

#11 | Dasar State Management

Praktikum 1: Dasar State dengan Model-View

Langkah 1: Buat Project Baru

Buatlah sebuah project flutter baru dengan nama master_plan di folder src week-11 repository GitHub Anda.



Langkah 2: Membuat model `task.dart`

Praktik terbaik untuk memulai adalah pada lapisan data (data layer). Ini akan memberi Anda gambaran yang jelas tentang aplikasi Anda, tanpa masuk ke detail antarmuka pengguna Anda. Di folder model, buat file bernama `task.dart` dan buat class `Task`. Class ini memiliki atribut `description` dengan tipe data `String` dan `complete` dengan tipe data `Boolean`, serta ada konstruktor. Kelas ini akan menyimpan data tugas untuk aplikasi kita.

```
class Task {  
  final String description;  
  final bool complete;  
  
  const Task({  
    this.complete = false,  
    this.description = '',  
  });  
}
```

Langkah 3: Buat file `plan.dart`

Kita juga perlu sebuah `List` untuk menyimpan daftar rencana dalam aplikasi to-do ini. Buat file `plan.dart` di dalam folder `models` dan isi kode seperti berikut.

```
import './task.dart';  
  
class Plan {  
  final String name;  
  final List<Task> tasks;
```

```
const Plan({this.name = '', this.tasks = const []});
}
```

Langkah 4: Buat file `data_layer.dart`

Kita dapat membungkus beberapa data layer ke dalam sebuah file yang nanti akan mengekspor kedua model tersebut. Dengan begitu, proses impor akan lebih ringkas seiring berkembangnya aplikasi. Buat file bernama `data_layer.dart` di folder `models`.

```
export 'plan.dart';
export 'task.dart';
```

Langkah 5: Pindah ke file `main.dart`

Ubah isi kode `main.dart` sebagai berikut.

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

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

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

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      theme: ThemeData(primarySwatch: Colors.purple),
      home: PlanScreen(),
    );
  }
}
```

Langkah 6: buat `plan_screen.dart`

Pada folder `views`, buatlah sebuah file `plan_screen.dart` dan gunakan templat `StatefulWidget` untuk membuat class `PlanScreen`. Isi kodenya adalah sebagai berikut. Gantilah teks 'Namaku' dengan nama panggilan Anda pada title `AppBar`.

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

class PlanScreen extends StatefulWidget {
  const PlanScreen({super.key});
```

```

    @override
    State createState() => _PlanScreenState();
  }

  class _PlanScreenState extends State<PlanScreen> {
    Plan plan = const Plan();

    @override
    Widget build(BuildContext context) {
      return Scaffold(
        // ganti 'Namaku' dengan Nama panggilan Anda
        appBar: AppBar(title: const Text('Master Plan Namaku')),
        body: _buildList(),
        floatingActionButton: _buildAddTaskButton(),
      );
    }
  }
}

```

Langkah 7: buat method _buildAddTaskButton()

Anda akan melihat beberapa error di langkah 6, karena method yang belum dibuat. Ayo kita buat mulai dari yang paling mudah yaitu tombol Tambah Rencana. Tambah kode berikut di bawah method build di dalam class `_PlanScreenState`.

```

import '../models/data_layer.dart';
import 'package:flutter/material.dart';

class PlanScreen extends StatefulWidget {
  const PlanScreen({super.key});

  @override
  State createState() => _PlanScreenState();
}

class _PlanScreenState extends State<PlanScreen> {
  Plan plan = const Plan();

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      // ganti 'Namaku' dengan Nama panggilan Anda
      appBar: AppBar(title: const Text('Master Plan Namaku')),
      body: _buildList(),
      floatingActionButton: _buildAddTaskButton(),
    );
  }

  Widget _buildAddTaskButton() {
    return FloatingActionButton(

```

```

        child: const Icon(Icons.add),
        onPressed: () {
          setState(() {
            plan = Plan(
              name: plan.name,
              tasks: List<Task>.from(plan.tasks)
                ..add(const Task()),
            );
          });
        },
      ),
    );
  }
}

```

Langkah 8: buat widget `_buildList()`

Kita akan buat widget berupa List yang dapat dilakukan scroll, yaitu `ListView.builder`. Buat widget `ListView` seperti kode berikut ini.

```

Widget _buildList() {
  return ListView.builder(
    itemCount: plan.tasks.length,
    itemBuilder: (context, index) =>
      _buildTaskTile(plan.tasks[index], index),
  );
}

```

Langkah 9: buat widget `_buildTaskTile`

Dari langkah 8, kita butuh `ListTile` untuk menampilkan setiap nilai dari `plan.tasks`. Kita buat dinamis untuk setiap index data, sehingga membuat view menjadi lebih mudah. Tambahkan kode berikut ini.

```

import '../models/data_layer.dart';
import 'package:flutter/material.dart';

class PlanScreen extends StatefulWidget {
  const PlanScreen({super.key});

  @override
  State createState() => _PlanScreenState();
}

class _PlanScreenState extends State<PlanScreen> {
  Plan plan = const Plan();

  @override
  Widget build(BuildContext context) {
    return Scaffold(

