Modul Praktikum

Pemrograman Mobile

8

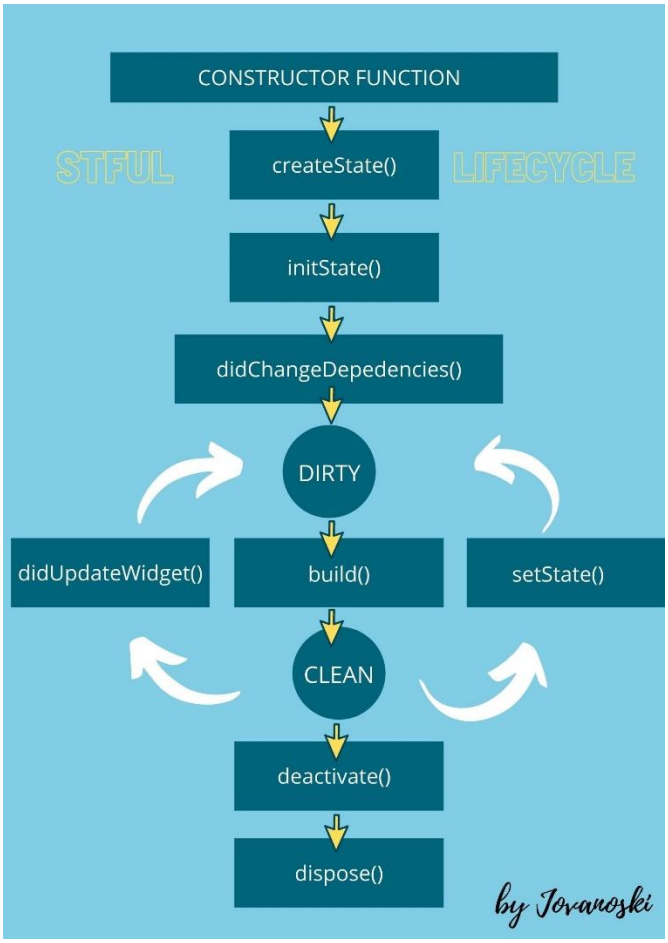
PROVIDER (STATE MANAGEMENT)

TUJUAN PEMBELAJARAN

A. Mahasiswa memahami cara kerja provider
B. Mahasiswa memahami lifecycle dari widgets flutter
C. Mahasiswa memahami penggunaan consumer
D. Mahasiswa bisa menerapkan multiple provider

DASAR TEORI

Widgets Lifecycle



Lifecycle widgets adalah urutan dari tahap-tahap pembuatan, pemeliharaan dan penghancuran dari sebuah widgets flutter. Lifecycle ini terjadi terhadap stateful widgets. Berikut adalah tahap tahapnya.

1. createState()

   Metode ini dipanggil saat kita membuat Widget Stateful baru. Ini adalah metode wajib. Ini akan mengembalikan sebuah instance dari State yang terkait dengannya.

```
class MyWidget extends StatefulWidget {
  const MyWidget({Key? key}) : super(key: key);

  @override
  State<MyWidget> createState() => _MyWidgetState();
}
```

2. initState()

   Ini adalah metode yang dipanggil saat Widget dibuat untuk pertama kalinya dan dipanggil tepat satu kali untuk setiap objek. Biasanya digunakan untuk mendeklarasi controller seperti text editing controller.

   Jika kita mendefinisikan atau menambahkan beberapa kode dalam metode initState() maka kode ini akan dieksekusi terlebih dahulu bahkan sebelum widget dibangun.

```
@override
void initState() {
  super.initState();
}
```

3. didChangeDependencies()

   Metode ini dipanggil segera setelah metode initState() saat pertama kali widget dibuat.

```
@override
void didChangeDependencies() {
  super.didChangeDependencies();
}
```

4. build()

   Metode ini adalah metode yang paling penting karena rendering semua widget bergantung padanya. Ini dipanggil setiap kali kita perlu merender Widget UI di layar.

