan pada posisi **start**.
- **crossAxisAlignment:** properti untuk penyesuaian (alignment) dari widget-anak **column**. Dimana secara default **Column** akan membuat semua anaknya menjadi rata tengah sementara kita membutuhkan rata kanan (**start**).

Maka untuk memperbaiki masalah tersebut, perlu dilakukan modifikasi pada kedua properti tersebut sesuai dengan kebutuhan yang sudah dijabarkan sebelumnya.

```
mainAxisAlignment: MainAxisAlignment.center,  
crossAxisAlignment: CrossAxisAlignment.start,
```

Sehingga layout dari anak dari widget **Column** bisa sesuai:



7.8 Penyesuaian - Customization

1. Jarak antara gambar dengan text sangat rapat, maka widget Column harus dibungkus dengan widget Padding

```
Expanded(  
  child: Padding(  
    padding: const EdgeInsets.all(8.0),  
    child: Column(  
      mainAxisAlignment:  
MainAxisAlignment.center,  
      crossAxisAlignment:  
CrossAxisAlignment.start,  
      children: [  
        Text('Bayu Angga Wijaya'),  
        Row(  
          children: [  
            Icon(Icons.check),  
            Text('Okee')  
          ],  
        ),  
      ],  
    ),  
  ),  
)
```

2. Tulisan `Bayu Angga Wijaya` menggunakan huruf tebal dan berwarna putih:

```
Text(  
  'Bayu Angga Wijaya',  
  style: TextStyle(  
    color: Colors.white,  
    fontWeight: FontWeight.bold  
  ),  
)
```

3. Tulisan `yesterday` berwarna abu-abu:

```
Text(  
  'yesterday',  
  style: TextStyle(  
    color: Colors.gray  
  ),  
)
```

4. Tulisan `okee` berwarna abu-abu:

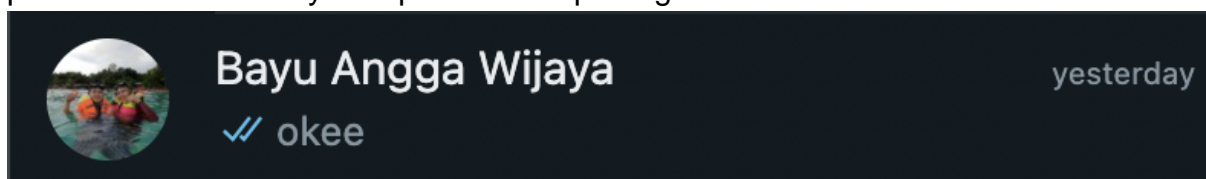
```
Text(  
  'Okee',  
  style: TextStyle(  
    color: Colors.grey  
  ),  
)
```

5. Icon berukuran lebih kecil dan berwarna biru:

```
Icon(  
  Icons.check,  
  size: 18,  
  color: Colors.blue,  
)
```

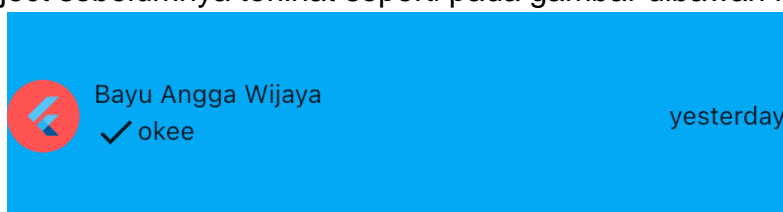
7.9 Composing and Customization View

Composing and Customization pada flutter adalah perubahan atau penyesuaian tampilan widget secara visual, kustomisasi ini bisa meliputi dimensi (jarak/ukuran), warna maupun bentuk dari widget itu sendiri. Alasan dilakukannya composing dan customization adalah untuk memperoleh visual sesuai dengan keinginan atau sesuai dengan kreasi dari pembuat program itu sendiri. Sebagai studi kasus, materi ini akan melanjutkan desain UI dan kode yang sudah dibuat pada pertemuan sebelumnya. Seperti terlihat pada gambar dibawah:



Dimana desain diatas bisa kita ubah disesuaikan dengan kreasi masing-masing, tapi harus tetap mengikuti aturan kustomisasi flutter. Kode program juga akan tetap melanjutkan kode program dari pertemuan sebelumnya dari pembahasan mengenai layouting widget. Code snippet untuk pertemuan minggu lalu bisa dilihat dari link berikut: https://raw.githubusercontent.com/kanaydo/composing_b/main/lib/main.dart

Visual dari project sebelumnya terlihat seperti pada gambar dibawah ini:



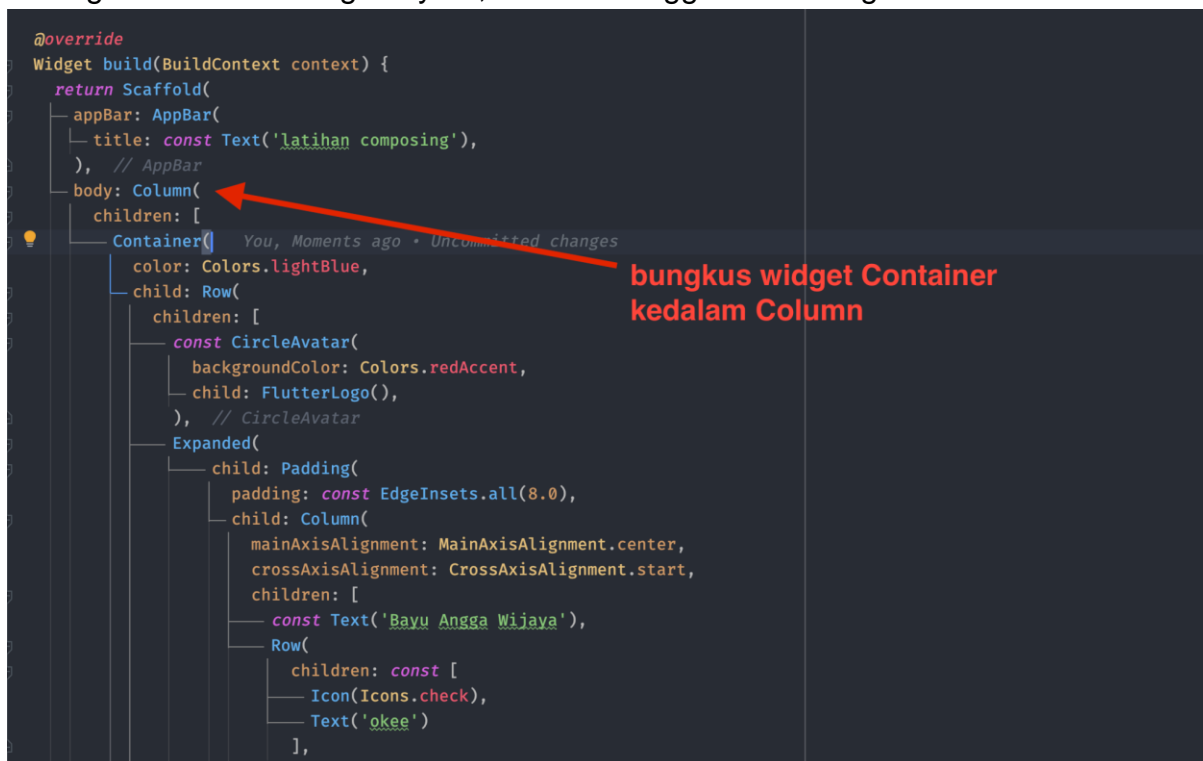
Dari studi kasus kali ini, beberapa widget yang bisa dilakukan Composing ataupun Customization adalah:

- **Container**: Memberikan dekorasi pada widget Container yang meliputi border dengan warna dan sudut melengkung pada masing-masing sisinya.
- **Text**: penyesuaian warna dan ukuran teks, dimana tulisan `Bayu Angga Wijaya` akan dicetak tebal dan memiliki ukuran yang cukup besar, tulisan `okee` dan `yesterday` akan memiliki ukuran yang lebih kecil.
- **Icon**: Penyesuaian warna dan ukuran pada icon checklist.

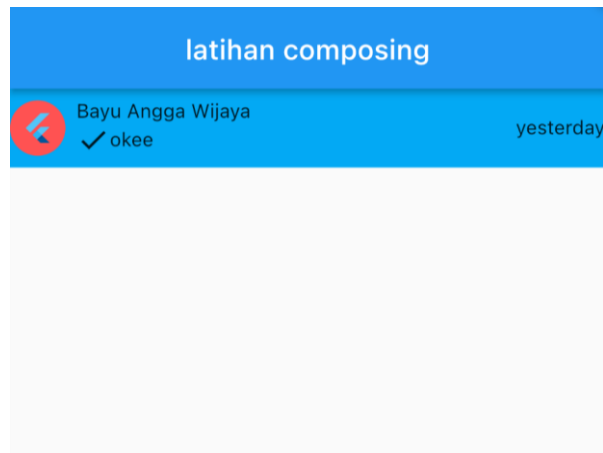
Normalisasi Project

Untuk project sebelumnya, perlu dilakukan penyesuaian untuk mendapatkan visual sesuai dengan desain. Seperti yang terlihat pada aplikasi, seluruh halaman dari aplikasi sepenuhnya terpenuhi dengan latar warna biru, hal tersebut dikarenakan oleh widget **Container** yang kita letakkan di dalam properti **body**, sehingga body secara keseluruhan adalah **Container**, maka dari itu perlu disesuaikan akar bentuk yang seharusnya dari **Container** bisa terlihat, widget Container tersebut bisa dibungkus kedalam widget layout, contoh menggunakan widget **Column**:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('latihan composing'),
    ), // AppBar
    body: Column(
      children: [
        Container(
          color: Colors.lightBlue,
          child: Row(
            children: [
              const CircleAvatar(
                backgroundColor: Colors.redAccent,
                child: FlutterLogo(),
              ), // CircleAvatar
              Expanded(
                child: Padding(
                  padding: const EdgeInsets.all(8.0),
                  child: Column(
                    mainAxisAlignment: MainAxisAlignment.center,
                    crossAxisAlignment: CrossAxisAlignment.start,
                    children: [
                      const Text('Bayu Angga Wijaya'),
                      Row(
                        children: const [
                          Icon(Icons.check),
                          Text('okee')
                        ],
                      ),
                    ],
                  ),
                ),
              ),
            ],
          ),
        ),
      ],
    ),
  );
}
```



Maka, visual dari program akan terlihat seperti pada gambar berikut:



Seperti yang terlihat pada gambar diatas, widget yang sudah dikerjakan akan berpindah ke atas, hal ini dikarenakan oleh widget **Column** akan memulai urutan widget anak mulai dari posisi awal secara vertikal, seperti yang sudah dibahas pada pertemuan sebelumnya yaitu properti **mainAxisAlignment** secara default adalah **MainAxisAlignment.start**.

Custom Container

Seperti yang sudah disebutkan sebelumnya, pada studi kasus ini, kita akan melakukan kustomisasi atau dekorasi pada beberapa aspek Container, yaitu:

- border dengan warna,
- sudut melengkung pada masing-masing sisinya.

Dekorasi pada **Container** dilakukan pada properti **decoration** dengan melakukan perubahan pada class **BoxDecoration**, sehingga jika sebelumnya kita sudah menggunakan properti yang berhubungan dengan decoration, **wajib** dipindahkan kedalam decoration, adapun properti dekorasi yang sudah kita gunakan adalah **color**, sehingga **color** wajib dipindahkan ke dalam **BoxDecoration**.


```
return Scaffold(
  appBar: AppBar(
    title: const Text('latihan composing'),
  ), // AppBar kanaydo, 32 minutes ago • starting project
  body: Column(
    children: [
      Container(
        decoration: BoxDecoration(
          color: Colors.lightBlue,
        ), // BoxDecoration
        child: Row(
          children: [
            const CircleAvatar(
              backgroundColor: Colors.redAccent,
              child: FlutterLogo(),
            ), // CircleAvatar
            Expanded(
              child: Padding(
                padding: const EdgeInsets.all(8.0),
                child: Column(
                  mainAxisAlignment: MainAxisAlignment.center,
                  crossAxisAlignment: CrossAxisAlignment.start,
                  children: [
                    const Text('Bayu Angga Wijaya'),
                    Row(
                      children: const [
                        Icon(Icons.check),
                        Text('okee')
                      ],
                    ) // Row
                  ],
                ), // Column
              ) // Padding
            ), // Expanded
            const Text('yesterday')
          ],
        ),
      ],
    ),
  ),
```

Tambahkan properti decoration ke Container, dan diisi dengan class BoxDecoration.

Kemudian pindahkan properti color kedalam class BoxDecoration

Sehingga untuk pengerjaan kustomisasi berikutnya akan dilakukan di dalam class **BoxDecoration**.

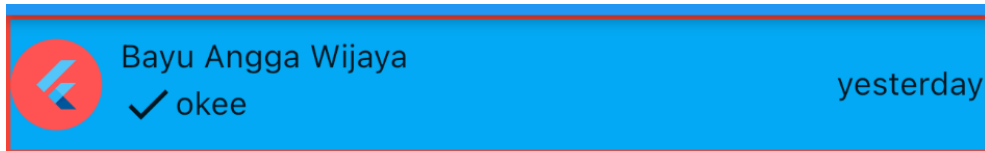
Yang pertama adalah pemberian Border dan Warna Border, class BoxDecoration akan diberikan properti border:

```
border: Border.all(
  color: Colors.red,
  width: 2
),
```

Seperti yang terlihat properti border akan diisi dengan class **Border.all** (dibaca: border dengan constructor all), yang artinya Container akan memiliki border pada semua sisinya, karna Container secara default adalah bentuk persegi maka akan memiliki 4 sisi, kemudian class Border tersebut memiliki 2 properti tambahan yaitu:

- **color**: untuk memberikan warna pada border tersebut.
- **width**: untuk menentukan ketebalan dari border tersebut.

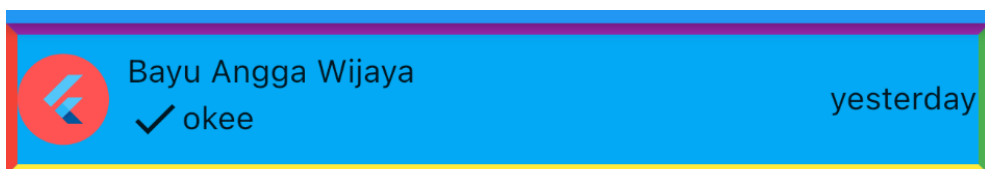
Maka kita akan memperoleh border dengan warna merah seperti pada gambar dibawah ini:



Pada kasus lain, jika ingin memberikan border hanya pada sisi tertentu, maka bisa menggunakan class **Border** (tanpa constructor all), kemudian definisikan sisi mana saja yang ingin diberikan border, bisa salah satu, atau semuanya:

```
border: Border(  
  top: BorderSide(  
    color: Colors.purple,  
    width: 5  
  ),  
  left: BorderSide(  
    color: Colors.red,  
    width: 5  
  ),  
  right: BorderSide(  
    color: Colors.green,  
    width: 5  
  ),  
  bottom: BorderSide(  
    color: Colors.yellow,  
    width: 5  
  ),  
)
```

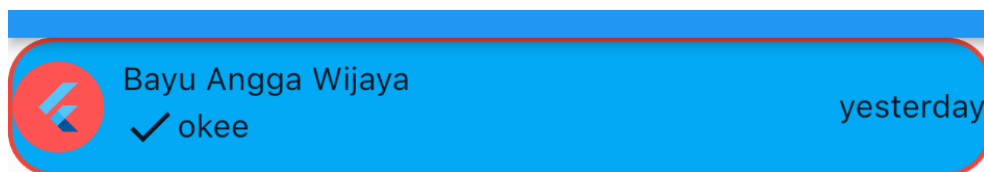
Maka hasilnya, border pada container akan terlihat seperti pada gambar berikut, sebagai catatan: tidak semua sisi harus digunakan, dan properti pada masing-masing sisi tidak harus sama.



Tahap berikutnya, kita akan membuat masing-masing sudut dari border tersebut melengkung, efek melengkung bisa dilakukan dengan memberikan property **borderRadius** pada **BoxDecoration**, diisi dengan class **BorderRadius.circular** (dibaca: BorderRadius constructor circular) dengan nilai 20 sebagai derajat kelengkungannya, semakin besar nilainya, maka akan semakin besar efek lengkungan yang dihasilkan:

```
borderRadius: BorderRadius.circular(20)
```

Secara visual, border yang dihasilkan akan terlihat seperti pada gambar dibawah ini:



Pada kasus lain, ada kemungkinan kita hanya membutuhkan efek lengkungan pada beberapa sisi saja, contohnya hanya pada sudut kiri atas dan sudut kanan bawah, maka bisa digunakan class **BorderRadius.only** (dibaca: BorderRadius constructor only):

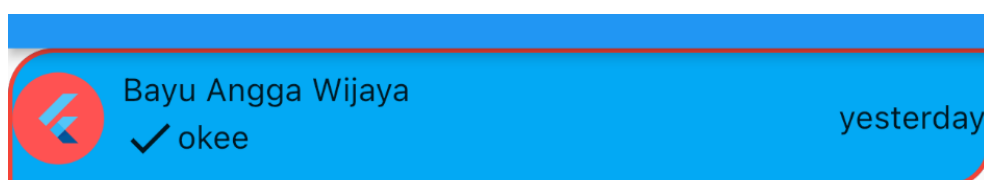
```
borderRadius: BorderRadius.only(  
  topLeft: Radius.circular(20),  
  bottomRight: Radius.circular(20)  
)
```

Pada class ini, kita bisa definisikan masing-masing sudut yang kita inginkan dan besaran sudut kelengkungan masing-masing sudut, adapun properti sudut yang bisa digunakan adalah:

- topLeft
- bottomRight
- topRight
- bottomLeft

Masing-masing properti ini bisa digunakan sendirian, atau digunakan secara bersamaan, dan memiliki besaran sudut masing-masing juga.

Pada contoh diatas, kita hanya memberikan efek lengkung pada 2 sudut saja, maka secara visual akan terlihat seperti pada gambar berikut:



Kustomisasi Dimensi pada Container

Seperti yang terlihat pada gambar, adanya spacing / jarak yang terlalu dekat. Baik jarak antara border dan widget di dalamnya (image dan text), maupun jarak antara border dengan sisi layar (kiri dan kanan). Permasalahan spacing pada flutter bisa diselesaikan dengan menggunakan class **EdgeInsets**, class **EdgeInsets** sendiri mempunyai beberapa constructor untuk beberapa fungsi tertentu, namun yang umum digunakan hanya 3 yaitu:

- **EdgeInsets.all**: memberikan jarak pada semua sisi:

```
EdgeInsets.all(8),
```

Kode diatas memiliki arti memberikan jarak sebanyak 8 ppi pada semua sisi (kiri, kanan, atas dan bawah).

- **EdgeInsets.symmetric**: memberikan jarak secara simetris atau nilai sama, baik secara vertikal maupun horizontal.

```
EdgeInsets.symmetric(horizontal: 8),
```

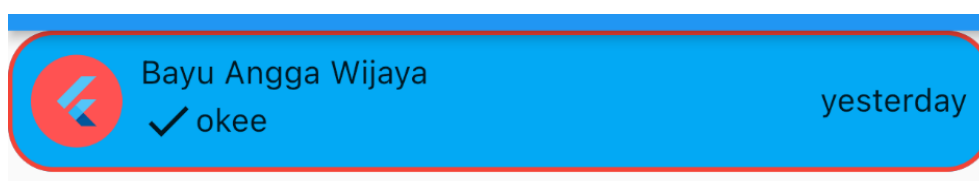
Kode diatas memiliki arti memberikan jarak sebanyak 8 ppi pada sisi horizontal (kiri dan kanan).

- **EdgeInsets.only**: memberikan jarak secara spesifik pada sisi tertentu.

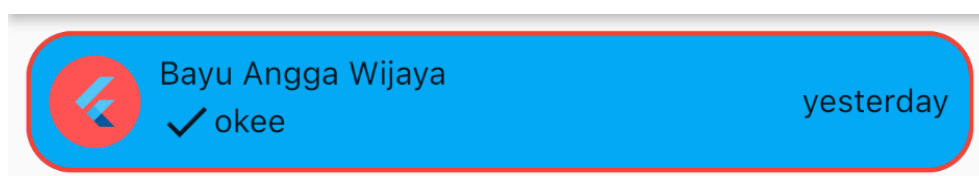
```
EdgeInsets.only(  
  left: 8,  
  right: 16  
)
```

Kode diatas memiliki arti memberikan jarak sebanyak 8 ppi pada sisi kiri dan 16 ppi pada sisi kanan.

Untuk menyelesaikan masalah jarak yang terlalu rapat antara border dan widget didalamnya, kita bisa menggunakan properti **padding** pada **Container** dan diisi dengan class **EdgeInsets**.



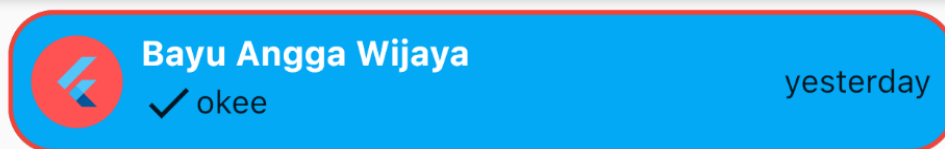
Sedangkan untuk menyelesaikan masalah jarak antara border dengan sisi layar (kiri dan kanan). Kita bisa menggunakan properti **margin** pada **Container** dan diisi dengan class **EdgeInsets**:



Penyesuaian Lain

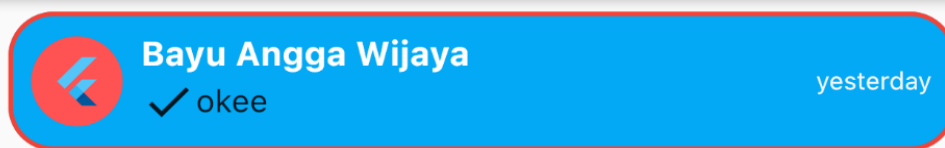
1. Tulisan `Bayu Angga Wijaya` menggunakan huruf tebal dan ukurannya lebih besar:

```
Text(  
  'Bayu Angga Wijaya',  
  style: TextStyle(  
    fontWeight: FontWeight.bold,  
    fontSize: 15,  
    color: Colors.white  
  ),  
)
```



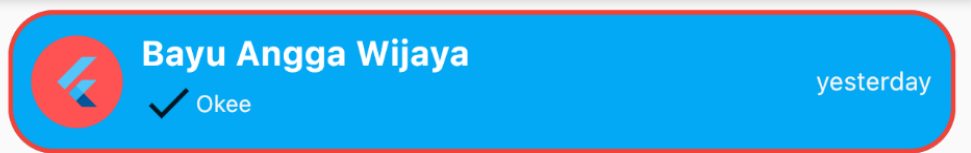
2. Tulisan `yesterday` memiliki warna dan ukurannya lebih kecil:

```
Text(  
  'yesterday',  
  style: TextStyle(  
    fontSize: 11,  
    color: Colors.grey.shade100  
  ),  
)
```



3. Tulisan `okee` berwarna abu-abu:

```
Text(  
  'Okee',  
  style: TextStyle(  
    fontSize: 10,  
    color: Colors.grey.shade100  
  ),  
)
```



4. Icon berukuran lebih kecil dan memiliki warna:

```
Icon(  
  Icons.check,  
  size: 15,  
  color: Colors.lightBlueAccent,  
)
```



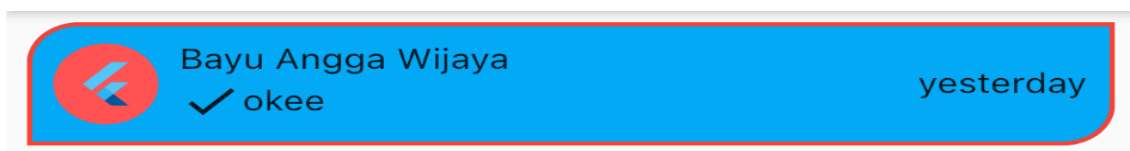
7.10 Implementasi ListView

ListView adalah bentuk default dari sebuah ListView class. Dengan ListView maka children atau widget yang ada di dalamnya akan menjadi scrollable (bisa di scroll). Penggunaan default ListView ini hanya untuk widget yang bersifat statis. Statis yang dimaksud bukan untuk isi kontennya melainkan lebih kepada jumlah widget di dalamnya.

Penggunaan default ListView :

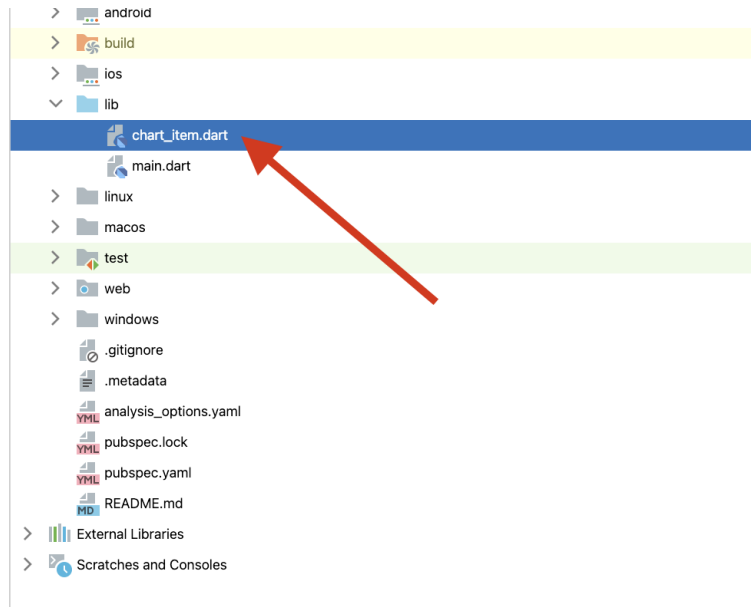
Contoh kasus misalkan kita ingin membuat sebuah halaman detail aplikasi baca berita. Dimana biasanya untuk halaman tersebut umumnya memiliki item judul dan deskripsi. Untuk panjang deskripsi pada sebuah berita beragam dan bisa sangat panjang.

Implementasi ListView pada materi ini akan melanjutkan dari materi sebelumnya, yaitu widget item dari aplikasi pesan whatsapp, seperti yang terlihat pada gambar berikut ini.



Untuk mempermudah proses pembacaan kode maka ada baiknya visual diatas dipisah menjadi Standalone Widget, sehingga nantinya saat diimplementasikan di ListView tidak akan membuat pengulangan penulisan kode yang banyak.

Pemisahan kode menjadi Standalone Widget bisa dilakukan dengan membuat file baru dengan nama *chat_item.dart* dan diletakkan di dalam folder lib:



Kemudian pindahkan pindahkan keseluruhan kode yang berfungsi membuat widget tersebut kedalam file tersebut, pada kasus ini keseluruhan widget kita dibungkus dalam widget **Container**, dengan menggunakan stateless widget sebagai widget utama dengan nama widget ChatItem, seperti yang terlihat pada kode dibawah ini:

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

class ChatItem extends StatelessWidget {
  const ChatItem({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Container(
      margin: EdgeInsets.all(8),
      padding: EdgeInsets.symmetric(horizontal: 8),
      decoration: BoxDecoration(
        color: Colors.lightBlue,
        borderRadius: BorderRadius.only(
          topLeft: Radius.circular(20),
          bottomRight: Radius.circular(20)
        ),
        border: Border.all(
          color: Colors.red,
          width: 2
        )
      ),
      child: Row(
```

```

children: [
  const CircleAvatar(
    backgroundColor: Colors.redAccent,
    child: FlutterLogo(),
  ),
  Expanded(
    child: Padding(
      padding: const EdgeInsets.all(8.0),
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        crossAxisAlignment: CrossAxisAlignment.start,
        children: [
          const Text('Bayu Angga Wijaya'),
          Row(
            children: const [
              Icon(Icons.check),
              Text('okee')
            ],
          ),
        ],
      ),
    ),
  ),
  const Text('yesterday')
],
);
}

```

Sampai pada tahap ini, kita sudah berhasil membuat sebuah widget yang memiliki sifat standalone dan independen. Langkah selanjutnya adalah menggunakan widget standalone tersebut di dalam widget ListView seperti yang sudah dibahas sebelumnya. Implementasi ListView sendiri bisa dilakukan dengan beberapa cara, dan memiliki sifat dan fungsionalitas berbeda juga.