```

```

        // ganti 'Namaku' dengan Nama panggilan Anda
        appBar: AppBar(title: const Text('Master Plan Namaku')),
        body: _buildList(),
        floatingActionButton: _buildAddTaskButton(),
    );
}

Widget _buildAddTaskButton() {
    return FloatingActionButton(
        child: const Icon(Icons.add),
        onPressed: () {
            setState(() {
                plan = Plan(
                    name: plan.name,
                    tasks: List<Task>.from(plan.tasks)..add(const Task()),
                );
            });
        },
    );
}

Widget _buildList() {
    return ListView.builder(
        itemCount: plan.tasks.length,
        itemBuilder: (context, index) => _buildTaskTile(plan.tasks[index],
index),
    );
}

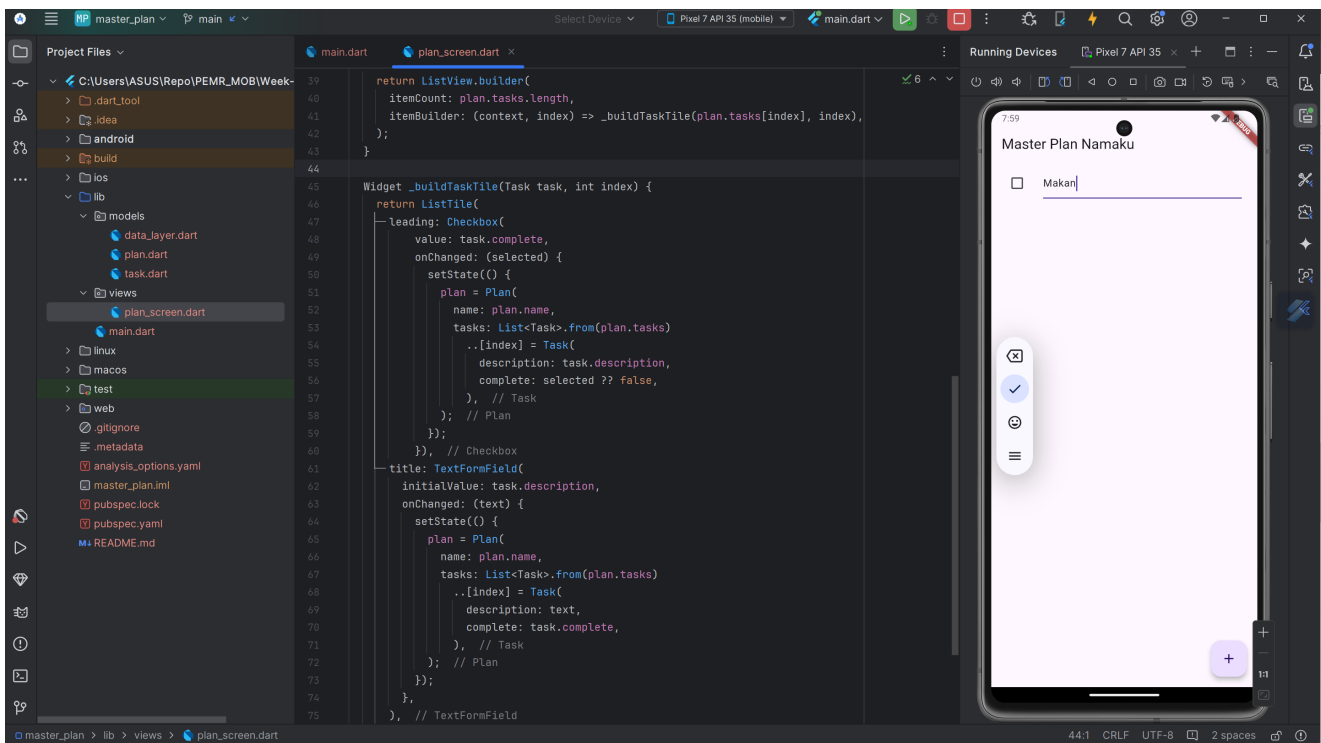
Widget _buildTaskTile(Task task, int index) {
    return ListTile(
        leading: Checkbox(
            value: task.complete,
            onChanged: (selected) {
                setState(() {
                    plan = Plan(
                        name: plan.name,
                        tasks: List<Task>.from(plan.tasks)
                            ..[index] = Task(
                                description: task.description,
                                complete: selected ?? false,
                            ),
                    );
                });
            },
        ),
        title: TextFormField(
            initialValue: task.description,
            onChanged: (text) {
                setState(() {
                    plan = Plan(
                        name: plan.name,
                        tasks: List<Task>.from(plan.tasks)
                            ..[index] = Task(

```

```

        description: text,
        complete: task.complete,
      ),
    ),
  ),
},
),
},
}
}

```



Langkah 10: Tambah Scroll Controller

Anda dapat menambah tugas sebanyak-banyaknya, menandainya jika sudah beres, dan melakukan scroll jika sudah semakin banyak isinya. Namun, ada salah satu fitur tertentu di iOS perlu kita tambahkan. Ketika keyboard tampil, Anda akan kesulitan untuk mengisi yang paling bawah. Untuk mengatasi itu, Anda dapat menggunakan `ScrollController` untuk menghapus focus dari semua `TextField` selama event scroll dilakukan. Pada file `plan_screen.dart`, tambahkan variabel scroll controller di class State tepat setelah variabel `plan`.

```

class _PlanScreenState extends State<PlanScreen> {
  Plan plan = const Plan();
  late ScrollController scrollController;