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    // Isi Kode Kamu
```

```
  );
}
```

5. didUpdateWidget()

   Metode ini digunakan ketika ada beberapa perubahan konfigurasi oleh widget Induk. Ini pada dasarnya dipanggil setiap kali kami melakukan hot reload aplikasi untuk melihat pembaruan yang dilakukan pada widget.

```
@override
void didUpdateWidget(covariant MyWidget oldWidget) {
  super.didUpdateWidget(oldWidget);
}
```

6. setState()

   Metode setState() menginformasikan kerangka kerja bahwa keadaan internal objek ini telah berubah sedemikian rupa sehingga dapat memengaruhi UI yang menyebabkan kerangka kerja menjadwalkan build untuk keadaan objek ini.

```
setState(() {

});
```

7. deactivate()

   Metode deactivate() dipanggil jika object ini dihilangkan dari widget tree secara permanen ataupun sementara.

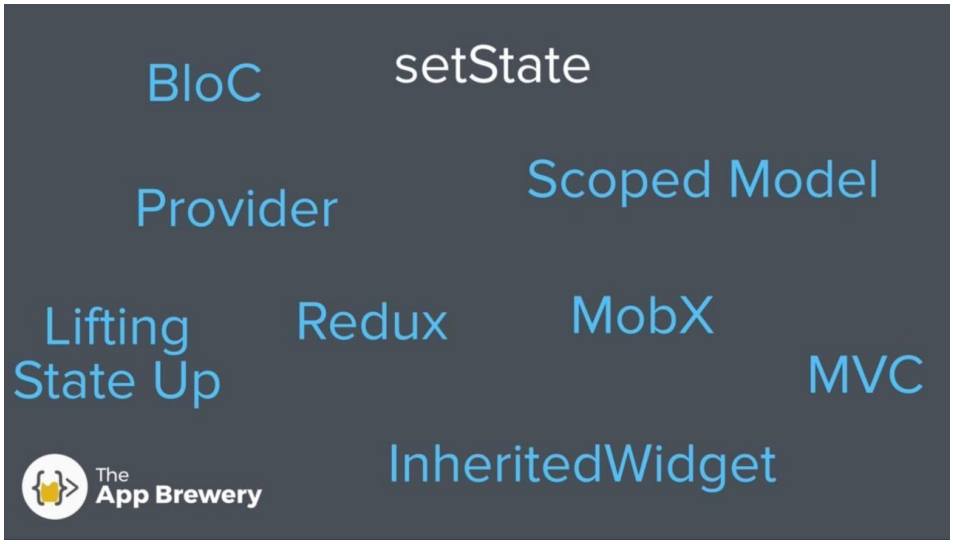
```
@override
void deactivate() {
  super.deactivate();
}
```

8. dispose()

   Metode dispose() hanya akan dipanggil jika object ini dihilangkan dari widget tree secara permanen. Metode ini biasanya digunakan untuk menghapus controller yang dideklarasikan di fungsi initState(). Hal ini bertujuan untuk mencegah memory leaks yang membuat aplikasi kita lag/freeze/force close.

