Eine vorteilhafte Code-Struktur für Flutter-Apps

Autor

Steffen Nowacki · PartMaster GmbH · www.partmaster.de

Abstract

Hier wird eine Code-Struktur vorgestellt, die durch die Anwendung von Entwurfsmustern [1] die Trennung der Verantwortlichkeiten [2] im Code von App-Projekten mit Flutter [3] befördert. Für die Code-Struktur werden die Bausteine Binder, Builder und Props sowie AppState, Reducer und Reduceable verwendet, die im Folgenden erklärt werden. Die Code-Struktur ist gut testbar, skalierbar und kompatibel zu verbreiteten App-Zustands-Verwaltungs-Lösungen, wie Riverpod [4] oder Bloc [5]. Und sie kann auch für eigene Lösungen auf Basis der eingebauten Flutter-Mittel StatefulWidget und InheritedWidget eingesetzt werden.
Wer seine Code-Struktur übersichtlicher gestalten will oder wer bei der Wahl einer App-Zustands-Verwaltungs-Lösung flexibel sein will, für den könnte der Artikel interessant sein.

Einleitung

Source Code scheint dem 2. Gesetz der Thermodynamik zu folgen und zur Aufrechterhaltung der Ordnung der ständigen Zuführung von äußerer Energie zu bedürfen. Flutter-App-Projekte sind da keine Ausnahme. Ein typisches Symptom sind build-Methoden mit wachsenden Widget-Konstruktor-Hierarchien, die von App-Logik infiltriert werden. Mit App-Logik meine ich die Fachlogik der UI und ihres Zustands im engeren Sinn - in Abgrenzung zur Fachlogik einer Anwendungsdomäne, die oft in Datenbanken, Backends oder externen Bibliotheken implementiert ist. Viele Flutter-Frameworks wurden und werden entwickelt, um eine saubere Code-Struktur zu unterstützen. Dabei geht es hauptsächlich um das Verwalten des Zustandes der App auf eine Art und Weise, die eine Trennung der Verantwortlichkeiten zwischen App-Logik und UI ermöglicht.
Bei einem unbedachten Einsatz solcher Frameworks besteht die Gefahr, dass sie neben ihrer eigentlichen Aufgabe, der Trennung der Verantwortlichkeiten, die App-Logik und die UI infiltrieren und unerwünschte Abhängigkeiten schaffen. Weil es viele Frameworks gibt (in der offiziellen Flutter-Dokumentation sind aktuell 13 Frameworks gelistet [6]) und die Entwicklung

sicher noch nicht abgeschlossen ist, kann es besonders für große und langlebige App-Projekte zur Herausforderung werden, zwischen Frameworks migrieren oder verschiedene Frameworks integrieren zu müssen.

Im Folgenden wird eine Code-Struktur für Flutter-Apps vorgestellt, die solche unerwünschten Infiltrationen vermeidet und so die Qualität von App-Logik- und UI-Code verbessert. Dabei geht es ausdrücklich nicht um die Einführung eines weiteren Frameworks sondern um die abgestimmte Anwendung von zwei Entwurfsmustern, Humble Object Pattern [7] und Reducer Pattern [8], auf den Flutter-App-Code.
Flutter beschreibt sich selbst mit dem Spruch "Alles ist ein Widget" [9]. Damit ist gemeint, dass alle Features in Form von Widget-Klassen implementiert sind, die sich wie Lego-Bausteine aufeinander stecken lassen. Das ist eine großartige Eigenschaft mit einer kleinen Kehrseite: Wenn man nicht aufpasst, vermischen sich in den resultierenden Widget-Bäumen schnell die Verantwortlichkeiten.

Die Verantwortlichkeiten um die es in diesem Dokument geht, sind einerseits die klassischen UI-Aufgaben:

1. Layout,
2. Rendering,
3. Gestenerkennung

und andererseits Aufgaben die sich auf den App-Zustand beziehen:

1. Lauschen auf App-Zustands-Änderungen,
2. Konvertierung des App-Zustandes in Anzeige-Properties,
3. Abbildung von Gesten auf App-Zustands-Operationen.

Die UI-Aufgaben sind eng an eine Umgebung gebunden, in der User Interfaces ablaufen können, wohingegen die Logik in den App-Zustands-Aufgaben nicht unbedingt an eine UI-Ablaufumgebung gebunden ist.