ListView

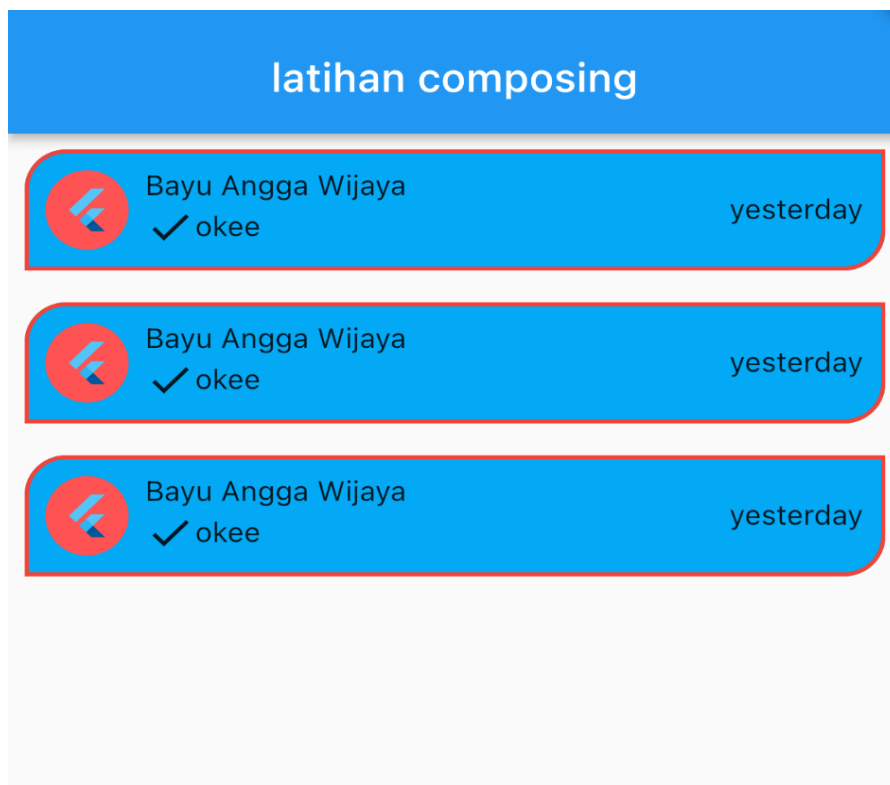
Widget ListView digunakan untuk membuat list atau daftar dengan item didalamnya yang memiliki jumlah sudah pasti jumlahnya(fix), dan secara langsung harus dideskripsikan sebagai widget children didalamnya. Sebagai contoh kasus, kita akan mengimplementasikan 3 chat item di dalam ListView, maka kita bisa mengganti

properti body pada widget Scaffold dengan widget ListView dan mendeklarasikan 3 widget ChatItem sebagai children, seperti kode yang terlihat dibawah ini:

```
class MyPage extends StatelessWidget {
  const MyPage({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('latihan ListView'),
      ),
      body: ListView(
        children: [
          ChatItem(),
          ChatItem(),
          ChatItem()
        ],
      ),
    );
  }
}
```

Jika kode diatas dijalankan, maka hasil yang akan ditampilkan adalah seperti gambar berikut ini:



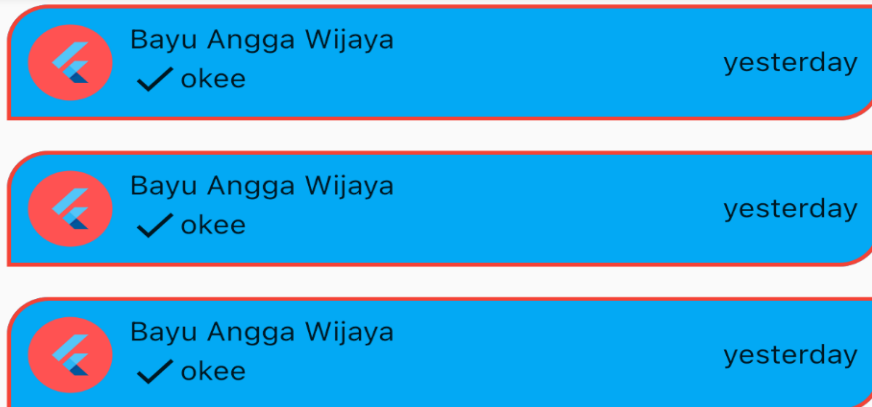
Terlihat pada contoh diatas kita menggunakan widget ChatItem sebagai anak dari ListView, namun tidak semata mata childen dari widget ListView ini harus menggunakan ChatItem, kita tetap bisa menggunakan widget lain sebagai widget anak dari ListView contoh misalkan kita ingin menggunakan widget **Text**:

```
class MyPage extends StatelessWidget {
  const MyPage({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('latihan composing'),
      ),
      body: ListView(
        children: [
          ChatItem(),
          ChatItem(),
          ChatItem(),
          Text('ini adalah teks di dalam listview'),
          Container(
            padding: EdgeInsets.symmetric(horizontal: 8),
            child: Text('ini adalah container di dalam listview')
          ),
        ],
      ),
    );
  }
}
```

Maka hasilnya akan seperti pada gambar berikut ini:

latihan composing



ini adalah teks didalam listview
ini adalah container didalam listview

ListView.builder

Implementasi lain dari widget ListView adalah menggunakan constructor builder yang digunakan untuk list yang bersifat dinamis (jumlah list item mengikuti dari jumlah data) maka gunakan ListView.builder. ListView.builder memiliki dua properti utama yaitu itemCount (jumlah list item) dan itemBuilder (untuk membangun tampilan dari list item). Tidak seperti widget ListView yang kita bahas sebelumnya ListView.builder lebih disarankan untuk menggunakan widget yang sama, yang artinya jika kita sudah menggunakan widget ChatItem, maka lebih baik untuk menghindari penambahan widget selain widget ChatItem. Hal ini disebabkan oleh menghindari adanya ambigu pada saat membaca program, dan adanya perbedaan performa pada saat widget tersebut ditampilkan kelayar.

Secara sederhana penggunaan kode listview.builder sebagai berikut:

```
class MyPage extends StatelessWidget {  
  MyPage({Key? key}) : super(key: key);  
  
  final data = [  
    ChatItem(),  
    ChatItem(),  
  ]  
}
```

```

    ChatItem(),
    ChatItem(),
    ChatItem(),
    ChatItem(),
    ChatItem(),
    ChatItem(),
    ChatItem(),
    ChatItem(),
    ChatItem(),
    ChatItem(),
    ChatItem(),
    ChatItem(),
    ChatItem(),
    ChatItem(),
    ChatItem(),
    ChatItem(),
    ChatItem(),
    ];

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('latihan composing'),
    ),
    body: ListView.builder(
      itemCount: data.length,
      itemBuilder: (context, index) {
        var item = data[index];
        return item;
      },
    )
  );
}
}

```

Pada kode diatas kita menggunakan variabel data sebagai media penampung anak dari **ListView**, namun pada kasus sebenarnya variabel data ini bisa jadi kita peroleh dari jaringan atau database dan lain sebagainya.

ListView.separated

Implementasi dari **ListView.separated** hampir sama dengan **ListView.builder** yaitu untuk list yang bersifat dinamis (jumlah list item mengikuti dari jumlah data), bedanya adalah **ListView.separated** akan memberikan widget sisipan diantara item dari list tersebut. **ListView.separated** menggunakan 3 properti utama, dan 2 diantaranya sama dengan **ListView.builder**:

- itemCount (jumlah list item),
- itemBuilder (untuk membangun tampilan dari list item).
- separatorBuilder, untuk membuat widget pembatas.

Sama seperti ListView.builder, ListView.separated juga lebih disarankan untuk menggunakan widget yang sama.

```
class MyPage extends StatelessWidget {
  MyPage({Key? key}) : super(key: key);

  final data = [
    "Januari",
    "Februari",
    "Maret",
    "April",
    "Mei",
    "Juni",
    "Juli",
    "Agustus",
    "September",
    "Oktober",
    "November",
    "Desember"
  ];

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('latihan composing'),
      ),
      body: ListView.separated(
        separatorBuilder: (context, index) {
          return Divider();
        },
        itemCount: data.length,
        itemBuilder: (context, index) {
          return ChatItem();
        },
      )
    );
  }
}
```

Pada kode diatas kita menggunakan variabel data sebagai media penampung anak dari **ListView**, namun pada kasus sebenarnya variabel data ini bisa jadi kita peroleh dari jaringan atau database dan lain sebagainya. Maka jika kode diatas dijalankan, maka hasil yang akan kita peroleh adalah seperti gambar berikut ini:



Seperti yang terlihat pada gambar diatas, properti **separatorBuilder** diisi dengan widget Divider. Divider adalah widget yang berfungsi untuk memberikan garis secara horizontal. Namun properti separatorBuilder ini tidak harus menggunakan widget Divider sebagai pembatasnya, kita tetap bisa menggunakan widget apapun, sebagai contoh:

```
class MyPage extends StatelessWidget {
  MyPage({Key? key}) : super(key: key);

  final data = [
    "Januari",
    "Februari",
    "Maret",
    "April",
    "Mei",
    "Juni",
    "Juli",
```

```

    "Agustus",
    "September",
    "Oktober",
    "November",
    "Desember"
  ];

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('latihan composing'),
      ),
      body: ListView.separated(
        separatorBuilder: (context, index) {
          return Container(
            decoration: BoxDecoration(
              border: Border.all(
                color: Colors.green
              )
            ),
            child: Text(
              'ini adalah pembatas'
            ),
          );
        },
        itemCount: data.length,
        itemBuilder: (context, index) {
          return ChatItem();
        },
      )
    );
  }
}

```

Pada kode program diatas kita bisa menggunakan widget Container yang di dalamnya ada widget Text sebagai pembatas dari masing-masing anak dari ListView. Dan hasil dari kode program diatas akan terlihat seperti pada gambar dibawah ini:

latihan composing



Bayu Angga Wijaya
✓ okee

yesterday

ini adalah pembatas



Bayu Angga Wijaya
✓ okee

yesterday

ini adalah pembatas



Bayu Angga Wijaya
✓ okee

yesterday

ini adalah pembatas



Bayu Angga Wijaya
✓ okee

yesterday

ini adalah pembatas



Bayu Angga Wijaya
✓ okee

yesterday

ini adalah pembatas

Passing Parameter ke Widget

Passing parameter pada widget anak dari widget ListView dilakukan agar masing-masing item bisa memiliki konten yang berbeda, seperti pada contoh sebelumnya kita masing menggunakan konten yang sama di masing-masing widget. Untuk bisa melakukan passing data ke widget item (pada kasus ini ChatItem), maka kita harus membuat widget tersebut bisa menerima properti tambahan. Dimana data yang kita kirim ke widget ChatItem adalah masing-masing data yang ada didalam variable **data** yaitu nama-nama bulan dengan tipe data String.

Lakukan modifikasi pada widget ChatItem sehingga kode berikut ini:

```
import 'package:flutter/material.dart';  
  
class ChatItem extends StatelessWidget {
```



```

final String bulan;
const ChatItem({Key? key, required this.bulan}) : super(key:
key);

@override
Widget build(BuildContext context) {
  return Container(
    margin: EdgeInsets.all(8),
    padding: EdgeInsets.symmetric(horizontal: 8),
    decoration: BoxDecoration(
      color: Colors.lightBlue,
      borderRadius: BorderRadius.only(
        topLeft: Radius.circular(20),
        bottomRight: Radius.circular(20)
      ),
      border: Border.all(
        color: Colors.red,
        width: 2
      )
    ),
    child: Row(
      children: [
        const CircleAvatar(
          backgroundColor: Colors.redAccent,
          child: FlutterLogo(),
        ),
        Expanded(
          child: Padding(
            padding: const EdgeInsets.all(8.0),
            child: Column(
              mainAxisAlignment: MainAxisAlignment.center,
              crossAxisAlignment: CrossAxisAlignment.start,
              children: [
                const Text('Bayu Angga Wijaya'),
                Row(
                  children: const [
                    Icon(Icons.check),
                    Text('okee')
                  ],
                )
              ],
            ),
          ),
        ],
      ),
    ),
  ),
)

```

```

        )
    ),
    Text(bulan)
],
),
);
}
}

```

Sampai disini, kita sudah berhasil membuat widget ChatItem bisa menerima kiriman data dari ListView, maka langkah selanjutnya adalah mengirim data dari ListView yang akan dilakukan di dalam properti itemBuilder, seperti yang terlihat pada kode, itemBuilder memiliki 2 output yaitu:

- context: merujuk pada konteks dari ListView itu sendiri
- Index: urutan dari data, dimulai dengan index 0

Maka dari itu pada bagian ListView, kita hanya perlu mengubah isi dari properti itemBuilder sehingga terlihat seperti kode program berikut:

```

itemBuilder: (context, index) {
    var bulan = data[index];
    return ChatItem(
        bulan: bulan,
    );
},

```

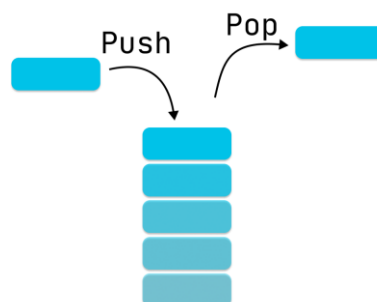
Dan program di jalankan, maka hasil yang kita peroleh adalah seperti pada gambar berikut ini:



7.11 Navigation dan Routing

Navigation and Routing adalah beberapa konsep inti dari semua aplikasi seluler, yang memungkinkan pengguna berpindah di antara halaman yang berbeda. Kita tahu bahwa setiap aplikasi seluler berisi beberapa layar untuk menampilkan berbagai jenis informasi. Misalnya, sebuah aplikasi dapat memiliki layar yang berisi berbagai produk. Ketika pengguna mengklik produk itu, segera akan muncul informasi rinci tentang produk itu. Di Flutter, layar dan halaman dikenal sebagai rute, dan rute ini hanyalah sebuah widget. Di Android, rute mirip dengan Activity, sedangkan di iOS setara dengan ViewController.

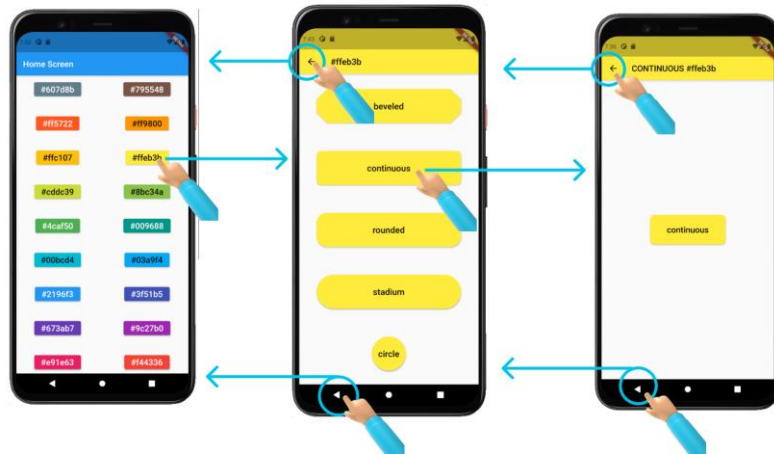
Flutter menyediakan kelas routing dasar `MaterialPageRoute` yang bekerja seperti tumpukan layar (stack), ia menggunakan prinsip LIFO (Last-In, First-Out). Seperti yang terlihat pada gambar berikut:



Ada dua method yang dapat digunakan pada Navigator widget yaitu :

- `Navigator.push()`: Metode push digunakan untuk menambahkan rute lain ke atas tumpukan screen (stack) saat ini. Halaman baru ditampilkan di atas halaman sebelumnya.
- `Navigator.pop()`: Metode pop menghapus rute paling atas dari tumpukan. Ini

menampilkan halaman sebelumnya kepada pengguna. Proses Navigator.push() bisa dilakukan dengan event dari pengguna contoh ketika tombol ditekan, sedangkan untuk proses Navigator.pop() bisa dilakukan dengan cara pengguna menekan tombol, pengguna menekan tombol navigasi pada AppBar atau bisa juga dengan menggunakan tombol fisik back pada ponsel itu sendiri.