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

Provider



Untuk State Management tentunya banyak solusi seperti diatas, bahkan kita sudah pernah menggunakan salah satunya yaitu setState. Tetapi menggunakan setState untuk aplikasi kompleks tidak disarankan karena alasan yang akan kita eksplor di modul ini.

Provider adalah salah satu solusi state management tersebut. Pada Google I/O 2019 Google Team merekomendasikan penggunaan package provider untuk state management. Tetapi kamu mungkin membutuhkan cara yang berbeda atau kombinasi dari solusi management, itu tidak salah. Itu tergantung use-case, gaya anda, dan kematuran framework.

Provider dibuat oleh Remi Rousselet yang terinspirasi dari state management dari google yaitu Provide. Remi membuatnya lebih baik sehingga dikenali dan dissuport oleh google team.

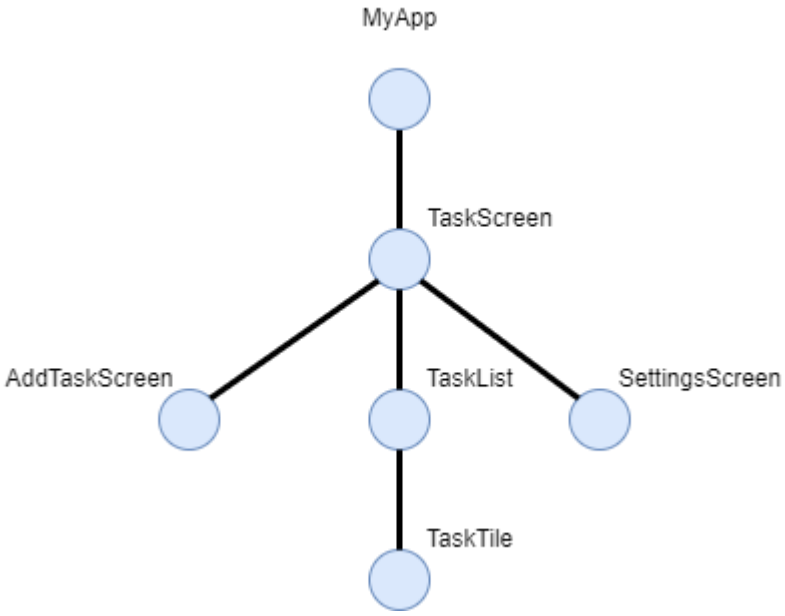
Provider bekerja dengan mengangkat state ke atas. Lalu widgets yang membutuhkan state tersebut bisa mengambil dan mendegarkan perubahan yang terjadi pada state tersebut dan mengubah dirinya secara otomatis. Jadi, kita tidak perlu menggunakan stateful widget untuk membuat widget tersebut bisa berubah.

