Flutter Spickzettel (Stromzählerapp)

Table of Contents

1) App-Start & Grundgerüst . . . 1
2) Stateless vs. Stateful . . . 2
3) Build-Methode (Wichtig) . . . 3
4) Scaffold& Layout . . . 3
Scaffold: . . . 4
Layout: . . . 4
5) Navigation (Seiten wischen) . . . 5
6) State & Lifecycle . . . 5
7) SharedPreferences (lokale Speicherung) . . . 6
8) Fortschritt (LinearProgressIndicator) . . . 6
9) Strings & Zahlen (kurz) . . . 7
10) `const`, `final`, `late` . . . 7
11) Farben & Theme . . . 7
12) Typische Stolperfallen (kurz) . . . 7
13) Kleines Aufräumen . . . 7
14) Hot Reload vs. Hot Restart . . . 8
15) Mini-Pattern (Trackingelement – komplett) . . . 8
Quellen: . . . 9

Kurzer, praxisnaher Überblick mit den Dingen, die ich im Projekt wirklich benutze.

1) App-Start & Grundgerüst

```
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.grey,
        scaffoldBackgroundColor: const Color(0xFF292929),
      ),
```

```
      home: const MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}
```

2) Stateless vs. Stateful

- **StatelessWidget**: Keine Eigen-Daten, die sich zur Laufzeit ändern.
- Bekommt alles über Konstrukter-Parameter
- Beispiele : Labels, Icons, reine Layout-Container, Buttons mit Callback

```
class Headline extends StatelessWidget {
  const Headline(this.text, {super.key});
  final String text;

  @override
  Widget build(BuildContext context) {
    return Text(text, style: const TextStyle(fontSize: 24));
  }
}
```

- **StatefulWidget**: hat eigenen Zustand(state), der sich ändern kann.
- Zustand in State-Klasse, nicht im Widget
- UI aktualisieren mit **setState(){….})**
- Typische Fälle: Eingaben, Animationen, Controller (Page/Text/Tab), Timer
- Beispiel (Counter):

```
class Counter extends StatefulWidget {
  const Counter({super.key});
  @override
  State<Counter> createState() => _CounterState();
}

class _CounterState extends State<Counter> {
  int _count = 0;

  void _inc() => setState(() => _count++);

  @override
  Widget build(BuildContext context) {
    return InkWell(
      onTap: _inc,
      child: Text('$_count', style: const TextStyle(fontSize: 32)),
    );
  }
```

```
}
```

```
Counter (StatefulWidget) → createState() → _CounterState (State)
                                 |
                                 └──▯ build() liest _count → Text('$_count')

Tap auf InkWell → _inc() → setState() → build() läuft erneut
```

3) Build-Methode (Wichtig)

- Beschreibt den Teil der Benutzeroberfläche, der durch dieses Widget dargestellt wird.
- Das Framework ruft diese Methode in verschiedenen Situationen auf. Beispiel:
  - Nach dem Aufruf von *initState* .
  - Nach dem Aufruf von *didUpdateWidget* .
  - Nach Erhalt eines Aufrufs von *setState* .
  - Nachdem sich eine Abhängigkeit dieses Statusobjekts geändert hat (z. B. ein InheritedWidget , auf das durch die vorherigen Build- Änderungen verwiesen wird).
  - Nach dem Aufruf von disable und anschließendem erneuten Einfügen des State- Objekts in den Baum an einer anderen Stelle.
  - Liefert den Widget-Tree.

```
class MyButton extends StatefulWidget {
  const MyButton({super.key, this.color = Colors.teal});

  final Color color;
  // ...
}

class MyButtonState extends State<MyButton> {
  // ...
  @override
  Widget build(BuildContext context) {
    return SpecialWidget(
      handler: () { print('color: ${widget.color}'); },
    );
  }
}
```

4) Scaffold& Layout

Scaffold:

- Grundgerüst der Seite (AppBar, Body, FAB, ...).
- immer Zentral als Top-Level-Container einer MaterialApp implementiert.

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

MaterialApp(
  home: Scaffold(
    appBar: AppBar(title: const Text('Meine App')),
    body: const Center(child: Text('Hallo Welt')),
    floatingActionButton: FloatingActionButton(
      onPressed: () {},
      child: const Icon(Icons.add),
    ),
  ),
);
```

Layout:

- 1. Wählen von einem Layout-Widget: https://docs.flutter.dev/ui/widgets/layout

```
Center(
  // Content to be centered here.
)
```

- 2 Erstellen von sichtbaren Widget: https://docs.flutter.dev/ui/widgets

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

  - Text : https://api.flutter.dev/flutter/widgets/Text-class.html
  - Bilder : https://api.flutter.dev/flutter/widgets/Image-class.html
  - icons : https://api.flutter.dev/flutter/material/Icons-class.html
- 3 sichtbares Widget zum Layout-Widget hinzufügen:

```
const Center(
  child: Text('Hello World'),
),
```

- Hinweise:
  - child = genau ein Kind → „Wrapper“, die Verhalten/Aussehen ändern (zentrieren, polstern, ausrichten, klickbar machen).

```
Center(
  child: Padding(
    padding: const EdgeInsets.all(16),
    child: Text('Hello World'),
  ),
)
```

- children = mehrere Kinder → „Layout-Container“, die positionieren (nebeneinander, untereinander, übereinander, scrollend).