```

Langkah 11: Tambah Scroll Listener

Tambahkan method `initState()` setelah deklarasi variabel `scrollController` seperti kode berikut.

```

class _PlanScreenState extends State<PlanScreen> {
  Plan plan = const Plan();
  late ScrollController scrollController;

  @override
  void initState() {
    super.initState();
    scrollController = ScrollController()
      ..addListener(() {
        FocusScope.of(context).requestFocus(FocusNode());
      });
  }
}

```

Langkah 12: Tambah controller dan keyboard behavior

Tambahkan `controller` dan keyboard behavior pada `ListView` di method `_buildList` seperti kode berikut ini.

```

Widget _buildList() {
  return ListView.builder(
    controller: scrollController,
    keyboardDismissBehavior: Theme.of(context).platform ==
      TargetPlatform.iOS
      ? ScrollViewKeyboardDismissBehavior.onDrag
      : ScrollViewKeyboardDismissBehavior.manual,
    itemCount: plan.tasks.length,
    itemBuilder: (context, index) => _buildTaskTile(plan.tasks[index],
index),
  );
}

```

Langkah 13: Terakhir, tambah method `dispose()`

Terakhir, tambahkan method `dispose()` berguna ketika widget sudah tidak digunakan lagi.

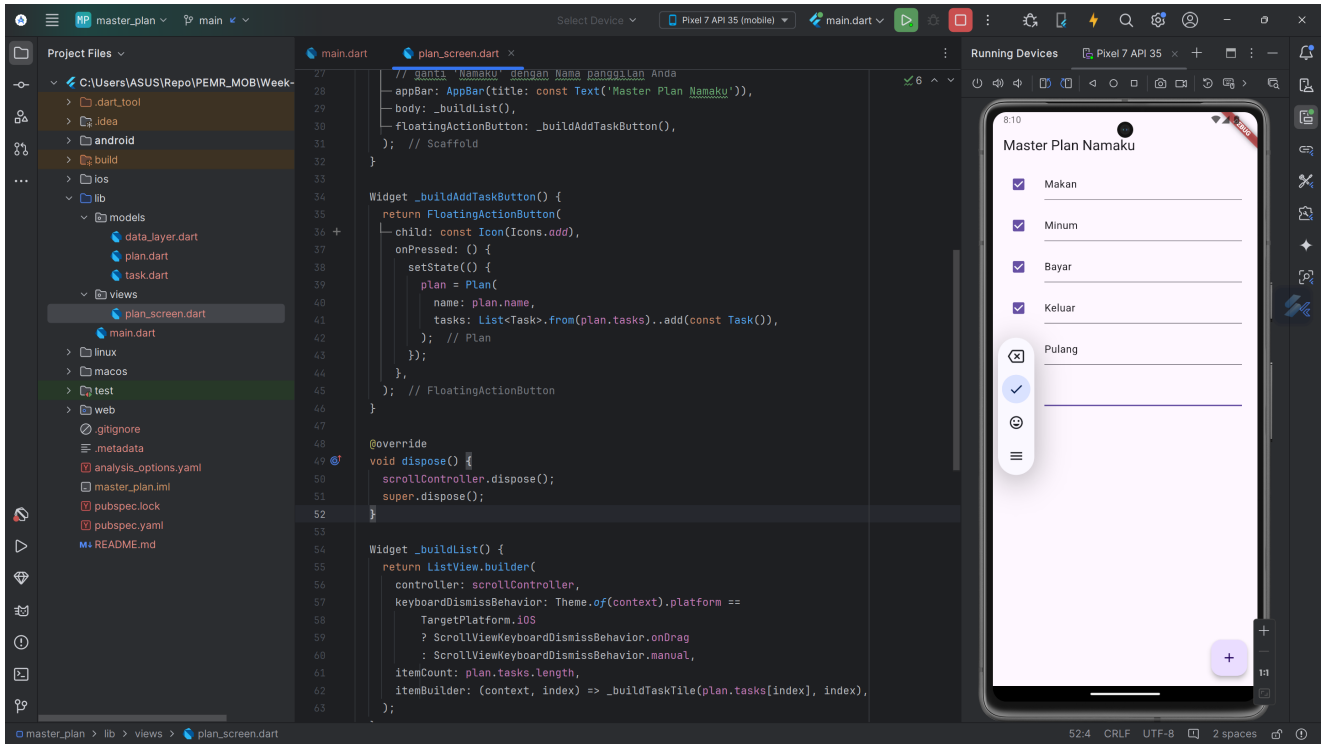
```

@override
void dispose() {
  scrollController.dispose();
  super.dispose();
}