Hal ini menyebabkan widget tree yang lebih sederhana karna kita hanya perlu menangani stateless widget, dan mencegah rebuild yang tidak perlu karna setiap anda memanggil setState(){}) widget akan rebuild secara keseluruhan.

PRAKTIK

Link Project: https://github.com/GTHGHT/praktikum_state_mngmnt_2023

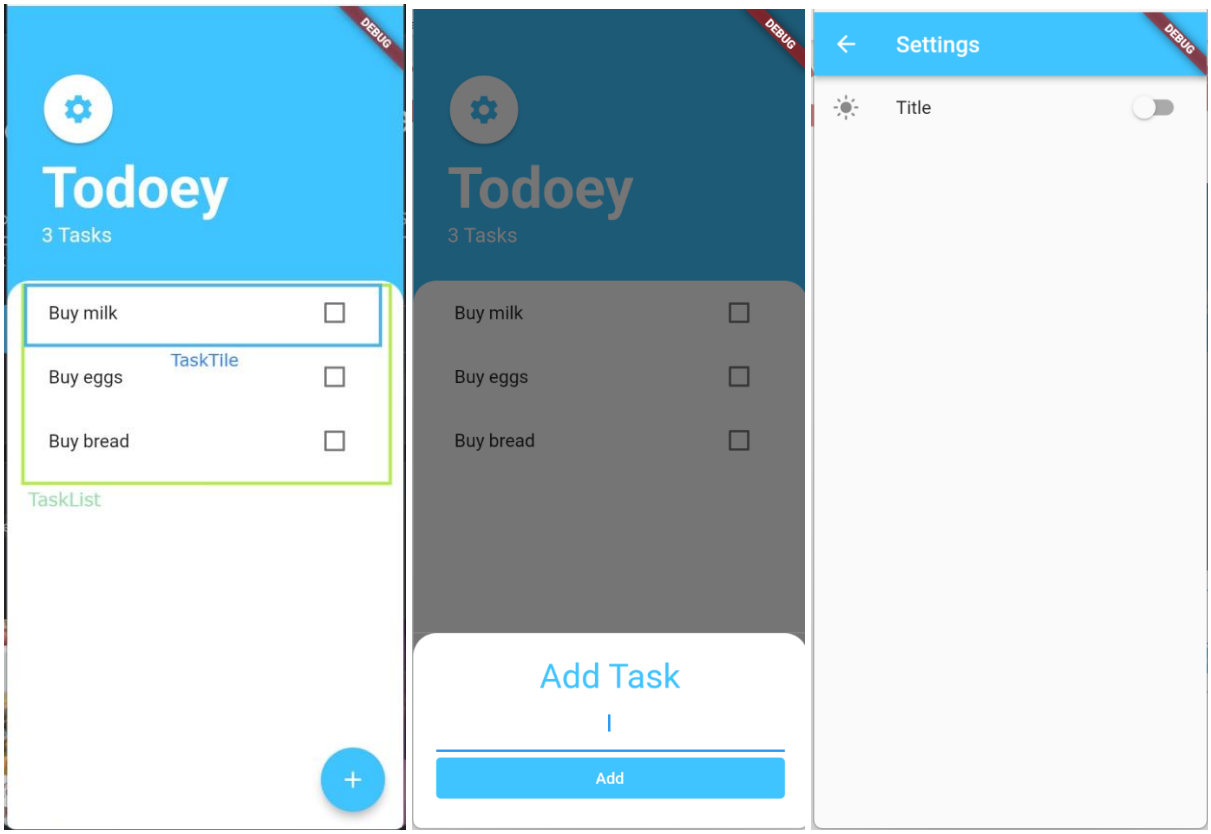
Preview Project



Project berikut menggunakan project yang berasal dari course udemy dari LondonAppBrewery yang anda bisa didapatkan di link berikut. Berikut adalah overview dari project praktikum:

- MyApp adalah Widget pertama yang mengurus MaterialApp
- TaskScreen adalah Widget halaman task yang menampilkan task, memanggil AddTaskScreen dan Settings Screen.
- TaskList adalah Widget daftar task
- TaskTile adalah Widget yang menampilkan nama task dan checkbox
- AddTaskScreen adalah widget yang menampilkan BottomSheet untuk menambahkan tasks
- SettingsScreen adalah widget yang menampilkan pilihan untuk menggunakan LightTheme atau DarkTheme

Preview Halaman



TasksScreen AddTaskScreen SettingsScreen

Latar Belakang

Pertama – tama kita mempunyai Class Task yang berguna untuk tipe data task yang berisi nama task dan status task.