Sebelum memulai proses routing, kita harus membuat terlebih dahulu halaman atau page yang nantinya akan menjadi objek perpindahan halaman, pada kasus ini, kita akan menggunakan 2 halaman saja, maka terlebih dahulu modifikasi file **main.dart** seperti pada potongan kode berikut ini:

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

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MainPage(),
    );
  }
}

class MainPage extends StatelessWidget {
```

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('latihan navigasi'),
    ),
    body: Center(
      child: ElevatedButton(
        child: Text('ke halaman setting'),
        onPressed: () {
          },
        ),
      ),
    );
}
```

File **main.dart** berisi halaman MainPage yang akan menjadi halaman utama kita, yang artinya saat aplikasi dimulai maka MainPage adalah halaman yang pertama kali ditampilkan. Seperti yang terlihat pada kode diatas, kita telah mendeklarasikan widget ElevateButton dengan event onPressed masih kosong, yang nantinya akan kita isi dengan event untuk berpindah halaman.

Selanjutnya adalah menambahkan halaman kedua, dengan cara tambahkan 1 file dart dengan nama **setting_page.dart** dan lengkapi halaman dengan widget yang akan mendukung fungsionalitasnya seperti pada berikut ini:

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

class SecondPage extends StatelessWidget {
  const SecondPage({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('second page'),
      ),
      body: Center(
        child: ElevatedButton(
          child: Text('back to main page'),
          onPressed: () {},
        ),
      ),
    );
  }
}
```

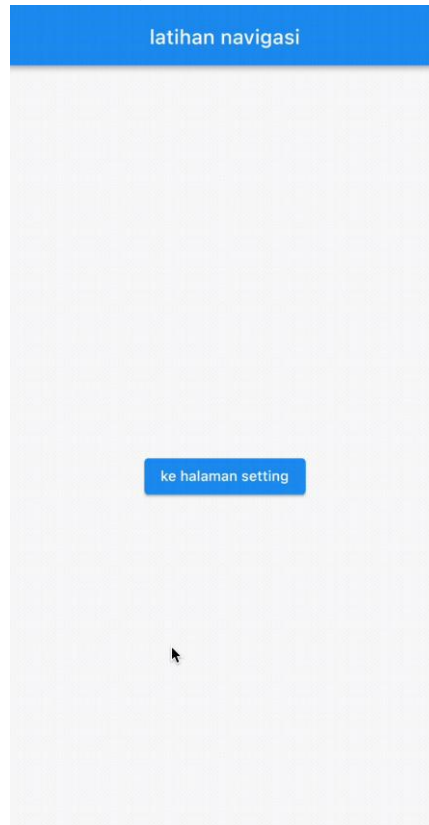
Navigator.push()

Metode Navigator.push() digunakan untuk menavigasi/beralih ke rute/halaman/layar baru. Di sini, metode push() menambahkan halaman/rute pada tumpukan dan kemudian mengelolanya dengan menggunakan Navigator. Sekali lagi kami menggunakan kelas MaterialPageRoute yang memungkinkan transisi antara rute menggunakan animasi khusus platform. Kode di bawah ini menjelaskan penggunaan metode Navigator.push().

```
ElevatedButton(
  child: Text('ke halaman setting'),
  onPressed: () {
```

```
Navigator.push(  
  context,  
  MaterialPageRoute(builder: (context) => SecondPage())  
);  
},  
),
```

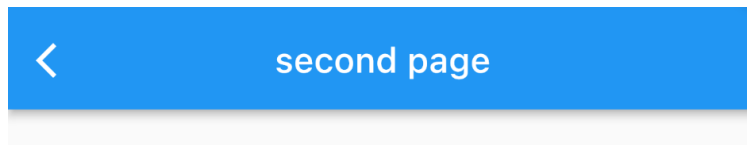
Seperti yang terlihat pada kode diatas, widget ElevatedButton yang ada pada halaman MainPage diisi dengan event Navigator.push untuk melakukan perpindahan ke halaman SecondPage.



Navigator.pop()

Navigator.pop() berfungsi untuk menutup rute SettingPage dan kembali ke MainPage. Metode pop() juga akan memungkinkan kita untuk menghapus rute saat ini dari tumpukan, yang dikelola oleh Navigator.

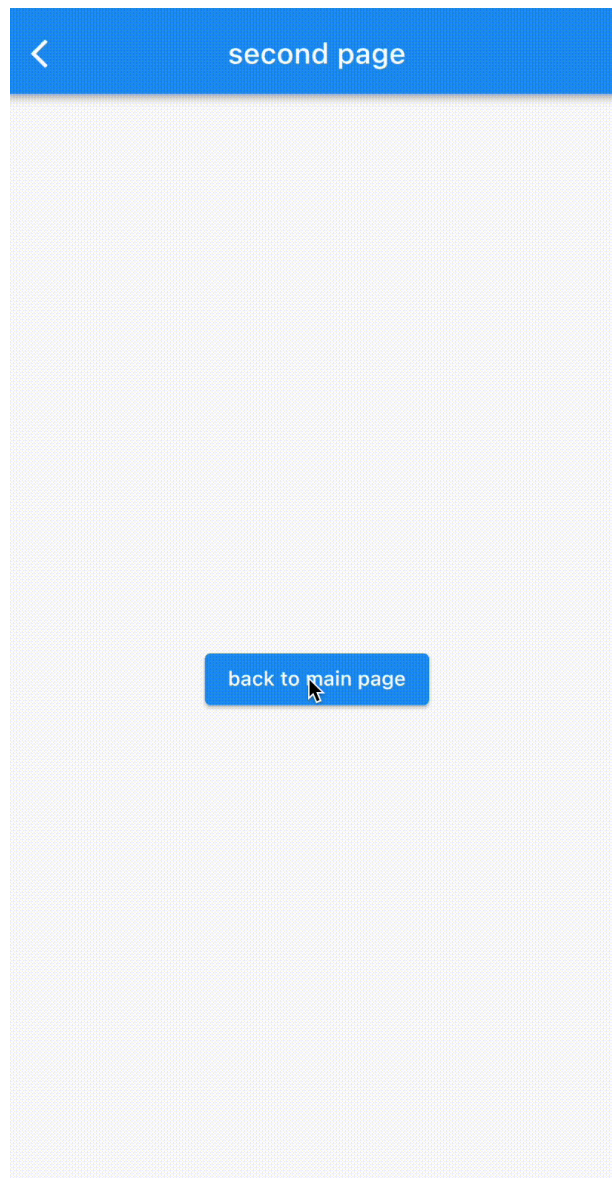
Jika halaman yang kita tuju memiliki widget AppBar yang dideklarasikan pada widget Scaffold, maka secara otomatis, flutter akan menambahkan tombol back pada AppBar tersebut, hal ini disebabkan karena flutter mendeteksi bahwa ada halaman lain di dalam tumpukan atau sederhananya ada halaman lain sebelum halaman SecondPage, yaitu halaman MainPage.



Namun pada beberapa skenario, ada beberapa kasus yang memungkinkan kita butuh widget lain untuk kembali ke halaman sebelumnya, baik itu melalui event yang terjadi, maupun memang menjadi keinginan pengguna itu sendiri, atau bahkan pada kasus misalkan halaman tersebut tidak memiliki AppBar, maka dari itu peran dari `Navigator.pop()` dibutuhkan. Event `Navigator.pop()` akan kita implementasikan ke dalam properti `onPressed` milik `ElevateButton` yang ada di dalam `SecondPage`:

```
ElevatedButton(  
  child: Text('back to main page'),  
  onPressed: () {  
    Navigator.pop(context);  
  },  
)
```

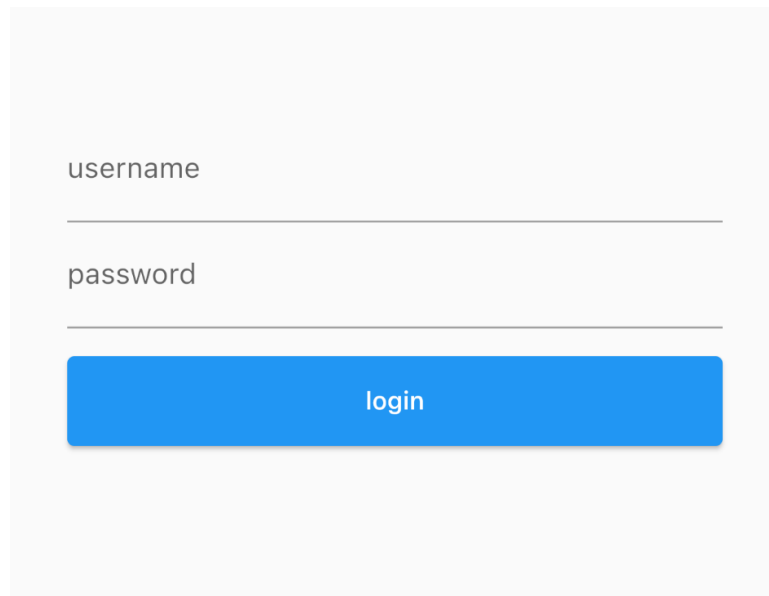
Jika program dijalankan, baik tombol `ElevateButton` maupun tombol back yang berada di dalam AppBar, keduanya akan memiliki kemampuan untuk kembali ke halaman yang sama



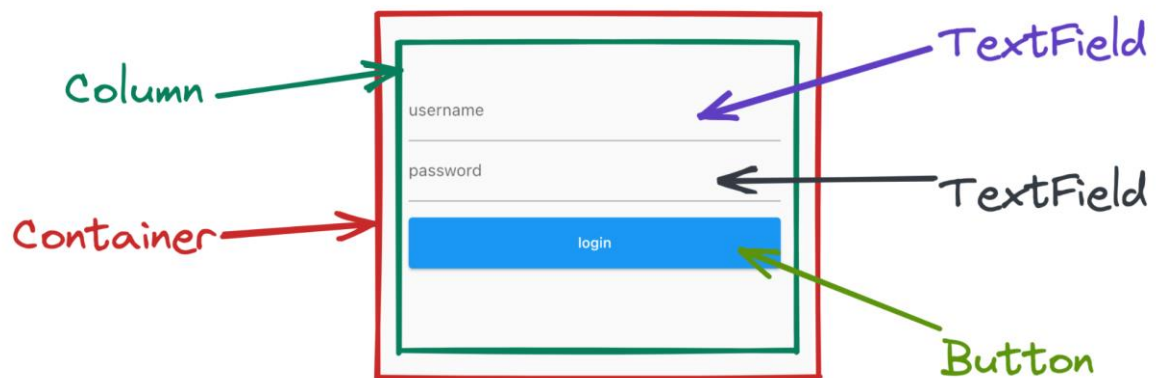
7.12 Input View

Input View adalah view yang berfungsi untuk menampung masukan dari pengguna, masukan tersebut bisa dalam bentuk input data seperti saat pengguna mengetik suatu teks, bisa juga masukan dari pengguna dalam bentuk interaksi seperti pada saat pengguna menekan suatu tombol untuk menyelesaikan suatu proses yang sedang berlangsung.

Gabungan dari beberapa input view disebut dengan istilah Form, dimana form ini akan menampung beberapa widget input yang memiliki tugas dan tujuan yang sama, untuk studi kasus kali ini, kita akan membuat form login.



Seperti biasa, sebelum melakukan proses coding, ada baiknya jika terlebih dahulu kita menganalisis widget-widget apa saja yang akan kita gunakan. Maka secara garis besar, widget-widget yang nantinya akan kita gunakan adalah:



Susunan Widget

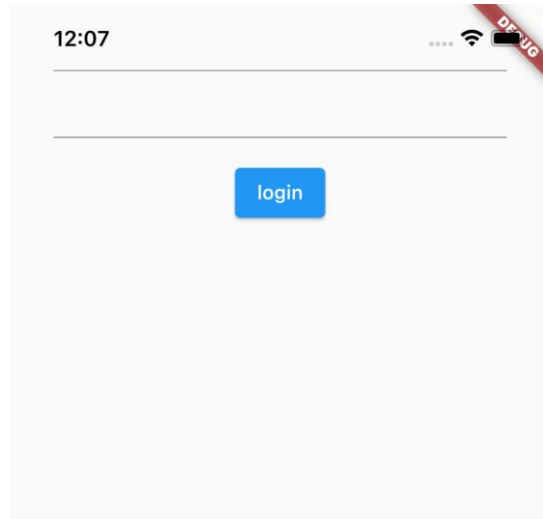
Dengan adanya analisis widget diatas, maka kita akan bisa langsung membuat susunan widget dalam bentuk program, sebelum nantinya dilakukan proses modifikasi properti dari masing-masing widget agar hasil akhir dari program bisa sesuai dengan keinginan, maka dari itu susunan dari widget-widget diatas adalah seperti pada kode program berikut ini:

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

class LoginPage extends StatelessWidget {
  const LoginPage({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Padding(
        padding: const EdgeInsets.symmetric(horizontal: 32),
        child: Column(
          children: [
            TextField(),
            TextField(),
            SizedBox(height: 16,),
            ElevatedButton(
              onPressed: () {},
              child: Text('login')
            )
          ],
        ),
      ),
    );
  }
}
```

Jika program diatas dijalankan, maka hasilnya masih jauh dari desain yang kita inginkan, seperti yang terlihat, dimana keseluruhan widget akan menumpuk pada bagian atas halaman seperti yang terlihat pada gambar berikut ini:

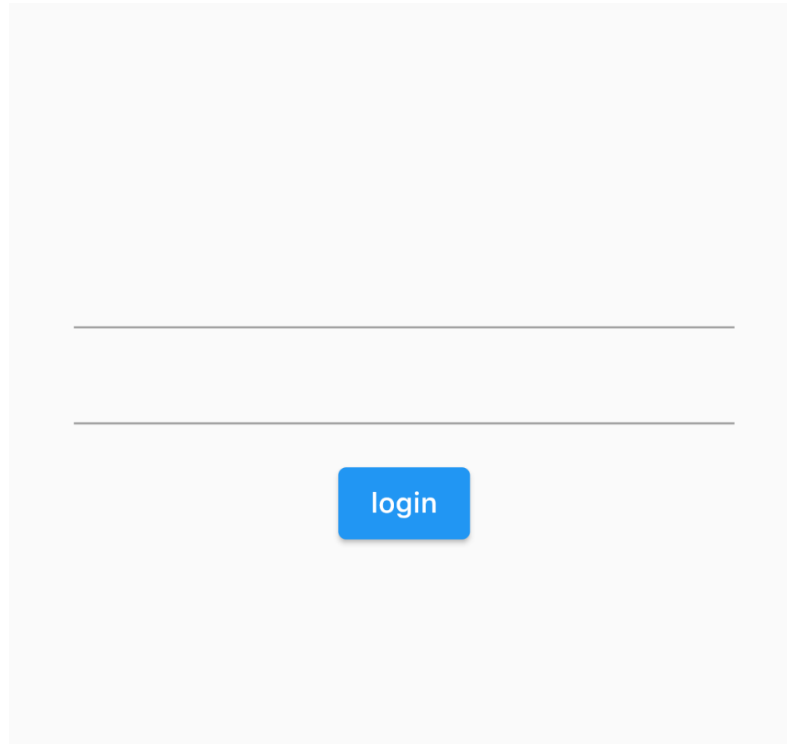


Penyesuaian yang pertama yang bisa kita lakukan adalah pada widget Column, yang tugasnya untuk membuat semua widget menjadi ke tengah layar menggunakan properti **mainAxisAlignment** yang secara default menggunakan value **MainAxisAlignment.start** maka kita harus ubah menjadi **MainAxisAlignment.center**

```
class LoginPage extends StatelessWidget {
  const LoginPage({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Padding(
        padding: const EdgeInsets.symmetric(horizontal: 32),
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            TextField(),
            TextField(),
            SizedBox(height: 16,),
            ElevatedButton(
              onPressed: () {},
              child: Text('login')
            )
          ],
        ),
      ),
    );
  }
}
```