```

Langkah 14: Hasil

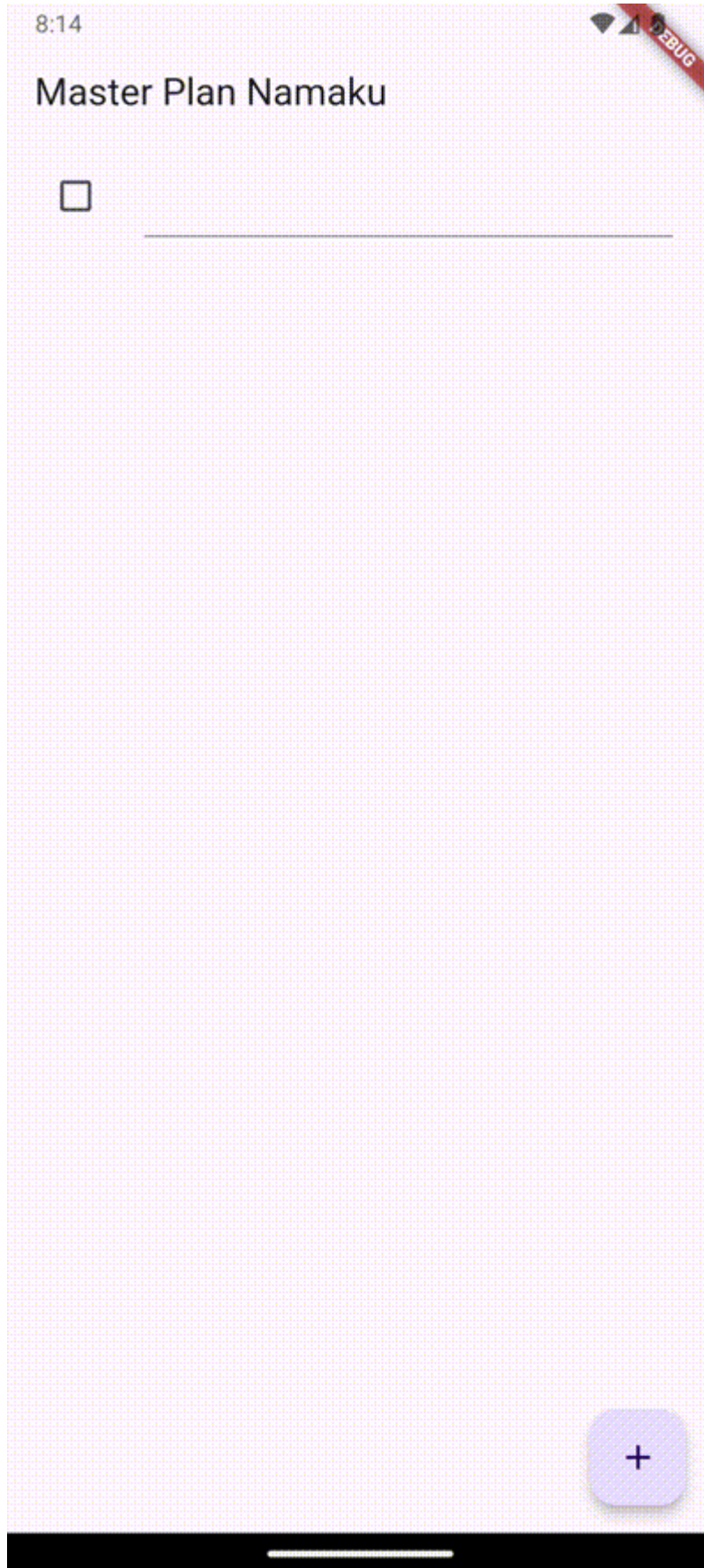
Lakukan **Hot** restart (bukan **hot** reload) pada aplikasi Flutter Anda.



Tugas Praktikum 1: Dasar State dengan Model-View

1. Selesaikan langkah-langkah praktikum tersebut, lalu dokumentasikan berupa GIF hasil akhir praktikum beserta penjelasannya di file README.md! Jika Anda menemukan ada yang error atau

tidak berjalan dengan baik, silakan diperbaiki.