```
class Task {
  final String name;
  bool isDone;

  Task({required this.name, this.isDone = false});

  void toggleDone() {
    isDone = !isDone;
  }
}
```

Pada TasksScreen kita mempunyai variable _tasks untuk menyimpan daftar tasks

```
class _TasksScreenState extends State<TasksScreen> {
  final List<Task> _tasks = [
    Task(name: 'Buy milk'),
    Task(name: 'Buy eggs'),
    Task(name: 'Buy bread'),
  ];
```

Yang mana AddTaskScreen, dan TaskTile membutuhkan juga daftar tersebut, maka kita berikan daftar tersebut ke dalam constructor.

```
class TasksList extends StatefulWidget {
  List<Task> tasks;

  TasksList({required this.tasks});
```

Pemanggilan TaskList

```
child: TasksList(
  tasks: _tasks,
),
```

Constructor AddTaskScreen

```
class AddTaskScreen extends StatelessWidget {
  List<Task> tasks;

  AddTaskScreen({super.key, required this.tasks});
```

Pemanggilan AddTaskScreen

```
child: AddTaskScreen(
  tasks: _tasks,
),
```

Masalah dengan pendekatan ini adalah jika kita lakukan penambahan tasks menggunakan AddTaskScreen, tasks tidak terlihat berubah kecuali kita lakukan perubahan seperti pengubahan status tasks. Hal ini bisa kita atasi dengan menggunakan ChangeNotifierProvider.

ChangeNotifierProvider

Pertama-tama kita buat file task_data.dart di folder model.

Buat class TaskData yang meng-extends ChangeNotifier.

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

class TaskData extends ChangeNotifier {

}
```

Lalu taruh state (data) yang ingin anda gunakan ke dalam class tersebut. Dalam Hal ini kita taruh list task ke dalam task data.

```
class TaskData extends ChangeNotifier {
  final List<Task> _tasks = [
    Task(name: 'Buy milk'),
    Task(name: 'Buy eggs'),
    Task(name: 'Buy bread'),
  ];
```

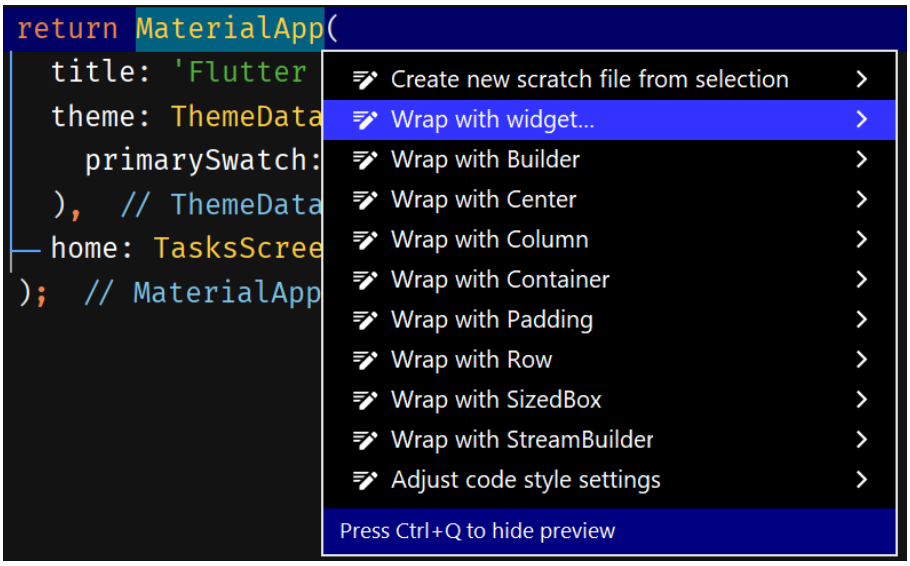
Karena kita perlu memanggil notifyListeners() untuk setiap perubahan _tasks maka buatlah fungsi untuk menambah, mengubah, dan menghapus isi dari _tasks.

```
void addTask(String newTaskTitle) {
  final task = Task(name: newTaskTitle);
  _tasks.add(task);
  notifyListeners();
}

void updateTask(Task task) {
  task.toggleDone();
  notifyListeners();
}

void deleteTask(Task task) {
  _tasks.remove(task);
  notifyListeners();
}
```

Lalu untuk mencegah penambahan task secara langsung. Kita masukkan ke dalam UnmodifiableListView. Dan tambahkan juga fungsi taskCount untuk mengambil jumlah tasks.

```
UnmodifiableListView<Task> get tasks {
  return UnmodifiableListView(_tasks);
}

int get taskCount {
  return _tasks.length;
}
```

Lalu menuju main.dart, pada MaterialApp tekan alt+enter. Lalu, pilih wrap with widget...



Ketik ChangeNotifierProvider lalu isi parameter create dengan TaskData() yang kita buat tadi.