Maka hasilnya akan menjadi seperti gambar berikut



Penyesuaian berikutnya adalah pada widget `TextField`, adapun kebutuhan dari desain yang kita inginkan adalah:

- **Adanya label pada masing-masing `TextField` (username dan password)**

Label pada widget `TextField` bisa diperoleh dengan menambahkan properti `decoration` yang diisi dengan class **`InputDecoration`**.

selanjutnya pada class **`InputDecoration`**, kita bisa memberikan 2 properti tambahan sebagai penyesuaian, yaitu

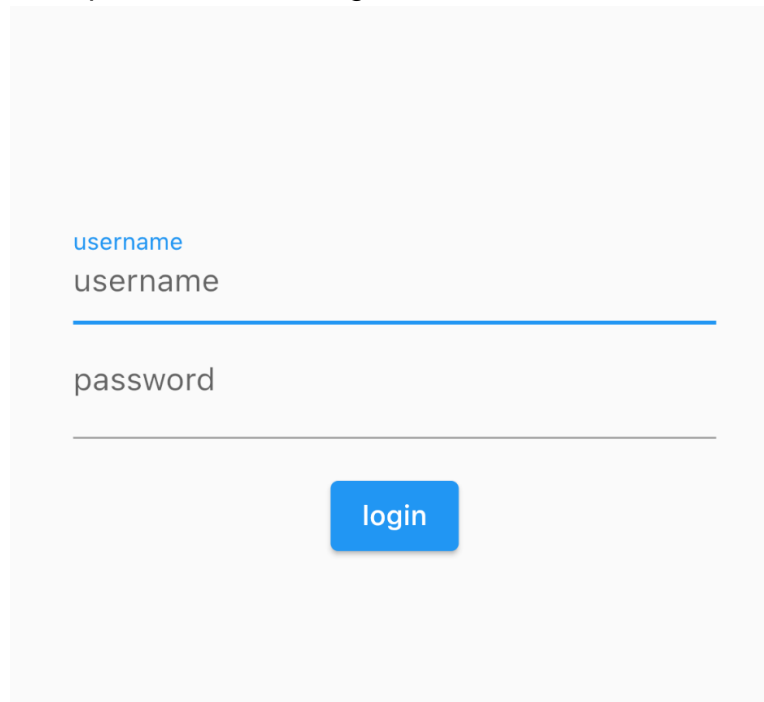
1. **label** adalah properti yang berfungsi untuk memberikan label tulisan kecil pada bagian atas `TextField` pada saat user sedang aktif mengetik.
2. **hint** adalah properti yang bertujuan untuk memberikan label yang muncul ketika `TextField` belum aktif digunakan, dan sebelum pengguna mulai mengetik.

Dengan penyesuaian kedua properti tersebut, kode `TextField` terlihat seperti berikut:

```
TextField(  
  decoration: InputDecoration(  
    label: Text('username'),  
    hintText: 'username'  
  ),  
)  
TextField(  
  decoration: InputDecoration(  
    label: Text('password')  
  ),  
)
```

```
),
```

Maka kita akan memperoleh hasil sebagai berikut:

A screenshot of a login form. It features two text input fields: the top one is labeled 'username' and the bottom one is labeled 'password'. Below the password field is a blue button with the text 'login' in white. The form is set against a light gray background.

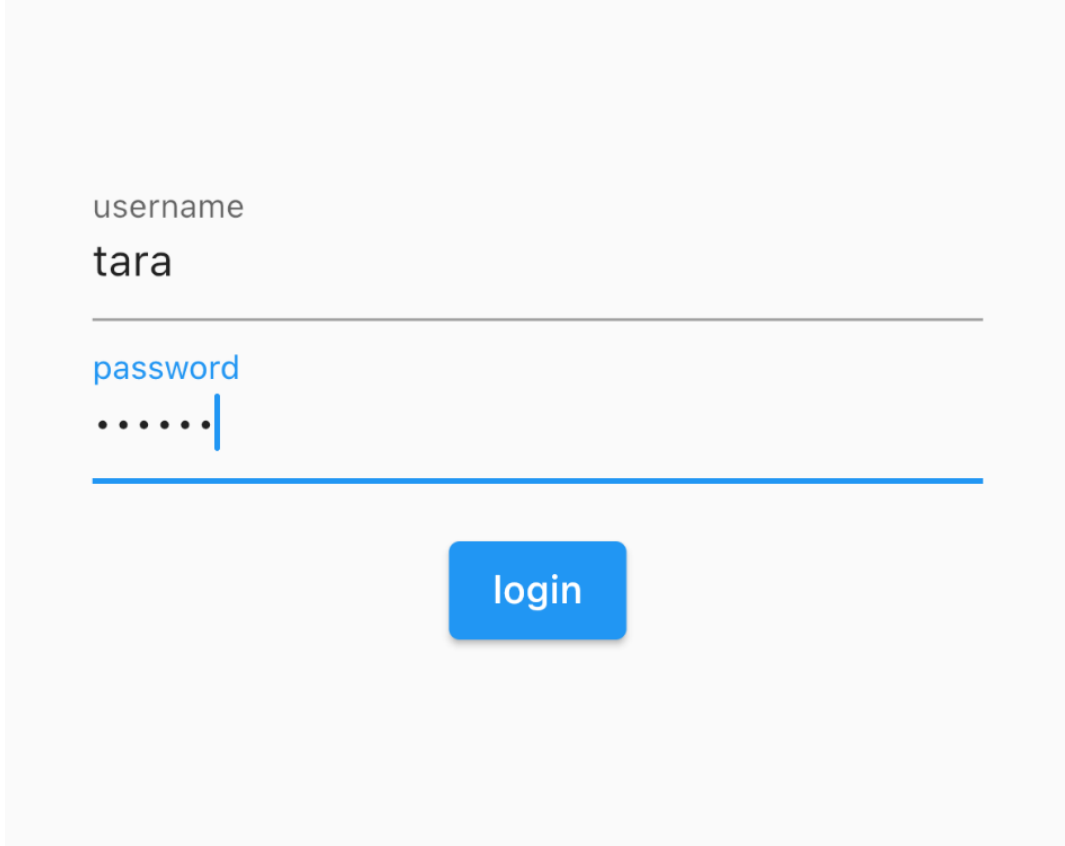
- **Adanya Sensor pada saat TextField password diisi**

Untuk memberikan efek sensor pada TextField yang berperan sebagai input dari password / kata sandi atau informasi yang bersifat rahasia lainnya, kita bisa menggunakan properti `obscureText`, yang secara default memiliki nilai `false`, sehingga perlu dilakukan perubahan menjadi `true`, seperti yang terlihat pada potongan kode program berikut ini:

```
TextField(  
  decoration: InputDecoration(  
    label: Text('username'),  
    hintText: 'username'  
  ),  
)  
TextField(  
  decoration: InputDecoration(  
    label: Text('password')  
  ),  
  obscureText: true,  
)
```

Seperti yang terlihat pada potongan kode diatas, kita hanya menggunakan properti `obscureText` pada password, karena kita hanya membutuhkan

kerahasiaan input pada password, input pada username tidak termasuk. Maka jika program dijalankan, hasilnya akan terlihat seperti pada gambar berikut ini:



A screenshot of a login form on a light gray background. The form consists of two input fields and a button. The first field is labeled 'username' and contains the text 'tara'. The second field is labeled 'password' and contains six dots followed by a vertical cursor line. Below the password field is a blue button with the text 'login' in white.

Sampai sejauh ini, form yang kita kerjakan sudah mendekati desain awal yang kita inginkan, penyesuaian terakhir yang perlu dilakukan adalah tombol login harus memiliki panjang yang sama dengan widget lainnya (TextField username dan password). Hal ini dikarenakan widget ElevatedButton secara default mengatur lebarnya sesuai dengan isi dari widget yang ada didalamnya (widget Text('login'))

Maka untuk memperoleh desain yang kita inginkan, kita harus mengubah properti style milih widget ElevatedButton yang akan diisi dengan ElevatedButton.styleFrom:

```
ElevatedButton(  
  style: ElevatedButton.styleFrom(  
    minimumSize: const Size.fromHeight(50),  
  ),  
  onPressed: () {},  
  child: Text('login')  
)
```

Seperti yang terlihat pada baris kode diatas, kita menambahkan properti minimumSize: const Size.fromHeight(50), yang artinya kita mengatur ukuran dari ElevatedButton adalah memiliki tinggi minimal 50 beserta lebar dengan value infinity atau tak terhingga sampai ElevatedButton tersebut mencapai pinggiran layar kita. Sehingga jika kode dijalankan, kita akan memperoleh hasil yang sama persis dengan desain yang kita inginkan sebelumnya.

username
tara

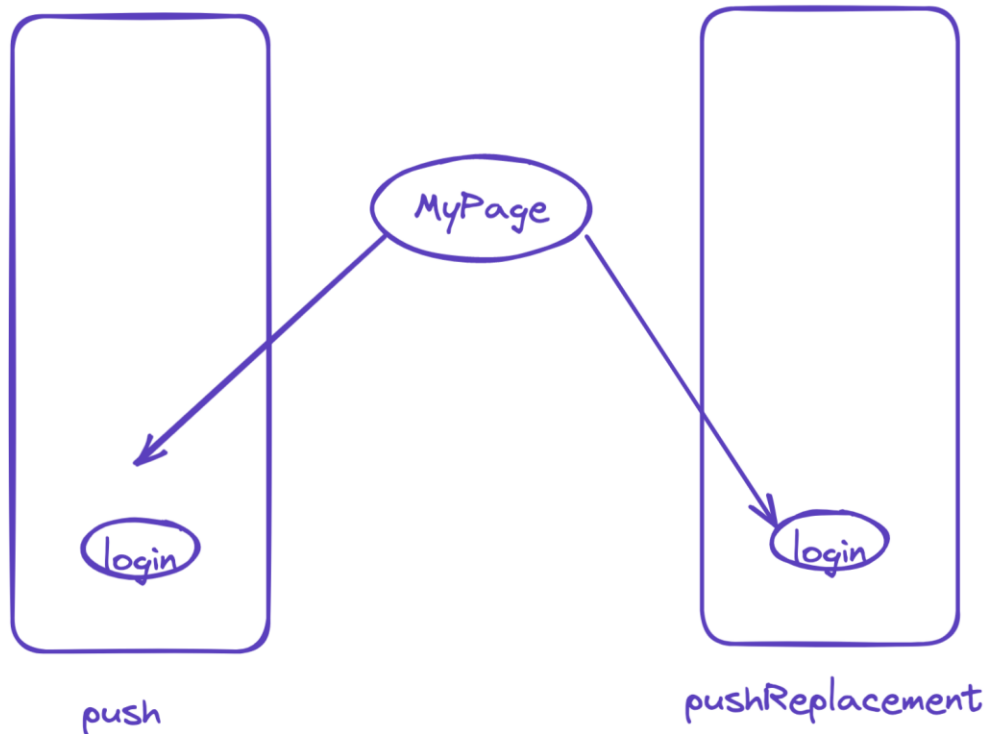
password
.....|

login

Pada materi sebelumnya, kita sudah membahas mengenai dasar dari navigasi pada flutter, dimana navigasi adalah fitur flutter yang memungkinkan aplikasi yang kita bangun untuk berpindah dari suatu halaman ke halaman lain. Beberapa fitur dari navigasi sudah kita bahas yaitu implementasi dari metode push dan pop pada navigasi. Maka, untuk tahap berikutnya kita akan membahas beberapa metode dari navigasi yang umum digunakan dimana metode-metode ini pada dasarnya hanya digunakan untuk menyelesaikan kasus tertentu saja, sehingga bisa disimpulkan bahwa penggunaannya akan sangat minim.

pushReplacement

pushReplacement digunakan untuk berpindah halaman tapi menutup rute saat ini, dimana push hanya akan menambahkan rute baru diatas rute saat ini. Fitur ini akan kita gunakan pada masalah yang sedang ada pada project kita, dimana saat kita berhasil login setelah berpindah pada halaman berikutnya halaman tersebut memiliki tombol back pada AppBar yang ada pada halaman tersebut. Sehingga perbedaan antara push dan pushReplacement bisa dilihat seperti pada gambar dibawah ini:



Untuk implementasinya pada kode project kita, yang perlu dilakukan hanya mengubah dari method push menjadi pushReplacement:

```
Navigator.pushReplacement(
    context, MaterialPageRoute(
        builder: (context) => MyPage()
    )
);
```

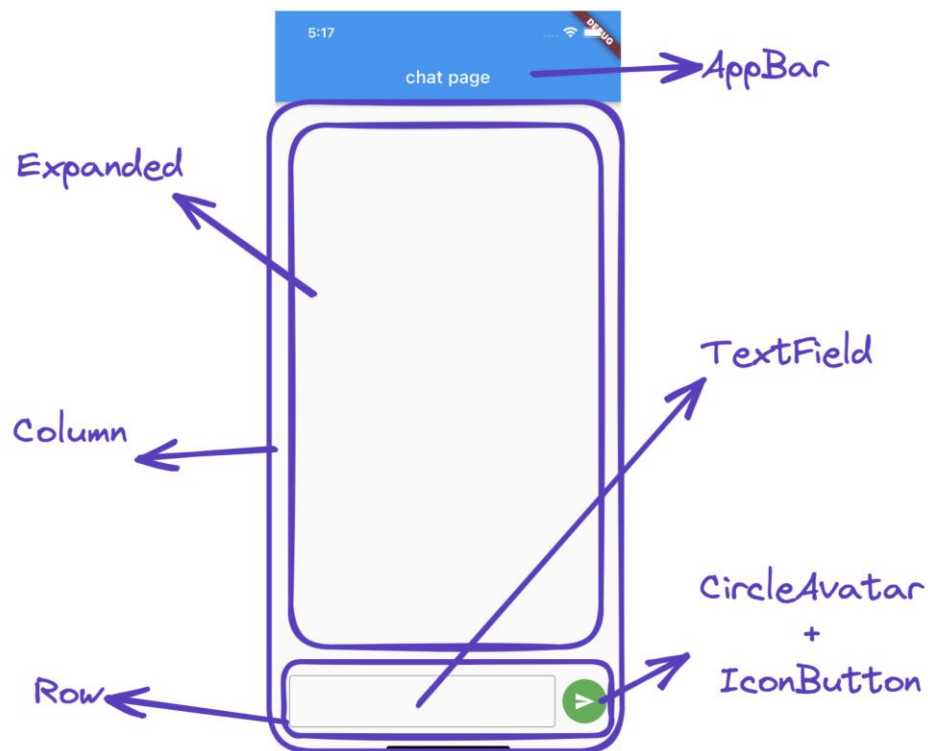
Push dengan Argument (passing data)

Push dengan argument / passing pada akan dibutuhkan ketika kita ingin data yang kita pilih bisa ditampilkan pada halaman tujuan, hal ini bertujuan untuk nantinya jika aplikasi yang aplikasi yang kita bangun ingin menampilkan detail rinci untuk data yang kita pilih, atau adanya suatu fitur khusus untuk data yang kita pilih. Contoh implementasi dari fitur ini seperti aplikasi Gmail, dimana pada halaman awal aplikasi hanya akan menampilkan list atau daftar dari judul dan pengirim email, selanjutnya ketika kita klik salah satu item dari daftar email tersebut, aplikasi akan mengarahkan kita ke halaman baru dimana pada halaman tersebut kita bisa melihat keseluruhan detail dari email tersebut, mulai dari jam pengiriman, isi, attachment dll.