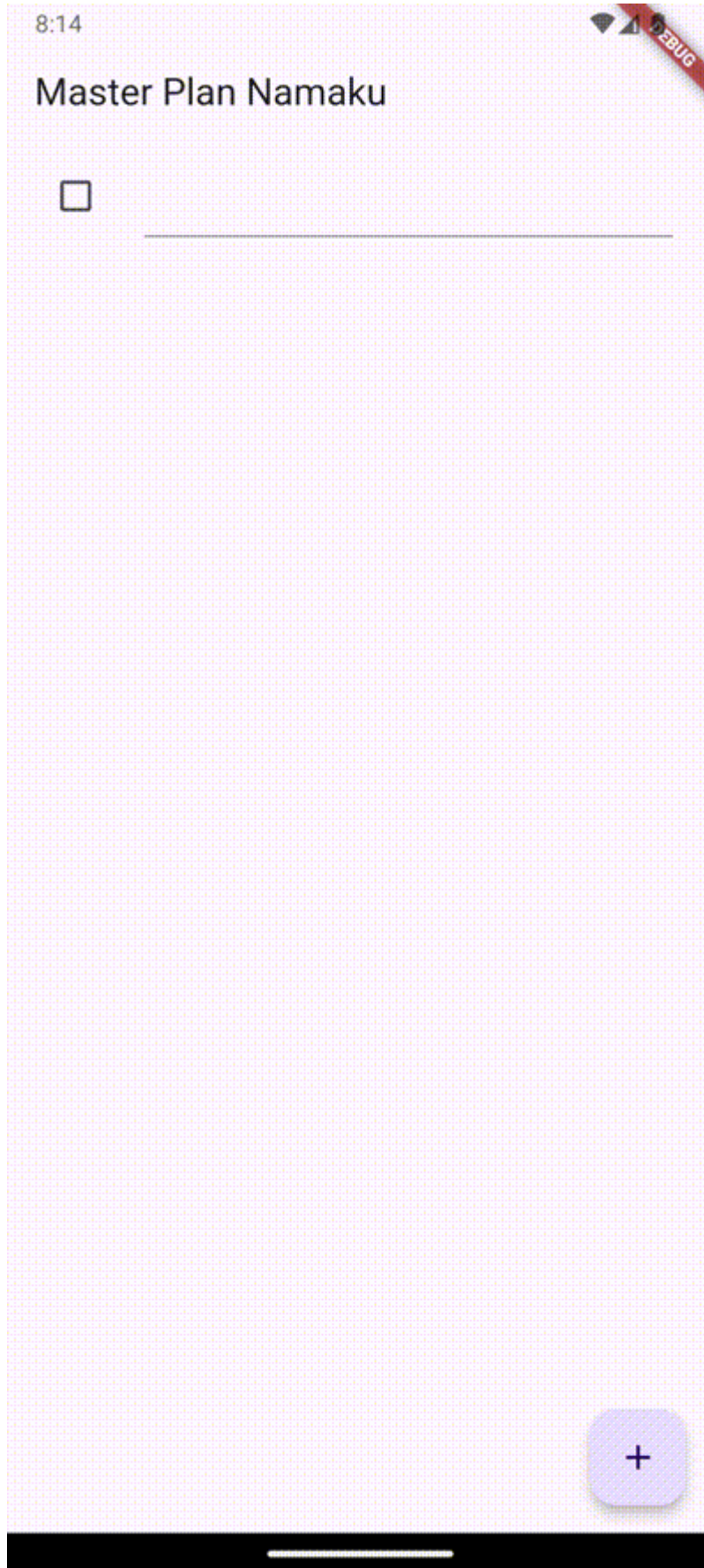
2. Jelaskan maksud dari langkah 4 pada praktikum tersebut! Mengapa dilakukan demikian?

Mengurangi Redundansi: Saat aplikasi berkembang dan memiliki lebih banyak file model, kita tidak perlu mengimpor masing-masing file satu per satu di setiap file lain yang membutuhkan akses ke model-model tersebut. Cukup mengimpor `data_layer.dart` akan mencakup semuanya.

3. Mengapa perlu variabel plan di langkah 6 pada praktikum tersebut? Mengapa dibuat konstanta ?

Variabel plan digunakan untuk merepresentasikan data atau model dari rencana yang akan dikelola di layar (screen) tersebut. Ini merupakan bagian dari logika aplikasi yang menghubungkan data dengan antarmuka pengguna (UI), khususnya untuk menampilkan dan memodifikasi daftar tugas atau rencana dalam aplikasi.

4. Lakukan capture hasil dari Langkah 9 berupa GIF, kemudian jelaskan apa yang telah Anda buat!



Membuat widget `_buildTaskTile` untuk menampilkan setiap tugas yang ada dalam daftar `plan.tasks` menggunakan komponen `ListTile`. Widget ini bertanggung jawab untuk

menampilkan setiap tugas secara dinamis berdasarkan index dari daftar tugas, serta memungkinkan pengguna untuk mengubah status dan deskripsi tugas.

5. Apa kegunaan method pada Langkah 11 dan 13 dalam lifecycle state ?

Menggunakan `dispose()` untuk memastikan bahwa widget membersihkan semua resources yang digunakan ketika tidak lagi dibutuhkan. Dalam siklus hidup state, `dispose()` adalah langkah terakhir yang memastikan memori dan sumber daya lain yang terkait dengan widget tidak digunakan secara berlebihan atau tetap aktif setelah widget dihapus.

6. Kumpulkan laporan praktikum Anda berupa link commit atau repository GitHub ke spreadsheet yang telah disediakan!

Praktikum 2: Mengelola Data Layer dengan `InheritedWidget` dan `InheritedNotifier`

Langkah 1: Buat file `plan_provider.dart`

Buat folder baru `provider` di dalam folder `lib`, lalu buat file baru dengan nama `plan_provider.dart` berisi kode seperti berikut.

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

class PlanProvider extends InheritedNotifier<ValueNotifier<Plan>> {
  const PlanProvider({super.key, required Widget child, required
    ValueNotifier<Plan> notifier})
    : super(child: child, notifier: notifier);

  static ValueNotifier<Plan> of(BuildContext context) {
    return context.
      dependOnInheritedWidgetOfExactType<PlanProvider>()!.notifier!;
  }
}
```

Langkah 2: Edit `main.dart`

Gantilah pada bagian atribut `home` dengan `PlanProvider` seperti berikut.