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

  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider(
      create: (BuildContext context) => TaskData(),
      child: MaterialApp(
        home: TasksScreen(),
      ),
    );
  }
}
```

Maka, state sudah dinaikkan ke atas. Cara mengakses state ini adalah dengan menggunakan Provider.of(context). Pertama-tama kita perbaiki terlebih dahulu bagian AddTaskScreen. Hapus tasks dari property AddTaskScreen.

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

Jangan lupa untuk menghapus parameter di TaskScreen juga.

```
child: AddTaskScreen(),
```

Lalu dibagian TextButton pada onPressed. Panggil fungsi addTask di TaskData menggunakan Provider.of(context). Tambah juga parameter listen: false karena kita tidak mengakses property state, hanya mengakses fungsinya.

```
TextButton(
  onPressed: () {
    Provider.of<TaskData>(context, listen: false)
        .addTask(newTaskTitle);
    Navigator.pop(context);
  },
```

Kita pindah ke TaskScreen, ubah text jumlah task menjadi seperti berikut.

```
Text(
  '${Provider.of<TaskData>(context).taskCount} Tasks',
  style: const TextStyle(
    color: Colors.white,
    fontSize: 18,
  ),
),
```

Consumer

Jika anda menggunakan Provider.of di banyak tempat pada sebuah widget maka widget anda bisa di wrap (kurung) dengan widget Consumer yang menyediakan data Provider.of menjadi sebuah variable.

Kita akan Memperbaiki ListTile agar menggunakan TaskData. Pertama-tama ubah TaskScreen menjadi Stateless Widget dan hapus List _tasks.

```
class TasksScreen extends StatelessWidget {

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

Dan hapus parameter task di TaskList.

```
child: TasksList(),
```

Lalu kita pindah ke TaskList, ubah TaskList menjadi Stateless Widget dan hapus List _tasks.

```
class TasksList extends StatelessWidget {

  const TasksList({super.key});

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

Wrap ListView.builder dengan widget Consumer dengan tipe data TaskData, lalu gunakan parameter builder seperti berikut.

```
return Consumer<TaskData>(
  builder: (context, taskData, child){

  },
```

Letakkan parameter child tersebut ke dalam builder.

```
return Consumer<TaskData>(
  builder: (context, taskData, child){
    return ListView.builder(
      itemBuilder: (context, index) {
        final task = widget.tasks[index];
        return TaskTile(
          taskTitle: task.name,
          isChecked: task.isDone,
          checkboxCallback: (checkboxState) {
            setState(() {
              widget.tasks[index].toggleDone();
            });
          },
          longPressCallback: () {
            setState(() {
              widget.tasks.removeAt(index);
            });
          },
        );
      },
      itemCount: widget.tasks.length,
    );
  },
);
```

ubah widget.tasks[index] menjadi taskData.tasks

```
final task = taskData.tasks[index];
```

ubah checkboxCallback menjadi memanggil fungsi updateTask di TaskData

```
checkboxCallback: (checkboxState) {
  taskData.updateTask(task);
},
```

ubah longPressCallback menjadi memanggil fungsi deleteTask di TaskData

```
longPressCallback: () {
  taskData.deleteTask(task);
},
```

Terakhir, ubah itemCount menjadi taskData.taskCount.

```
itemCount: taskData.taskCount,
```

MultiProvider

Jika Anda memiliki lebih dari satu provider dalam satu project, anda bisa menggunakan MultiProvider untuk mendeklarasikannya di atas MaterialApp.

Kita ingin menerapkan fungsi DarkMode/LightMode yang ada di SettingsScreen. Pertama-tama buat file theme_mode_data dan isi seperti berikut.

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

class ThemeModeData extends ChangeNotifier {
  ThemeMode _themeMode = ThemeMode.system;

  ThemeMode get themeMode => _themeMode;

  bool get isDarkModeActive => _themeMode == ThemeMode.dark;

  void changeTheme(ThemeMode themeMode) {
    _themeMode = themeMode;
    notifyListeners();
  }
}
```