Sebagai studi kasus pada project kita, sebelumnya kita sudah memiliki daftar percakapan dengan konten pengirim dan pesan terakhir. Maka untuk fitur selanjutnya, saat salah satu item dari percakapan ini diklik, maka aplikasi akan dialihkan ke halaman baru yang akan menampilkan detail dari percakapan itu sendiri, termasuk semua pesan yang ada didalamnya nantinya.

Langkah pertama yang akan kita lakukan adalah membuat terlebih dahulu halaman

yang akan kita tuju nantinya, adapun anatomi dari halaman tujuan adalah seperti gambar berikut ini:



Halaman tersebut akan dibuat menggunakan stateless widget dengan nama file `chat_page.dart`, sebagai catatan widget `Expanded` hanya akan diisi dengan widget `ListView` dengan children kosong, hal ini dikarenakan materi ini hanya akan berfokus pada pengiriman data dari satu halaman ke halaman lain, sedangkan konten dari widget `Expanded` dan `ListView` tersebut akan kita bahas pada pertemuan berikutnya yang akan membahas tentang basic state management.

File `chat_page.dart`

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

class ChatPage extends StatelessWidget {
  const ChatPage({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('chat page'),
      ),
      body: SafeArea(
        child: Column(
```

```

children: [
  Expanded(
    child: ListView()
  ),
  Container(
    padding: const EdgeInsets.symmetric(horizontal: 16),
    child: Row(
      children: [
        Expanded(
          child: TextField(
            decoration: InputDecoration(
              border: OutlineInputBorder(),
            ),
          ),
        ),
        SizedBox(width: 8),
        CircleAvatar(
          radius: 25,
          backgroundColor: Colors.green,
          child: IconButton(
            color: Colors.white,
            onPressed: () {},
            icon: Icon(Icons.send)
          ),
        ),
      ],
    ),
  ],
);
}

```

Langkah berikutnya adalah melakukan navigasi dari class ChatItem yang sudah kita buat sebelumnya. Penting untuk diingat bahwa widget ChatItem belum memiliki kemampuan untuk menerima interaksi klik atau tap dari pengguna. Agar sebuah widget bisa menerima interaksi tap atau klik dari pengguna, kita bisa menggunakan 2 opsi widget yang memiliki beberapa fitur berbeda, jadi perlu disesuaikan dengan kebutuhan:

1. **InkWell**: widget yang digunakan untuk menerima interaksi tap/klik dari

pengguna, dimana widget ini memiliki feedback berupa efek ripple pada widget yang kita tap/klik. Contoh dari efek ripple ini bisa dilihat saat kita klik pada pesan whatsapp akan ada sedikit flash putih atau abu-abu sebelum halaman berpindah.

2. **GestureDetector**: widget yang digunakan untuk menerima interaksi tap/klik dari pengguna, namun widget ini tidak memiliki efek feedback seperti ripple pada InkWell, namun widget ini tetap memiliki kelebihan dibandingkan dengan InkWell yaitu kemampuan menerima interaksi selain tap, seperti drag, zoom dll.

Maka bisa disimpulkan bahwa InkWell akan memberikan User Experience tambahan karena memiliki feedback saat pengguna melakukan tap, sedangkan GestureDetector menyediakan interaksi yang lebih bervariasi. Untuk kebutuhan project ini kita akan menggunakan GestureDetector untuk menerima interaksi dari pengguna.

```
class ChatItem extends StatelessWidget {
  final ChatData chatData;

  const ChatItem({Key? key, required this.chatData}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Container(...); // Container
  }
}
```

Widget ChatItem yang kita miliki saat ini terlihat seperti pada gambar diatas, dimana keseluruhan isi dari widget dibungkus kedalam widget Container. Sehingga untuk menerapkan interaksi tap, widget Container tersebut akan kita bungkus dengan salah satu widget diatas, sebagai contoh kita akan menggunakan widget GestureDetector dimana widget ini memiliki properti onTap yang berfungsi untuk memproses apa yang akan kita lakukan sebagai respon dari tap tersebut.

```
class ChatItem extends StatelessWidget {
  final ChatData chatData;

  const ChatItem({Key? key, required this.chatData}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: () {
        // Action
      },
      child: Container(...), // Container
    ); // GestureDetector
  }
}
```

Selanjutnya kita akan melakukan proses dari interaksi tap yang sudah dilakukan,

yaitu berpindah ke halaman ChatPage yang sudah dibuat sebelumnya dengan menggunakan class Navigator dan method push.

```
class ChatItem extends StatelessWidget {
  final ChatData chatData;

  const ChatItem({Key? key, required this.chatData}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: () {
        Navigator.push(
          context,
          MaterialPageRoute(
            builder: (context) => ChatPage()
          ) // MaterialPageRoute
        );
      },
      child: Container(...), // Container
    ); // GestureDetector
  }
}
```

Kita sudah bisa berpindah dari halaman daftar chat ke halaman detail chat, namun belum mengirimkan data apapun. Pengiriman data pada navigasi bisa dilakukan dengan cara yang sama saat kita mengirim ChatData ke dalam ChatItem pada pertemuan sebelumnya. Yaitu dengan memberikan properti tambahan pada widget tujuan. Pada kasus ini kita akan mengirimkan ChatData ke ChatPage, yang artinya kita perlu menambahkan properti ChatData pada widget ChatPage:

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

class ChatPage extends StatelessWidget {
  final ChatData chatData;
  const ChatPage({
    Key? key,
    required this.chatData
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('chat page'),
      ), // AppBar
      body: SafeArea(...), // SafeArea
    ); // Scaffold
  }
}
```

You, 40 minutes ago • Uncommitted changes

Kembali ke widget ChatItem yang saat ini akan berubah menjadi error dikarenakan ChatPage sekarang membutuhkan ChatData, maka yang perlu kita lakukan hanyalah meneruskan chatData yang sudah kita miliki di ChatItem ke dalam ChatPage.

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

import 'main.dart';

class ChatItem extends StatelessWidget {
  final ChatData chatData;

  const ChatItem({Key? key, required this.chatData}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: () {
        Navigator.push(
          context,
          MaterialPageRoute(
            builder: (context) => ChatPage(
              chatData: chatData,
            ) // ChatPage
          ) // MaterialPageRoute
        );
      },
      child: Container(...), // Container
    ); // GestureDetector
  }
}
```

Untuk saat ini kita sudah berhasil mengirim data dari halaman daftar chat ke halaman detail chat, hanya saja data yang sudah kita kirim tersebut belum terlihat secara visual. Untuk hal tersebut kita bisa menampilkan data nama dari pemilik chat yang ada untuk kita tampilkan di AppBar yang ada pada halaman ChatPage:


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

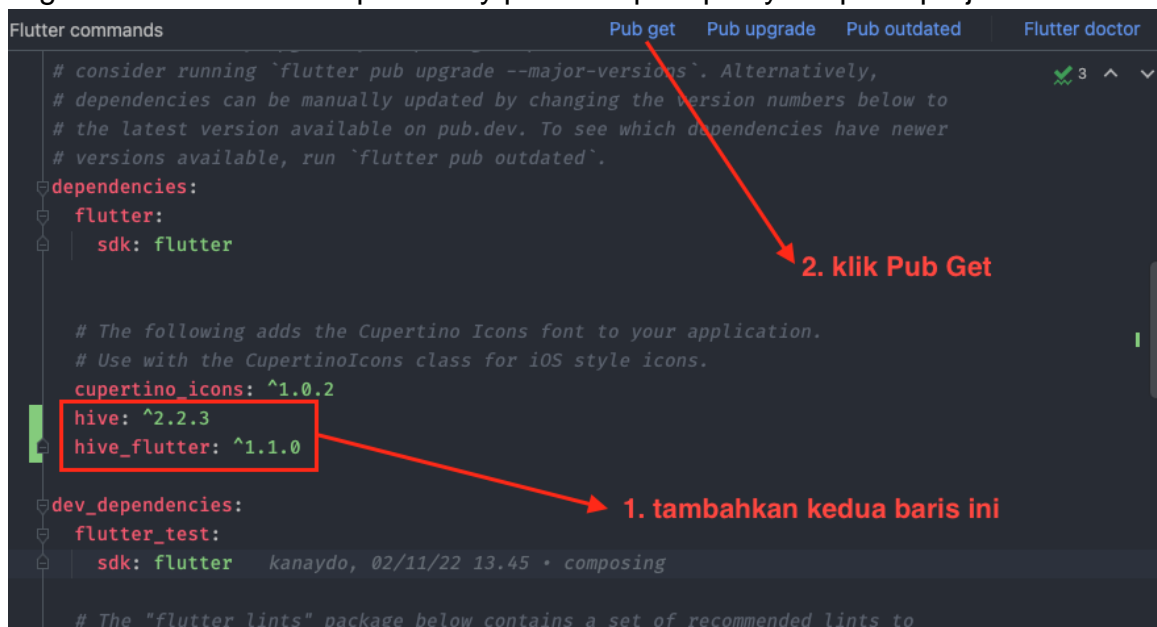
class ChatPage extends StatelessWidget {
  final ChatData chatData;
  const ChatPage({
    Key? key,
    required this.chatData
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(chatData.name),
      ), // AppBar
      body: SafeArea(...), // SafeArea
    ); // Scaffold
  }
}
```

Menyimpan Sesi Login

Sejauh ini kita sudah berhasil membuat fitur login pada aplikasi kita, namun ada satu kendala yaitu setelah kita berhasil login dan aplikasi di restart ulang, maka kita diharuskan untuk login ulang. Hal ini tentunya tidak tepat untuk aplikasi yang berada di pasaran, dimana login hanya akan dibutuhkan sekali saja. Untuk itu kita perlu menyimpan informasi bahwa user sudah login atau belum.

Untuk saat ini flutter sendiri belum memiliki fitur yang bisa menangani masalah ini, sehingga kita akan membutuhkan library pihak ketiga untuk membantu kita menyelesaikan masalah tersebut, yaitu flutter_hive. Langkah pertama adalah dengan menambahkan dependency pada file pubspec.yaml pada project kita:



```
Flutter commands Pub get Pub upgrade Pub outdated Flutter doctor
# consider running `flutter pub upgrade --major-versions`. Alternatively,
# dependencies can be manually updated by changing the version numbers below to
# the latest version available on pub.dev. To see which dependencies have newer
# versions available, run `flutter pub outdated`.
dependencies:
  flutter:
    sdk: flutter

  # The following adds the Cupertino Icons font to your application.
  # Use with the CupertinoIcons class for iOS style icons.
  cupertino_icons: ^1.0.2
  hive: ^2.2.3
  hive_flutter: ^1.1.0

dev_dependencies:
  flutter_test:
    sdk: flutter kanaydo, 02/11/22 13.45 • composing

# The "flutter_lints" package below contains a set of recommended lints to
```

2. klik Pub Get

1. tambahkan kedua baris ini

Selanjutnya adalah melakukan inisialisasi class Hive pada file main.dart di dalam function main(), sebelum kita menjalankan function runApp:

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

import 'package:hive_flutter/hive_flutter.dart';

void main() async {
  await Hive.initFlutter();
  await Hive.openBox('session');

  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
    );
  }
}
```

2. import paket flutter_hive

1. inisialisasi flutter hive

Buat sebuah class di dalam file session.dart yang akan bertugas menampung data dari login pengguna.

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

class Session {
  final box = Hive.box('session');

  void setLogin() {
    box.put('IS_LOGIN', true);
  }

  bool isLogin() {
    return box.get('IS_LOGIN') ?? false;
  }
}
```

You, 2 minutes ago • Uncommitted changes

Pada halaman login page, kita perlu menginisialisasi class Session, dan melakukan setLogin setelah login berhasil sebagai pertanda bahwa user sudah login:

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

    final session = Session(); → 1. inialisasi class Session

    return Scaffold(
      body: Padding(
        padding: const EdgeInsets.symmetric(horizontal: 32),
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            TextField(...), // TextField
            TextField(...), // TextField
            SizedBox(height: 16,),
            ElevatedButton(
              style: ElevatedButton.styleFrom(
                minimumSize: const Size.fromHeight(50)
              ),
              onPressed: () {
                final username = usernameController.text;
                final password = passwordController.text;

                if (username == 'user' && password == '123456') {

                    session.setLogin(); → 2. setLogin jika login berhasil

                    ScaffoldMessenger.of(context).showSnackBar(
                      SnackBar(
                        content: Text('login berhasil'),
                        backgroundColor: Colors.green,
                      ) // SnackBar
                    );
                    Navigator.pushReplacement(
                      context, MaterialPageRoute(
                        builder: (context) => MyPage()
                      ) // MaterialPageRoute
                    );
                } else {
```

Kembali ke file main.dart pada class MyApp, kita akan menginisialisasi class Session yang sudah kita buat, dan membuat ternary operator untuk pengecekan apakah user sudah login atau belum. Jika sudah login maka akan langsung ke MyPage jika tidak maka akan ke LoginPage:

```
class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {

    final session = Session(); → 1. inisialisasi class Session

    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ), // ThemeData
      home: session.isLogin() ? MyPage() : LoginPage(),
    ); // MaterialApp
  }
}
```

7.13 Basic State Management

State management adalah sebuah cara untuk mengatur data / state kita bekerja, bisa juga untuk memisahkan antara logic dan view, dimana logic tersebut juga bisa reusable. Cara kerja state management seperti provide and listen, artinya adalah kamu bisa memasukkan state yang kemungkinan bisa berubah sewaktu waktu, lalu widget yang subscribe (listen) dengan provider yang kita buat akan berubah sesuai dengan state yang berubah.

Setiap bagian dalam flutter terbuat dalam bentuk widget. Terdapat dua widget yang memiliki perlakuan state yang berbeda, yaitu stateful dan stateless widget.

Apa perbedaan keduanya?

1. Stateless merupakan widget yang dimuat secara statis dimana seluruh konfigurasi yang dimuat di dalamnya telah diinisialisasikan sejak awal widget itu dimuat. Artinya, state jenis ini tidak dapat diubah dan tidak akan pernah berubah. Stateless biasa digunakan hanya untuk tampilan saja seperti button, item, box container, dan lain-lain.
2. Stateful merupakan suatu widget yang sifatnya dinamis atau dapat berubah-ubah, kebalikan dari stateless widget. Pada stateful, kamu dapat menggunakan property initState yang berfungsi untuk menginisialisasi state yang akan pertama kali dijalankan. Stateful widget dapat mengubah atau mengupdate tampilan, menambah widget lainnya, mengubah nilai variabel, icon, warna, dan masih banyak lagi.

$$\text{UI} = f(\text{state})$$

The layout on the screen Your build methods The application state

Flutter bersifat deklaratif, artinya flutter membangun user interfacenya dengan merefleksikan setiap perubahan state. State ini lah yang menjadi trigger untuk meredraw tampilan user interface. Ketika terjadi perubahan, UI akan merebuild secara otomatis dari awal. Untuk fungsi ini flutter sendiri sudah memiliki state management bawaan yaitu **setState** yang hanya bisa digunakan pada Statefull widget. setState sendiri bisa digunakan secara sepenuhnya untuk membangun aplikasi, namun pada aplikasi dengan skala besar dan memiliki kompleksitas tinggi seperti:

- Perlunya komunikasi antar widget, setState sendiri untuk saat ini tidak mendukung fitur ini.
- Menjalankan ulang method build, yang mana akan membuat aplikasi begitu berat jika ada banyak widget di dalam screen. Yang mana seharusnya hanya perlu build widget yang mengalami perubahan.
- Akan membuat kesulitan bagi developer untuk memisahkan logic aplikasi dengan UI.

Dari beberapa kelemahan dari setState diatas maka lahirlah state management library yang kebanyakan berfokus pada penyelesaian masalah-masalah diatas. Dimana masing-masing state management library memiliki kelebihan dan kekurangan masing-masing sehingga perlu adanya analisis kebutuhan sebelum memutuskan state management library mana yang akan digunakan dalam pengerjaan sebuah projek.

Jenis-jenis State Management Flutter

State management memang merupakan salah satu topik pembahasan yang sangat kompleks apabila kita ingin memperdalam tentang flutter. Namun, untuk menggunakan state management, kamu dapat memanfaatkan beberapa package state management berikut.

1. Provider (<https://pub.dev/packages/provider>)

Provider merupakan state management yang paling sederhana dan mudah digunakan. Provider menyediakan sebuah teknik mengolah state yang dapat digunakan untuk memmanage data didalam aplikasi. Manfaat menggunakan provider antara lain:

- Mengalokasikan resource menjadi lebih sederhana
- Lazy-loading

- Mengurangi boilerplate saat membuat kelas baru setiap saat
- Support dengan devtool, karena status aplikasi kamu akan terlihat di flutter devtool
- Peningkatan skalabilitas untuk class yang memanfaatkan mekanisme listen yang dibangun secara kompleks.

2. **Riverpod** (<https://pub.dev/packages/riverpod>)

Riverpod mirip dengan provider yang compile-safe untuk digunakan dan teruji (testable). Kamu tidak akan menemukan ProviderNotFoundException atau error dalam menangani proses dalam mengelola state. Karena, selagi kode kamu berhasil di kompilasi, maka aplikasi akan tetap dijalankan.

Riverpod mendukung multiple providers dengan tipe yang sama, proses asynchronous, dan mampu menambahkan provider dari file mana saja.

3. **Redux** (https://pub.dev/packages/flutter_redux)

Bagi kamu yang web developer, mungkin kamu sudah tidak asing dengan yang namanya redux. Redux adalah arsitektur aliran data searah yang memudahkan pengembangan, pemeliharaan, dan pengujian aplikasi. Redux berasal dari javascript yang membuat predictable state container untuk aplikasi.

Berikut ini adalah manfaat ketika kamu menggunakan Redux :

- Sangat scalable dan terstruktur dengan baik.
- Memiliki alur yang jelas yang membuatnya menjadi mudah mengelola bahkan mengembalikan perubahan state, atau debug aplikasi.
- Perubahan state dilakukan dengan fungsi.
- Limited API.

4. **BLoC** (https://pub.dev/packages/flutter_bloc)

BLoC atau Business Logic Component adalah design pattern yang membantu kamu untuk memisahkan presentation dengan business logic. Sehingga komponen pada project terbagi menjadi presentational component, BLoC, dan backend. Pattern ini memperbolehkan developer untuk fokus dalam mengkonversikan event menjadi state. BLoC mengelola state dengan menggunakan pendekatan stream atau reactive. Secara umum, data akan bergerak dari BLoC ke UI, atau sebaliknya dalam bentuk streams.

BLoC sendiri memiliki 3 point utama, yaitu:

- Simple
- Powerful.
- Testable

Mudah untuk dilakukan proses testing.

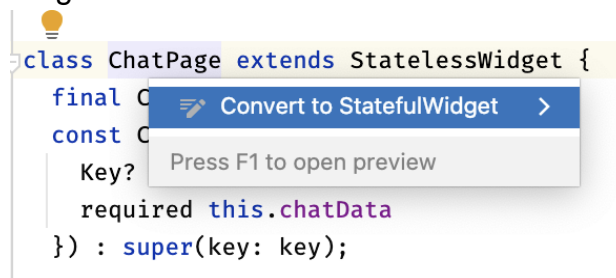
5. **GetX** (<https://pub.dev/packages/get>)

GetX merupakan salah satu pilihan terbaik untuk state management yang memiliki performa tinggi, memiliki injection dependency yang cerdas, serta memiliki manajemen route yang cepat dan praktis. GetX tidak akan memberatkan aplikasi, meskipun memiliki fitur yang banyak, namun masing-

masing fitur berada dalam container terpisah dan akan mulai dijalankan setelah dipakai. Misal, jika kamu hanya menggunakan state management, maka hanya state management lah yang akan di compile, tidak termasuk route dan lainnya.

Implementasi State Management (setState)

State management setState sendiri hanya bisa digunakan pada widget stateful, jika function setState tidak akan ditemukan pada widget stateless, sehingga kita harus mengkonversi widget atau halaman yang kita miliki menjadi statefull dengan cara tekan **alt + enter** pada widget stateless yang akan kita ubah menjadi statefull. Kemudian klik menu Convert to StatefulWidget maka android studio akan secara otomatis mengubah widget tersebut.



Jika sudah berhasil maka widget akan terpisah menjadi 2 class yang terdiri dari widget itu sendiri dan class state dari widget yang kita miliki.

```
class ChatPage extends StatefulWidget {  
  final ChatData chatData;  
  const ChatPage({  
    Key? key,  
    required this.chatData  
  }) : super(key: key);  
  
  @override  
  State<ChatPage> createState() => _ChatPageState();  
}  
  
class _ChatPageState extends State<ChatPage> {  
  @override  
  Widget build(BuildContext context) {...}  
}
```

Langkah selanjutnya adalah mendeklarasikan variable yang akan mengalami perubahan, deklarasi variable ini harus dilakukan pada class state bukan class widget (pada studi kasus ini pada class _ChatPageState). Kita akan mendeklarasikan variable message dengan isi pesan kosong.


```

class _ChatPageState extends State<ChatPage> {
  String message = '';
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.chatData.name),
      ), // AppBar

```

Jika variabel sudah dideklarasikan, maka kita perlu menugaskan widget untuk menampung value atau nilai dari variabel tersebut untuk ditampilkan pada screen, karna tipe data kita adalah string maka widget yang akan bertanggung jawab untuk menampilkan isi dari variabel tersebut adalah widget Text:

```

body: SafeArea(
  child: Column(
    children: [
      Expanded(
        child: Center(
          child: Text(message)
        ) // Center
      ), // Expanded You, Moments ago • Uncommitted changes
      Container(
        padding: const EdgeInsets.symmetric(horizontal: 16),

```

initState [optional]

initState adalah fungsi pada flutter yang berfungsi untuk melakukan suatu tugas pada awal atau saat widget tersebut ditampilkan kelayar. Implementasi dari initState ini sangat luas salah satu contoh real-nya adalah, saat kita membuka sebuah aplikasi maka aplikasi tersebut akan loading dan mengambil data terbaru. Hal tersebut terjadi karena fungsi atau tugas mengambil data baru dilakukan saat halaman tersebut dimulai atau dibuka. Untuk contoh kasus ini kita akan menampilkan pesan 'belum ada pesan diterima' saat aplikasi pertama dibuka.

```

@override
void initState() {
  message = 'belum ada pesan diterima';
  super.initState();
}


```

Untuk menangkap pesan yang diketik oleh pengguna, seperti yang sudah dibahas pada topik sebelumnya, kita akan menggunakan class TextEditingController yang

akan bertugas sebagai controller penampung isi dari TextField:

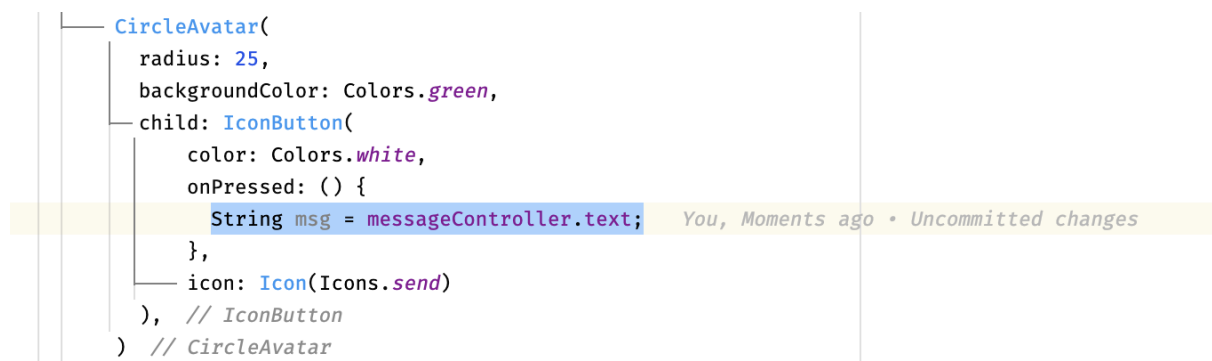
```
class _ChatPageState extends State<ChatPage> {  
  final messageController = TextEditingController();  
  
  String message = '';  
  
  @override  
  void initState() {  
    message = 'belum ada pesan diterima';  
    super.initState();  
  }  
}
```

Kemudian daftarkan controller tersebut ke widget TextField yang sudah kita buat sebelumnya.



```
Container(  
  padding: const EdgeInsets.symmetric(horizontal: 16),  
  child: Row(  
    children: [  
      Expanded(  
        child: TextField(  
          controller: messageController,  
          decoration: InputDecoration(  
            border: OutlineInputBorder(),  
          ), // InputDecoration  
        ), // TextField  
      ), // Expanded  
    ],  
  ),  
  SizedBox(width: 8,),  
)
```

Tahap selanjutnya adalah menangkap isi dari TextField dengan bantuan controller pada saat tombol send ditekan oleh pengguna, untuk melakukan hal tersebut maka kita akan menambahkan property onPressed pada IconButton.



```
CircleAvatar(  
  radius: 25,  
  backgroundColor: Colors.green,  
  child: IconButton(  
    color: Colors.white,  
    onPressed: () {  
      String msg = messageController.text;  
    },  
    icon: Icon(Icons.send)  
  ), // IconButton  
) // CircleAvatar
```

Tahap terakhir adalah menjalankan method setState untuk memberitahu widget kita bahwa ada perubahan pada variabel message, dan flutter akan menjalankan ulang method build sehingga screen kita akan menampilkan data terbaru.

```
onPressed: () {  
  String msg = messageController.text;  
  
  setState(() {  
    message = msg;  
  });  
  
},  
icon: Icon(Icons.send) You, A minute ago • Uncommitte
```

Complete code:

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

class ChatPage extends StatefulWidget {
  final ChatData chatData;
  const ChatPage({
    Key? key,
    required this.chatData
  }) : super(key: key);

  @override
  State<ChatPage> createState() => _ChatPageState();
}

class _ChatPageState extends State<ChatPage>{
  final messageController = TextEditingController();

  String message = '';

  @override
  void initState() {
    message = 'belum ada pesan diterima';
    super.initState();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.chatData.name),
      ),
      body: SafeArea(
        child: Column(
          children: [
            Expanded(
              child: Center(
                child: Text(message)
              )
            ),
            Container(
```

```

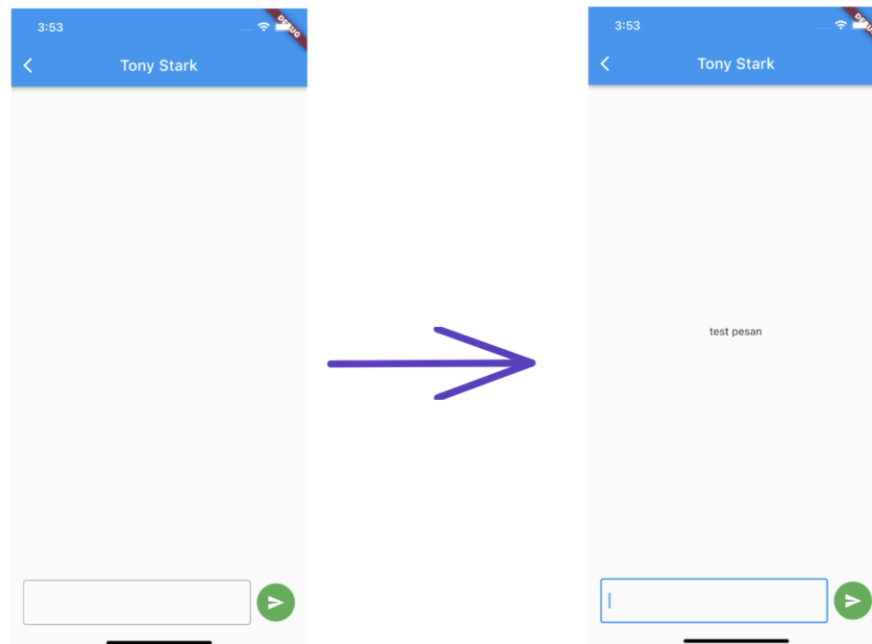
padding: const EdgeInsets.symmetric(horizontal: 16),
child: Row(
  children: [
    Expanded(
      child: TextField(
        controller: messageController,
        decoration: InputDecoration(
          border: OutlineInputBorder(),
        ),
      ),
    ),
    SizedBox(width: 8,),
    CircleAvatar(
      radius: 25,
      backgroundColor: Colors.green,
      child: IconButton(
        color: Colors.white,
        onPressed: () {
          String msg = messageController.text;

          setState(() {
            message = msg;
          });
        },
        icon: Icon(Icons.send)
      ),
    ),
  ],
),
),
);
}

```

Selanjutnya kita akan mengawasi perubahan data pada suatu variabel dan mengupdate UI / tampilan aplikasi sesuai dengan data terbaru. Dimana contoh kasus yang sudah kita terapkan adalah aplikasi akan menampilkan pesan apapun

yang diketik oleh pengguna ketika tombol kirim ditekan.