```
import 'package:flutter/material.dart';
import './views/plan_screen.dart';
import './provider/plan_provider.dart';
import './models/data_layer.dart';

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

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

```

@override
Widget build(BuildContext context) {
  return MaterialApp(
    theme: ThemeData(primarySwatch: Colors.purple),
    home: PlanProvider(
      notifier: ValueNotifier<Plan>(const Plan()),
      child: const PlanScreen(),
    ),
  );
}

```

Langkah 3: Tambah method pada model plan.dart

Tambahkan dua method di dalam model class Plan seperti kode berikut.

```

import './task.dart';

class Plan {
  final String name;
  final List<Task> tasks;

  const Plan({this.name = '', this.tasks = const []});

  int get completedCount => tasks.where((task) => task.complete).length;

  String get completenessMessage =>
    '$completedCount out of ${tasks.length} tasks';
}

```

Langkah 4: Pindah ke PlanScreen

Edit **PlanScreen** agar menggunakan data dari **PlanProvider**. Hapus deklarasi variabel **plan** (ini akan membuat error). Kita akan perbaiki pada langkah 5 berikut ini.

Langkah 5: Edit method _buildAddTaskButton

Tambahkan **BuildContext** sebagai parameter dan gunakan **PlanProvider** sebagai sumber datanya.

```

Widget _buildAddTaskButton(BuildContext context) {
  ValueNotifier<Plan> planNotifier = PlanProvider.of(context);
  return FloatingActionButton(
    child: const Icon(Icons.add),
    onPressed: () {
      Plan currentPlan = planNotifier.value;
      planNotifier.value = Plan(
        name: currentPlan.name,

```

```

        tasks: List<Task>.from(currentPlan.tasks)..add(const Task()),
    );
},
);
}

```

Langkah 6: Edit method _buildTaskTile

Tambahkan parameter `BuildContext`, gunakan `PlanProvider` sebagai sumber data. Ganti `TextField` menjadi `TextFormField` untuk membuat inisial data `provider` menjadi lebih mudah.

```

Widget _buildTaskTile(Task task, int index, BuildContext context) {
  ValueNotifier<Plan> planNotifier = PlanProvider.of(context);
  return ListTile(
    leading: Checkbox(
      value: task.complete,
      onChanged: (selected) {
        Plan currentPlan = planNotifier.value;
        planNotifier.value = Plan(
          name: currentPlan.name,
          tasks: List<Task>.from(currentPlan.tasks)
            ..[index] = Task(
              description: task.description,
              complete: selected ?? false,
            ),
        );
      },
    ),
    title: TextFormField(
      initialValue: task.description,
      onChanged: (text) {
        Plan currentPlan = planNotifier.value;
        planNotifier.value = Plan(
          name: currentPlan.name,
          tasks: List<Task>.from(currentPlan.tasks)
            ..[index] = Task(
              description: text,
              complete: task.complete,
            ),
        );
      },
    ),
  );
}

```

Langkah 7: Edit _buildList

Sesuaikan parameter pada bagian `_buildTaskTile` seperti kode berikut.

```
Widget _buildList(Plan plan) {
  return ListView.builder(
    controller: scrollController,
    itemCount: plan.tasks.length,
    itemBuilder: (context, index) =>
      _buildTaskTile(plan.tasks[index], index, context),
  );
}
```

Langkah 8: Tetap di class PlanScreen

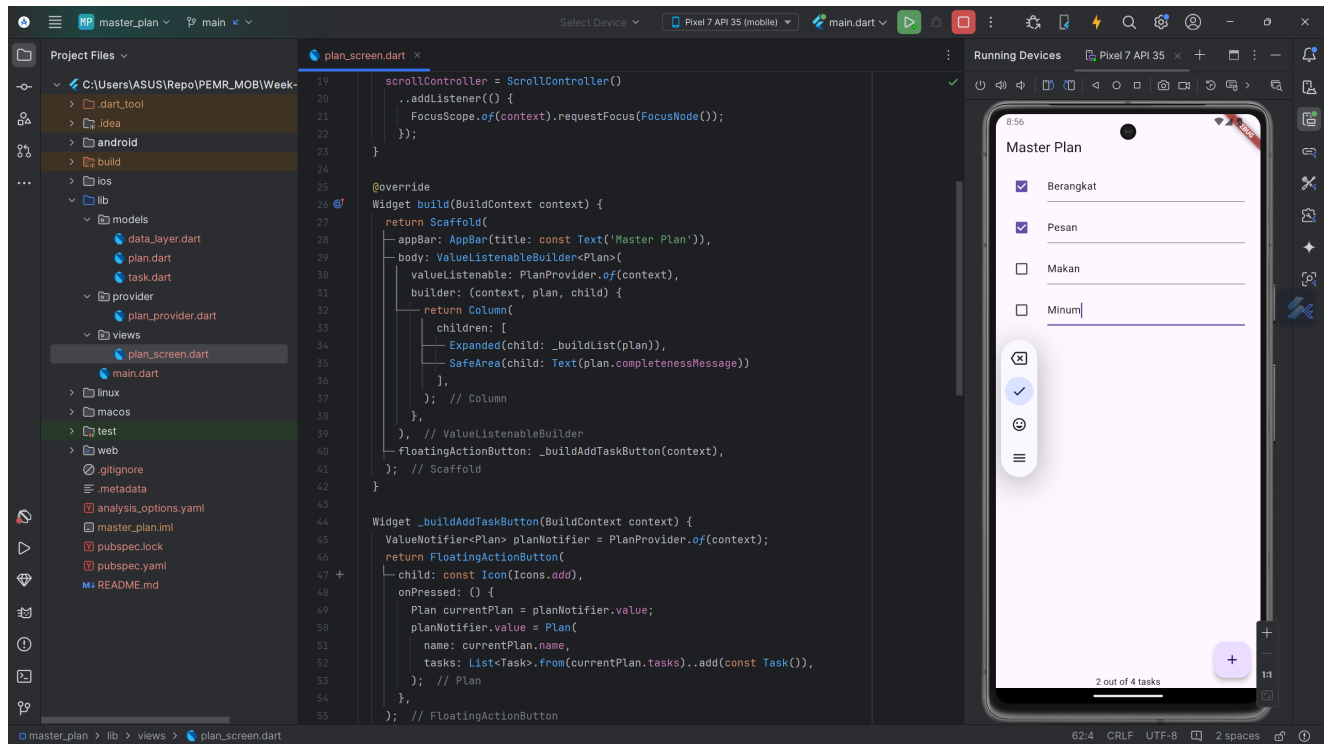
Edit method build sehingga bisa tampil progress pada bagian bawah (footer). Caranya, bungkus (wrap) `_buildList` dengan widget Expanded dan masukkan ke dalam widget Column seperti kode pada Langkah 9.