Kemudian kita pindah ke main.dart. Ubah ChangeNotifierProvider menjadi MultiProvider dan isi parameter providers dengan TaskData dan ThemeModeData. Lalu isi parameter themeMode pada MaterialApp dengan ThemeModeData. Tambah juga darkTheme agar terlihat perubahan Tema nantinya.

```
return MultiProvider(
  providers: [
    ChangeNotifierProvider(
      create: (BuildContext context) => TaskData(),
    ),
    ChangeNotifierProvider(
      create: (BuildContext context) => ThemeModeData(),
    )
  ],
  child: MaterialApp(
    themeMode: Provider.of<ThemeModeData>(context).themeMode,
```

```
    darkTheme: ThemeData.dark(useMaterial3: true),
    home: TasksScreen(),
  ),
);
```

Lalu wrap MaterialApp dengan Widget Builder.

```
child: Builder(
  builder: (context) {
    return MaterialApp(
      themeMode: Provider.of<ThemeModeData>(context).themeMode,
      darkTheme: ThemeData.dark(useMaterial3: true),
      home: TasksScreen(),
    );
  }
),
```

Pindah ke settings_screen.dart, ubah SettingsScreen menjadi Stateless widget dan hapus property isDarkModeActive dan fungsi changeTheme.

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

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

Ubah pada bagian icon isDarkModeActive menjadi menggunakan ThemeModeData.

```
leading: Icon(
  Provider.of<ThemeModeData>(context).isDarkModeActive
      ? Icons.dark_mode
      : Icons.light_mode,
```

Ubah pada bagian Switch untuk menggunakan ThemeModeData.

```
trailing: Switch(
  value: Provider.of<ThemeModeData>(context).isDarkModeActive,
  onChanged: (bool value) {
    Provider.of<ThemeModeData>(context, listen: false).changeTheme(
      value ?  ThemeMode.dark: ThemeMode.light,
    );
  },
),
```

Terakhir ubah OnTap pada ListTile untuk Menggunakan ThemeModeData.

```
onTap: () {
  Provider.of<ThemeModeData>(context, listen: false).changeTheme(
    Provider.of<ThemeModeData>(context, listen: false).isDarkModeActive
```

```
          ? ThemeMode.light
          : ThemeMode.dark,
    );
},
```

Materi Tambahan

Ada cara yang lebih baik untuk mengakses state yang disediakan provider. Yaitu Menggunakan Context.read<>(), Context.watch<>(), Context.select<,>()

Context.read<>(),

Cara ini digunakan untuk mengakses state tanpa mendegarkan perubahan yang terjadi pada state tersebut. Cara ini biasa digunakan untuk mengakses fungsi pengubah state. Contoh:

```
context.read<ThemeModeData>().changeTheme(ThemeMode.system);
```

Context.watch<>(),

Cara ini digunakan untuk mengakses state dan mendegarkan perubahan yang terjadi pada state tersebut. Cara ini biasa digunakan untuk mengakses property state tanpa mengaksesnya lebih dalam. Contoh:

```
context.watch<ThemeModeData>().themeMode
```

Context.select<,>()

Cara ini digunakan untuk mengakses state dan mendegarkan perubahan yang terjadi pada property tertentu saja. Cara ini biasa digunakan untuk mengakses property state dan mengaksesnya lebih dalam. Contoh:

Misalnya kita memiliki state seperti berikut:

```
class AccessServices extends ChangeNotifier {
  UserModel _userModel = UserModel.initial();
```

Dengan UserModel seperti ini:

```
class UserModel{
  final String uid;
  String username;
  String image;
  String email;
```

Maka cara mengakses uid dari userModel adalah seperti berikut:

```
context.select<AccessServices, String>(
    (value) => value.userModel.uid)
```

Daftar Pustaka

https://morioh.com/p/5155e9e21cf8

https://www.barajacoding.or.id/lifecycle-pada-aplikasi-flutter/

https://www.udemy.com/course/flutter-bootcamp-with-dart/