```
Row( //horizontal
  mainAxisAlignment: MainAxisAlignment.spaceBetween,
  children: const [
    Icon(Icons.flash_on),
    Text('Strom'),
    Icon(Icons.chevron_right),
  ],
)
```

- Kombination:

```
Padding(                        // Single-Child
  padding: const EdgeInsets.all(16),
  child: Row(                   // Multi-Child
    children: const [
      Icon(Icons.bolt),
      SizedBox(width: 8),
      Text('Leistung'),
    ],
  ),
);
```

5) Navigation (Seiten wischen)

- `PageView` + `PageController`

```
final controller = PageController(initialPage: 0);
PageView(controller: controller, children: const [DetailPage(...), DetailPage(...)] )
```

6) State & Lifecycle

- `initState()` **nicht** `async` machen. Async-Arbeit in Helper auslagern.
- Nach `await` ggf. `if (!mounted) return;` vor `setState`.

```
@override
void initState() {
  super.initState();
  _load(); // async Helper
}
```

7) SharedPreferences (lokale Speicherung)

- Einmal holen:

```
final Future<SharedPreferences> _prefs = SharedPreferences.getInstance();
```

- Laden (Helper):

```
Future<void> _load() async {
  final prefs = await _prefs;
  if (!mounted) return;
  setState(() => _counter = prefs.getInt(_storageKey) ?? 0);
}
```

- Speichern:

```
Future<void> _save() async {
  (await _prefs).setInt(_storageKey, _counter);
}
```

- Stabiler Key (z. B. pro Tag & Einheit):

```
late final String _storageKey;
@override
void initState() {
  super.initState();
  final now = DateTime.now();
  _storageKey = '${now.year}-${now.month}-${now.day}-${widget.unit}';
  _load();
}
```

8) Fortschritt (LinearProgressIndicator)

- `value` muss zwischen `0.0` und `1.0` liegen.

```
double get _progress => (_counter / widget.max).clamp(0.0, 1.0);
```

```
LinearProgressIndicator(value: _progress, minHeight: 12);
```

9) Strings & Zahlen (kurz)

- String-Interpolation statt `+`:

```
Text('$_counter / ${widget.max.toInt()} ${widget.unit}');
```

- Begrenzen:

```
_counter = (_counter + 200).clamp(0, widget.max.toInt());
```

10) `const`, `final`, `late`

- `const`: compile-time konstant (auch bei Widgets → Performance).
- `final`: einmal zuweisen, dann fix (runtime).
- `late final`: später genau einmal setzen (z. B. in `initState`).

11) Farben & Theme

- Vordefiniert: `Colors.grey`, `Colors.white`.
- Hex-ARGB: `const Color(0xFF292929)` (FF = volle Deckkraft).
- `primarySwatch` erwartet **MaterialColor** (z. B. `Colors.grey`).
- Moderner Weg (optional): `ThemeData(colorScheme: ColorScheme.fromSeed(…))`.

12) Typische Stolperfallen (kurz)

- `initState` **nie** `async` → async-Helper nutzen.
- Nicht `SharedPreferences` mit `SharedPreferencesWithCache` mischen.
- Nicht `Color(Colors.white)` schreiben → `Colors.white` ist schon `Color`.
- Progress immer 0..1 (sonst Fehler/keine Anzeige).

13) Kleines Aufräumen

- `import 'dart:ffi';` brauche ich hier nicht → entfernen.
- Klassen-Namen: PascalCase (`DetailPage` statt `detailPage`) – Stilfrage, aber sauber.

14) Hot Reload vs. Hot Restart

- **Hot Reload** (Blitz): Code injizieren, State bleibt (schnell).
- **Hot Restart** (): App neu starten, State weg (nötig bei `initState`/`late`/Theme-Änderungen).

15) Mini-Pattern (Trackingelement – komplett)

```
class Trackingelement extends StatefulWidget {
  const Trackingelement({
    super.key,
    required this.color,
    required this.iconData,
    required this.unit,
    required this.max,
  });
  final Color color;
  final IconData iconData;
  final String unit;
  final double max;

  @override
  State<Trackingelement> createState() => _TrackingelementState();
}

class _TrackingelementState extends State<Trackingelement> {
  final Future<SharedPreferences> _prefs = SharedPreferences.getInstance();
  int _counter = 0;
  late final String _storageKey;

  double get _progress => (_counter / widget.max).clamp(0.0, 1.0);

  @override
  void initState() {
    super.initState();
    final now = DateTime.now();
    _storageKey = '${now.year}-${now.month}-${now.day}-${widget.unit}';
    _load();
  }

  Future<void> _load() async {
    final prefs = await _prefs;
    if (!mounted) return;
    setState(() => _counter = prefs.getInt(_storageKey) ?? 0);
  }

  Future<void> _incrementCounter() async {
```

```
    setState(() => _counter = (_counter + 200).clamp(0, widget.max.toInt()));
    (await _prefs).setInt(_storageKey, _counter);
  }

  @override
  Widget build(BuildContext context) {
    return InkWell(
      onTap: _incrementCounter,
      child: Column(
        children: [
          Padding(
            padding: const EdgeInsetsDirectional.fromSTEB(32, 64, 32, 0),
            child: Row(
              children: [
                Icon(widget.iconData, color: Colors.white, size: 50),
                const SizedBox(width: 12),
                Text('$_counter / ${widget.max.toInt()} ${widget.unit}',
                  style: const TextStyle(color: Colors.white, fontSize: 35)),
              ],
            ),
          ),
          LinearProgressIndicator(value: _progress, color: widget.color, minHeight:
12),
        ],
      ),
    );
  }
}
```

Quellen:

- api.flutter.dev
- flutter.de
- https://docs.flutter.dev/