Builder, Binder und Props

Das erste Ziel ist, den Flutter-Code so zu strukturieren, dass im Widget-Baum die UI-Aufgaben streng von den App-Zustands-Aufgaben separiert werden. Um das zu erreichen, wird das Humble Object Pattern von Micheal Feathers [10] angewandt. Die Zusammenfassung des Humble Object Pattern lautet:

> Wenn Code nicht gut testbar ist, weil er zu eng mit seiner Umgebung verbunden ist,

> extrahiere die Logik in eine separate, leicht zu testende Komponente, die von ihrer Umgebung entkoppelt ist.

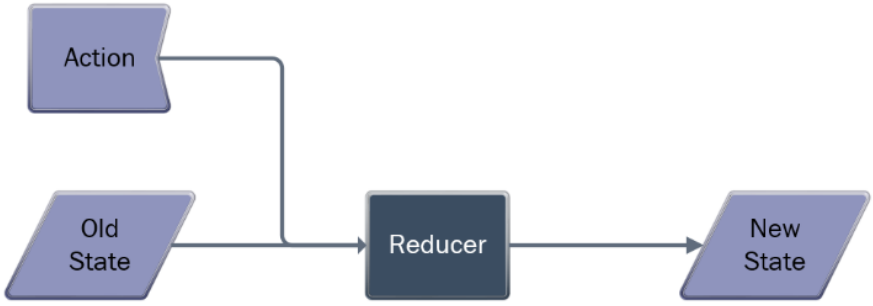
Auf eine Widget-Klasse bezogen heißt das:

1. Wenn die Widget-Klasse sowohl UI-Aufgaben als auch App-Zustands-Aufgaben löst, dann wird diese Widget-Klasse in eine Builder-Klasse und eine Binder-Klasse geteilt.
2. Die Binder-Klasse lässt sich über App-Zustandsänderungen benachrichtigen, konvertiert den aktuellen App-Zustand in die Properties für die Builder-Klassse, stellt für die Nutzergesten die Callback-Objekte mit den App-Zustands-Operationen zur Verfügung und liefert in der build-Methode ein Widget der Builder-Klasse zurück.
3. Die Builder-Klasse ist ein StatelessWidget. Sie bekommt von der Binder-Klasse im Konstruktor die vorkonfektionierten Properties und Callbacks und erzeugt in der build-Methode einen Widget-Baum aus Layout-, Renderer und Gestenerkennungs-Widgets.
4. Für die vorkonfektionierten Properties und Callbacks wird eine Props-Klasse definiert - eine reine Datenklasse mit ausschließlich finalen Feldern und einem const Konstruktor.

AppState, Reducer und Reduceable

Das zweite Ziel ist, den Code so zu strukturieren, dass die App-Logik streng vom eingesetzten Zustands-Verwaltungs-Framework und von Flutter insgesamt separiert wird. Um das zu erreichen, wird das Reducer Pattern angewandt. Das Reducer Pattern modelliert den App-Zustand als Ergebnis einer Faltungsfunktion [11] des initialen App-Zustands und der Folge der bisherigen App-Zustands-Änderungs-Aktionen. Dan Abramov und Andrew Clark haben dieses Konzept im Framework Redux [12] verwendet und für den Kombinierungsoperator, der aus dem aktuellen App-Zustand und einer Aktion einen neuen App-Zustand berechnet, den Namen *Reducer* populär gemacht:

> Reducers sind Funktionen, die den aktuellen Zustand und eine Aktion als Argumente nehmen und ein neues Zustandsergebnis zurückgeben. Mit anderen Worten: `(state, action) => newState` [^8].



Auf App-Code bezogen heißt das:

1. Für den App-Zustand wird eine AppState-Klasse definiert - eine reine Datenklasse mit ausschließlich finalen Feldern und einem const Konstruktor. Die App-Zustands-Verwaltung stellt eine get-Methode für den aktuellen App-Zustand zur Verfügung.