Pada contoh kasus diatas, kita mengawasi perubahan state pada variabel String yang berisi pesan yang diketik oleh pengguna. Pada praktiknya, `setState` tidak hanya sebatas mengawasi perubahan variabel String saja, secara default `setState` mendukung semua tipe data mulai dari Number (int dan double), String, List, File dan lain sebagainya, dan state ini tidak hanya sebatas 1 variabel saja, dalam satu halaman ataupun widget kita bisa menggunakan sebanyak mungkin variable state, namun hal ini pasti akan berbanding lurus dengan performa aplikasi dimana semakin banyak variable state yang digunakan, maka akan semakin banyak pula waktu yang akan dibutuhkan aplikasi untuk menampilkannya dan otomatis akan berpengaruh ke performa aplikasi.

Maka dari itu penggunaan variabel-variabel state dan kapan waktu untuk me-render ulang tampilan harus benar-benar diperhitungkan dan didesain secara benar, tahap inilah yang disebut dengan istilah state management. Hal ini bisa dilakukan dengan 2 hal, yaitu manage state dan variable state secara baik dan benar atau bisa juga menggunakan state management library yang sesuai dan cocok untuk requirement aplikasi.

Project Lanjutan

Untuk tahap berikutnya dari lanjutan project aplikasi yang kita bangun hanya menampilkan satu pesan yang diketik oleh pengguna, dan pesan tersebut akan ditimpa oleh pesan yang baru. Fitur tersebut tentunya tidak sesuai dengan aplikasi chat yang sebenarnya, dimana aplikasi chat yang sebenarnya menampung semua pesan yang diinput atau dikirim oleh pengguna dan ditampilkan dalam bentuk list jadi

tidak ada pesan yang menimpa pesan lain. Untuk memenuhi kebutuhan fitur tersebut, ada baiknya jika kita membuat list requirement (kebutuhan) yang akan kita terapkan pada projek ini, adapun requirement yang akan kita buat adalah sebagai berikut:

1. Pesan ditampilkan dalam bentuk list (*gunakan ListView.builder*),
2. Adanya dekorasi pada masing-masing pesan dan berada di sebelah kanan,
3. List Pesan di scroll dari bawah ke atas (*gunakan reverse: true pada ListView*),
4. Pesan baru akan selalu mengambil posisi paling bawah (*keyword: flutter list insert*),
5. Cegah pengguna menambahkan pesan kosong,
6. Bersihkan TextField jika pesan sudah berhasil ditambahkan.

Untuk memenuhi requirement tersebut, hal pertama yang perlu dilakukan adalah melakukan perubahan pada struktur state yang akan kita gunakan, pada kasus ini, perubahan struktur state hanya sebatas tipe data dari variabel state dan widget yang akan menampung atau menampilkan data dari variabel tersebut.

Untuk deklarasi variable state, pesan yang kita terima dari pengguna tetap dalam bentuk string tapi dalam jumlah yang banyak, maka dari itu kita akan membuat variabel state dengan tipe data List dengan content String yang secara default akan kita deklarasi sebagai list kosong:

```
class _ChatPageState extends State<ChatPage> {  
  
  // deklarasi variable daftar/list pesan  
  final List<String> messages = [];  
  
  final messageController = TextEditingController();  
  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: Text(widget.chatData.name),  
      ), // AppBar kanaydo, 09/01/23 17.46 • add chat page  
      body: SafeArea(  

```

Kemudian kita akan membuat widget yang akan bertanggung jawab sebagai media atau widget yang akan menampilkan isi dari variabel tersebut, karena variable kita menggunakan tipe data List maka widget yang akan kita gunakan adalah ListView dengan factory builder, dan masing-masing pesan didalam list akan ditampilkan menggunakan widget Text seperti biasanya.

```

)   body: SafeArea(
)     child: Column(
)       children: [
)         Expanded(
)           child: ListView.builder(
)             itemCount: messages.length,
)             itemBuilder: (BuildContext context, int index) {
)               final message = messages[index]; You, 4 minutes
)               return Text(message);
)             },
)           ) // ListView.builder
)         ), // Expanded
)       Container(
)         padding: const EdgeInsets.symmetric(horizontal: 16),
)         child: Row(

```

Seperti yang terlihat pada gambar sebelumnya, widget **ListView.builder** menggunakan 2 properti yaitu:

1. **itemCount**: yang berfungsi untuk memberitahu ListView jumlah data yang akan ditampilkan, properti ini bersifat wajib karena jika tidak dideklarasikan, maka ListView akan beranggapan bahwa jumlah datanya tak terhingga (infinite).
2. **itemBuilder**: yang berfungsi untuk membangun (build masing-masing widget pesan yang kita miliki) dimana kembalinya adalah widget. Untuk saat ini kita akan menggunakan widget Text.

Menerima input dari pengguna

Sama seperti pada pertemuan sebelumnya, kita tetap menggunakan **TextEditingController** untuk mengambil input dari pengguna yang ada pada widget **TextField**, dan akan ditampung pada variabel sementara:

```

— CircleAvatar(
  radius: 25,
  backgroundColor: Colors.green,
  child: IconButton(
    color: Colors.white,
    onPressed: () {

      /// ambil input pengguna dari TextField
      /// dan tampung pada variable message
      String message = messageController.text;

    },
    icon: Icon(Icons.send)
  ), // IconButton
) // CircleAvatar

```

Jika pesan dari pengguna sudah berhasil ditampung, maka langkah selanjutnya adalah menambahkan pesan tersebut ke dalam variabel *messages* yang sudah kita deklarasikan sebelumnya, dimana perubahan isi dari tersebut adalah state, maka kita akan menggunakan fungsi *setState* untuk memperbaharui isi dari variabel *messages* tadi dan memperbaharui tampilan sesuai dengan data yang baru.


```

CircleAvatar(
  radius: 25,
  backgroundColor: Colors.green,
  child: IconButton(
    color: Colors.white,
    onPressed: () {
      // ambil input pengguna dari TextField
      // dan tampung pada variable message
      String message = messageController.text;

      // gunakan fungsi setState
      setState(() {
        // tambahkan message kedalam messages dengan fungsi add
        messages.add(message);
      });
    },
    icon: Icon(Icons.send)
  ), // IconButton kanaydo, 09/01/23 17.46 • add chat page
) // CircleAvatar

```

Sampai sejauh ini, jika projek dijalankan aplikasi sudah bisa menampilkan pesan yang di input oleh pengguna dalam bentuk list, dan pesan yang sudah diketik sebelumnya tidak akan ditimpa oleh pesan yang baru.

Decorate Message Item

Kita bisa melakukan dekorasi pada item message pada properti itemBuilder yang ada pada widget ListView. Properti ini bebas untuk di dekorasi sesuai dengan kebutuhan, sebagai contoh: widget Text akan kita bungkus menggunakan widget Card dan akan kita letakkan di sebelah kanan:

```

child: ListView.builder(
  itemCount: messages.length,
  itemBuilder: (BuildContext context, int index) {

    final message = messages[index];
    return Padding(
      padding: const EdgeInsets.all(8),
      child: Align(
        alignment: Alignment.centerRight,
        child: Card(
          child: Padding(
            padding: const EdgeInsets.all(8),
            child: Text(message),
          ), // Padding
        ), // Card
      ), // Align
    ); // Padding
  },
) // ListView.builder

```

Reverse ListView

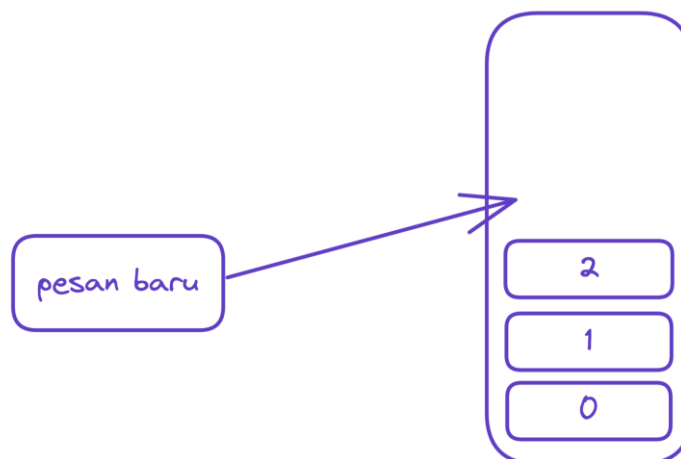
Untuk projek saat ini, ListView yang kita miliki masing menggunakan scroll behaviour secara default, dimana data akan discroll dari atas ke bawah (user akan menggunakan swipe up / ke atas) untuk melihat data selanjutnya. Hal ini tentu tidak sesuai dengan requirement dari aplikasi chat, dimana aplikasi chat pada umumnya menggunakan scroll dari bawah ke atas untuk load melihat data berikutnya.

untuk memperbaiki kita bisa menambahkan properti `reverse: true` pada ListView

```
child: ListView.builder(  
  /// tambahkan properti reverse: true  
  reverse: true,  
  itemCount: messages.length,  
  itemBuilder: (BuildContext context, int index) {
```

Posisi Pesan Baru Salah

Untuk saat ini program sudah berjalan namun posisi pesan yang baru akan bertambah diatas, ini tentunya tidak sesuai dengan aplikasi chat standard yang seharusnya pesan baru bertambah di bagian bawah. Hal ini bisa terjadi karena secara default jika kita menggunakan fungsi `add` pada list, maka value yang baru akan menggunakan index terakhir ditambah 1, contoh seperti ilustrasi di bawah 'pesan baru' saat ditambahkan ke dalam list akan menggunakan index ke 3, sehingga inilah yang menjadi penyebab pesan baru bertambah di atas list.

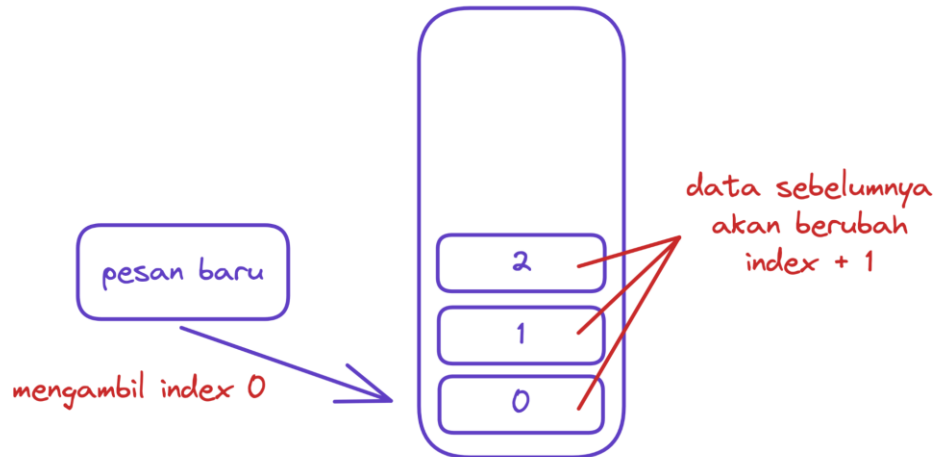


Untuk memperbaiki masalah ini, kita akan mengganti method `add` menjadi method `insert` untuk menambahkan pesan baru ke dalam list. Method `insert` sendiri memiliki 2 argumen yaitu:

1. **index**: dengan tipe data int, yang digunakan sebagai penunjuk ke index mana kita akan menambahkan data yang baru.
2. **element**: tipe data sesuai dengan konten dari list yang akan kita gunakan,

untuk kasus kita kali ini harus menggunakan tipe data String.

Method insert ini akan kita gunakan untuk menambahkan pesan baru di posisi bawah atau awal dari list, yang artinya kita akan menargetkan index 0, dan data lainnya akan otomatis mengalami perubahan pada index (bertambah 1)



Untuk sisi code-nya sendiri kita hanya perlu mengubah method add menjadi method insert dengan target index adalah 0 (awal dari list).

```
CircleAvatar(  
  radius: 25,  
  backgroundColor: Colors.green,  
  child: IconButton(  
    color: Colors.white,  
    onPressed: () {  
      /// ambil input pengguna dari TextField  
      /// dan tampung pada variable message  
      String message = messageController.text;  
  
      /// gunakan fungsi setState  
      setState(() {  
        /// ubah menjadi method insert  
        messages.insert(0, message);  
      });  
    },  
    icon: Icon(Icons.send)  
  ), // IconButton  
)  
// CircleAvatar
```

Mencegah Pesan Kosong ditambahkan ke List

Aplikasi yang kita bangun harus bisa melakukan filter atau pengecekan, apakah user ada mengetik pesan di TextField, jika ada baru lakukan penambahan ke dalam list, jika tidak maka abaikan. Hal ini bisa kita lakukan dengan logic sederhana, yaitu function setState hanya akan kita eksekusi jika pesan yang diketik tidak kosong.

```
onPressed: () {  
  // ambil input pengguna dari TextField  
  // dan tampung pada variable message  
  String message = messageController.text;  
  
  // check jika pesan tidak kosong  
  if (message.isNotEmpty) {  
    // gunakan fungsi setState  
    setState(() {  
      // ubah menjadi method insert  
      messages.insert(0, message);  
    });  
  }  
},
```

Membersihkan TextField jika pesan berhasil ditambahkan

Widget TextField perlu untuk dibersihkan atau dikosongkan untuk mencegah pesan yang sama dikirim berulang kali dan juga untuk mempermudah pengguna untuk tidak lagi menghapus atau mengosongkan TextField secara manual. Untuk melakukan hal tersebut kita tetap menggunakan fitur dari TextEditingController yaitu method clear.

```
onPressed: () {  
  String message = messageController.text;  
  
  if (message.isNotEmpty) {  
    setState(() {  
      messages.insert(0, message);  
  
      // panggil method clear pada controller  
      messageController.clear();  
    });  
  }  
},
```

BAB VIII PENUTUP

Selamat anda telah selesai membaca buku pemrograman mobile dengan flutter. Pemrograman mobile merupakan hal yang sangat penting di era yang serba teknologi saat ini, oleh sebab itu sangatlah penting untuk mempelajarinya. Flutter merupakan framework yang ditujukan untuk Bahasa pemrograman mobile berbasis multiplatform, artinya developer dapat membuat aplikasi yang dapat berjalan di berbagai platform seperti Android, IOS, maupun Web dalam sekali coding.

DAFTAR PUSTAKA

- Biessek, A. (2019). *Flutter for Beginners: An introductory guide to building cross-platform mobile applications with Flutter and Dart 2*. Packt Publishing Ltd.
- Cornez, T. (2015). *Android Programming Concepts*. Jones & Bartlett Publishers.
- DiMarzio, J. (2016). *Beginning android programming with android studio*. John Wiley & Sons
- Horton, J. (2015). *Android programming for beginners*. Packt Publishing Ltd.
- Saputra, W. A. (2020). *Pemrograman Berbasis Objek Pemrograman Mobile Dengan Android Studio*. Deepublish.
- Syaputra, R., & Ganda, Y. P. W. (2019). *Happy Flutter: Membuat Aplikasi Andorid dan iOS dengan Mudah menggunakan Flutter-UDACODING*. Udacoding.
- Tashildar, A., Shah, N., Gala, R., Giri, T., & Chavhan, P. (2020). Application development using flutter. *International Research Journal of Modernization in Engineering Technology and Science*, 2(8), 1262-1266.