Langkah 9: Tambah widget SafeArea

Terakhir, tambahkan widget SafeArea dengan berisi completenessMessage pada akhir widget Column. Perhatikan kode berikut ini.

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: const Text('Master Plan')),
    body: ValueListenableBuilder<Plan>(
      valueListenable: PlanProvider.of(context),
      builder: (context, plan, child) {
        return Column(
          children: [
            Expanded(child: _buildList(plan)),
            SafeArea(child: Text(plan.completenessMessage))
          ],
        );
      },
    ),
    floatingActionButton: _buildAddTaskButton(context),
  );
}
```

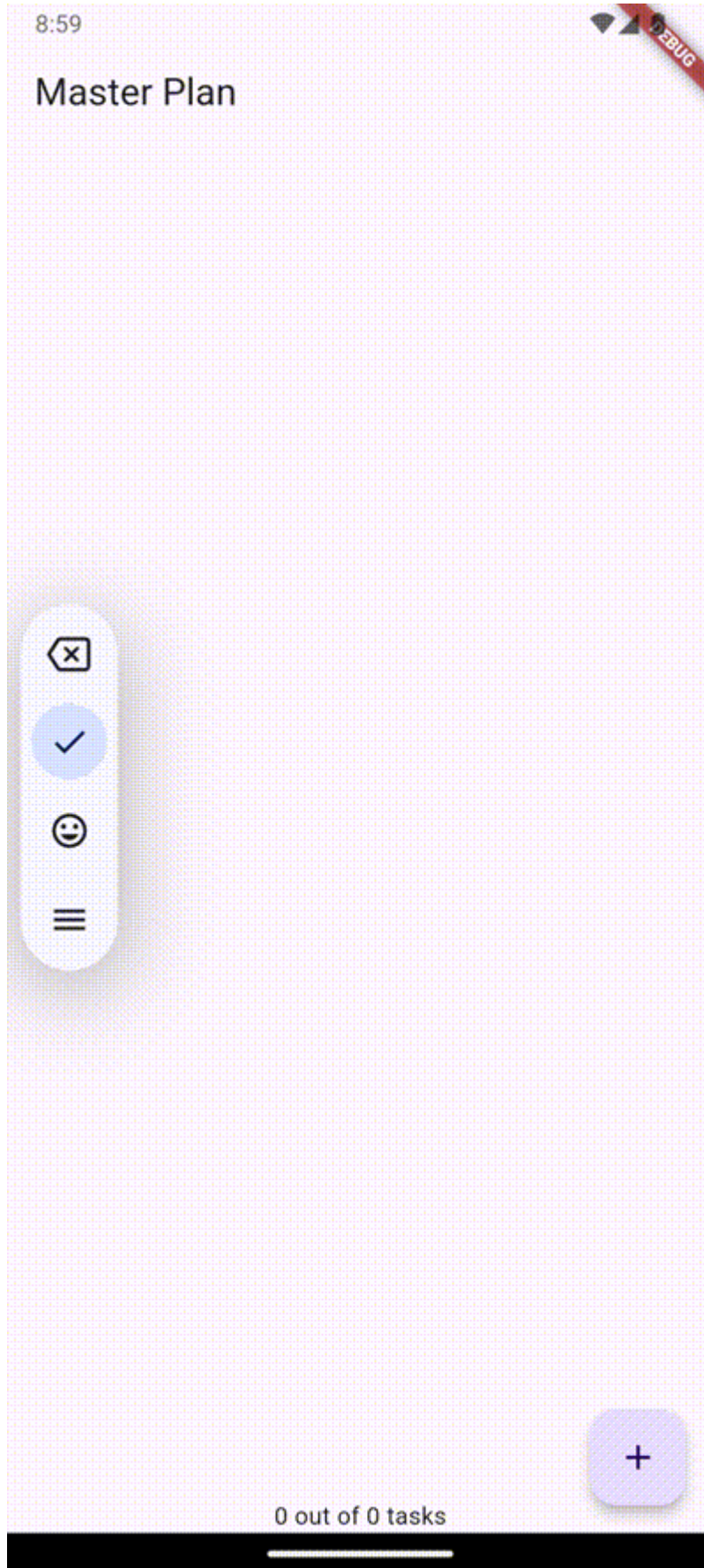
Hasil:



Tugas Praktikum 2: InheritedWidget

1. Selesaikan langkah-langkah praktikum tersebut, lalu dokumentasikan berupa GIF hasil akhir praktikum beserta penjelasannya di file README.md! Jika Anda menemukan ada yang error atau

tidak berjalan dengan baik, silakan diperbaiki sesuai dengan tujuan aplikasi tersebut dibuat.