2. Die App-Zustands-Verwaltung stellt eine reduce-Methode zur Verfügung, die einen Reducer als Parameter akzeptiert. Ein Reducer ist eine reine synchrone Funktion, die eine Instanz der AppState-Klasse als Parameter bekommt und eine neue Instanz der AppState-Klasse als zurückgibt. Beim Aufruf führt die reduce-Methode den übergebenen Reducer mit dem aktuellen App-Zustand als Parameter aus und speichert den Rückgabewert des Reducer-Aufrufs als neuen App-Zustand ab.

3. Die App-Zustands-Verwaltung stellt eine Möglichkeit zur Verfügung, sich über Zustandsänderungen benachrichtigen zu lassen. In der minimalen Variante recht es aus, wenn als Benachrichtigung in einem Widget ein setState oder markNeedsBuild ausgelöst wird. Diese Benachrichtigung sollte auch selektiv nur für ausgesuchte Änderungen möglich sein.

Für die ersten beiden Anforderungen lässt sich leicht eine minimale Schnittstelle definieren:

1. eine get-Methode für den App-Zustand

2. eine reduce-Methode zum Ändern des App-Zustands

```
abstract class Reduceable<S> {
  S get state;
  Reduce<S> get reduce;
}

typedef Reduce<S> = void Function(Reducer<S>);

abstract class Reducer<S> {
  S call(S state);
}
```

Die dritte Anforderung ist stark vom eingesetzten App-Zustands-Verwaltungs-Framework abhängig (insbesondere die selektive Benachrichtigung) und wird später für ausgewählte Lösungen (StatefulWidget/InheritedWidget, Riverpod, Bloc) diskutiert.

Mit Hilfe des vorgestellten Konzepts mit den Klassen AppState, Reducer, Reduceable sollte es möglich sein, die App-Logik komplett vom ausgewählten Zustands-Verwaltungs-Framework zu entkoppeln. Die App-Logik wird hauptsächlich in Form von verschiedenen Reducer-Implementierungen bereitgestellt. Hinzu kommen Konverter, die aus einem Reduceable und den Reducern die verschiedenen Props-Klassen für die Builder-Widgets aus dem vorherigen Kapitel und für die selektiven Benachrichtigungen aus diesem Kapitel konstruieren können.

Tutorial

Nun soll die vorgestellte Code-Struktur an einem kleinen Beispiel illustriert werden.

Ausgangslage

Als Vorlage wird das wohlbekannte Flutter-Counter-App-Projekt verwendet. In diesem Projekt spielt die Klasse _MyHomePageState die zentrale Rolle:

```
class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

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

Die Klasse _MyHomePageState trägt die verschiedensten Verantwortungen:

1. Layout

```
          mainAxisAlignment: MainAxisAlignment.center,
```

2. Rendering

```
            style: Theme.of(context).textTheme.headline4,
```

3. Gestenerkennng

```
        onPressed:
```

4. App-Zustands-Speicherung

```
  int _counter = 0;
```

5. Bereitstellung von Operationen für App-Zustands-Änderungen

```
  void _incrementCounter() {
```

6. Widget-Benachrichtigung nach App-Zustands-Änderungen

```
    setState(() {
```

7. Konvertierung des App-Zustands in Anzeige-Properties

```
              '$_counter',
```

8. Abbildung von Gesten-Callbacks auf App-Zustands-Änderungens-Operationen

```
        onPressed: _incrementCounter,
```

Props

Wir wollen nun die Code-Struktur von `_MyHomePageState` verbessern und beginnen mit der Definition einer neuen Klasse, der Props. Die Props sind eine Datenklasse für eine korrespondierende Widget-Klasse, die alle in der build-Methode der Widget-Klasse zur Erstellung des Widget-Baums benötigten Properties enthält. Die Property-Namen in der Props-Klasse werden so gewählt, dass eine leichte Zuordnung zu den Properties im Widget-Baum, denen sie zugewiesen werden sollen, möglich ist. Der Name *Props* ist eine in React [13] übliche Abkürzung für die Properties von StatelessWidgets und die unveränderlichen Properties (im Gegensatz zu veränderbaren Status-Properties) von StatefulWidgets.
Für die Klasse `_MyHomePageState` könnte die zugehörige Props-Klasse so aussehen:

```
class MyHomePageProps {
  final String title;
  final String counterText;
  final Callable<void> onIncrementPressed;

  MyHomePageProps({
    required this.title,
    required this.counterText,
    required this.onIncrementPressed,
  });