2. Jelaskan mana yang dimaksud **InheritedWidget** pada langkah 1 tersebut! Mengapa yang digunakan **InheritedNotifier**?

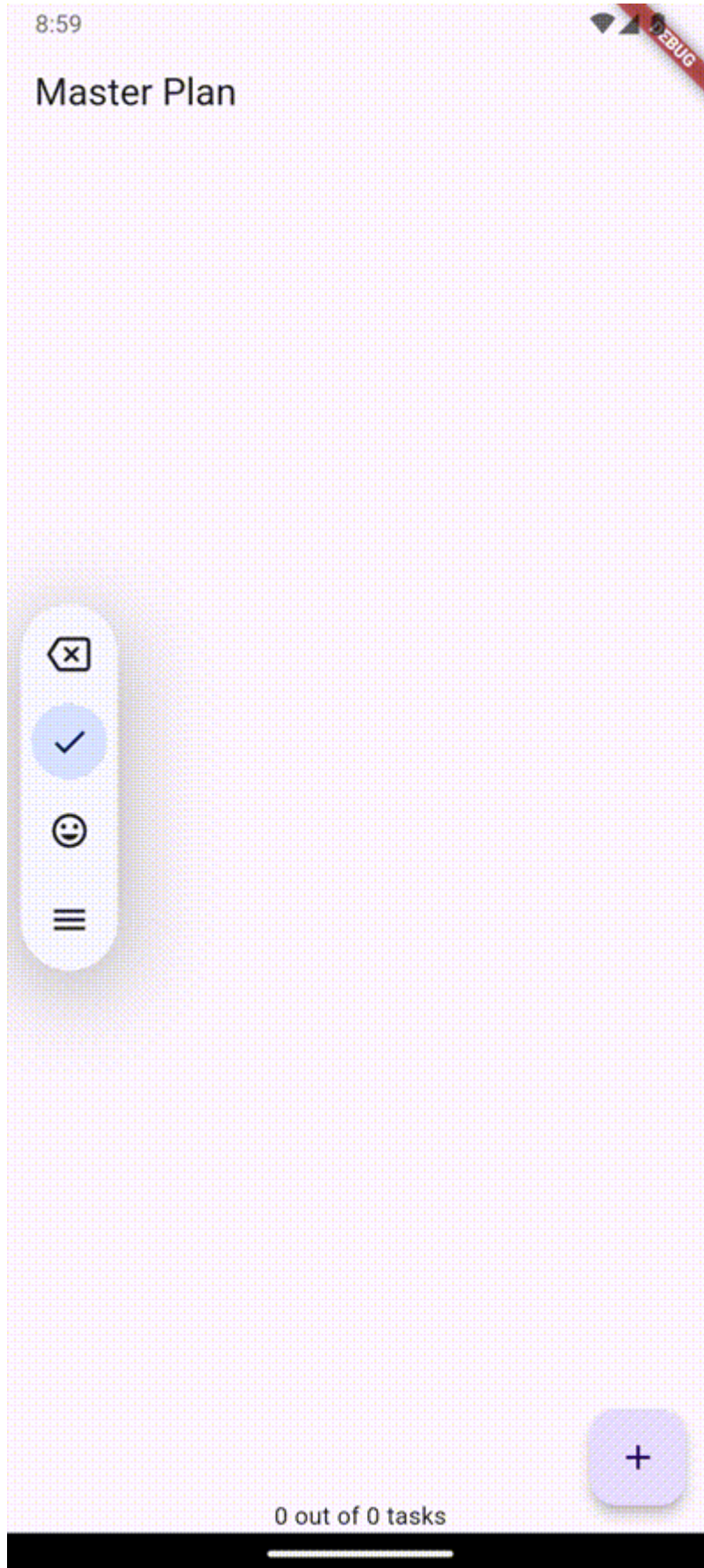
`InheritedWidget` menyediakan cara untuk berbagi data di antara widget dalam tree, sedangkan `InheritedNotifier` memperluas fungsionalitas dengan menggunakan `Notifier` (seperti `ValueNotifier`) untuk lebih mengontrol kapan widget harus memperbarui UI.

3. Jelaskan maksud dari method di langkah 3 pada praktikum tersebut! Mengapa dilakukan demikian?

Kedua method ini memberikan informasi penting mengenai status penyelesaian tugas dalam aplikasi.

- `completedCount` menghitung berapa banyak tugas yang sudah selesai.
- `completenessMessage` membuat pesan untuk menampilkan perbandingan antara tugas yang sudah selesai dan total tugas.

4. Lakukan capture hasil dari Langkah 9 berupa GIF, kemudian jelaskan apa yang telah Anda buat!



5. Kumpulkan laporan praktikum Anda berupa link commit atau repository GitHub ke spreadsheet yang telah disediakan!