  MyHomePageProps.reduceable(Reduceable<MyAppState> reduceable)
      : title = reduceable.state.title,
        counterText = '${reduceable.state.counter}',
        onIncrementPressed = VoidCallable(
          reduceable,
          IncrementCounterReducer(),
        );

  @override
  int get hashCode => hash3(title, counterText, onIncrementPressed);

  @override
  bool operator ==(Object other) =>
      other is MyHomePageProps &&
      title == other.title &&
      counterText == other.counterText &&
      onIncrementPressed == other.onIncrementPressed;
}
```

Um aus dem App-Zustand einen Props-Wert zu erzeugen, bedarf es einer Konvertierung. Diese Konvertierung ist in Form des Props-Konstruktors `MyHomePageProps.reduceable` implementiert. In diesem Konvertierungs-Konstruktor wird festgelegt, wie App-Zustands-Werte für die Anzeige kombiniert bzw. formatiert werden und auf welche App-Zustands-Operationen die erkannten Nutzergesten in den Nutzergesten-Callbacks abgebildet werden.
Damit die Props als Wert für selektive Benachrichtigungen über App-Zustands-Änderungen eingesetzt werden können, müssen die hashCode und operator== Methoden der Props-Klasse nach Wertsemantik [14] funktionieren. Das setzt voraus, dass diese Methoden bei allen Properties ebenfalls nach Wertsemantik funktionieren. Da mir nicht ganz klar ist, wie dies bei Funktionsobjekten gewährleistet werden kann, habe ich für das Callback-Property nicht den Standard-Flutter-Typ VoidCallback verwendet, sondern eine eigene Klasse `VoidCallable` mit überschriebenen hashCode und operator== Methoden.

```
abstract class Callable<T> {
  T call();
}

class VoidCallable<S> extends Callable<void> {
  VoidCallable(this.reduceable, this.reducer);

  final Reduceable<S> reduceable;
  final Reducer<S> reducer;

  @override
  void call() => reduceable.reduce(reducer);

  @override
  int get hashCode => hash2(reduceable, reducer);

  @override
  bool operator ==(Object other) =>
      other is VoidCallable<S> &&
      reducer == other.reducer &&
      reduceable == other.reduceable;
}
```

Builder

Die Builder-Klasse ist für den Bau des Widget-Baums aus Layout-, Rendering- und Gestenerkennungs-Widgets verantwortlich. Dazu verwendet sie eine Instanz vom Typ der korrespondierenden Props-Klasse, die ihr im Konstruktor übergeben wird. Mit Hilfe der vorkonfektionierten Properties in den Props lässt sich nun leicht aus der _MyHomePageState-Klasse eine stateless Builder-Klasse für den Widget-Baum extrahieren.

```
class MyHomePageBuilder extends StatelessWidget {
  const MyHomePageBuilder({super.key, required this.props});

  final MyHomePageProps props;

  @override
  Widget build(context) => Scaffold(
        appBar: AppBar(
          title: Text(props.title),
        ),
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              const Text(
                'You have pushed the button this many times:',
              ),
              Text(
                props.counterText,
                style: Theme.of(context).textTheme.headline4,
              ),
            ],
          ),
        ),
        floatingActionButton: FloatingActionButton(
          onPressed: props.onIncrementPressed,
          tooltip: 'Increment',
          child: const Icon(Icons.add),
        ),
      );
}
```

Binder

Zu jeder Builder-Klasse gibt es eine korrespondierende Binder-Klasse. Die Binder-Klasse ist für die Bindung an den App-Zustand verantwortlich. Sie muss dafür sorgen, dass bei einer Änderung des App-Zustandes neue Props erzeugt werden und mit diesen Props eine neue Instanz der Builder-Klasse gebaut wird. Wie sie das umsetzt, hängt vom verwendeten App-Zustands-Verwaltungs-Framework ab. Für das Beispiel nutzen wir für den Anfang kein externes Framework wie Riverpod oder Bloc, sondern nur StatefulWidget und InheritedWidget. Die in der build-Methode von `MyHomePageBinder` verwendete InheritedValueWidget-Klasse, eine Ableitung der InheritedWidget-Klasse, wird im Abschnitt *App-Zustandsverwaltung* vorgestellt.

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

  @override
  Widget build(context) => MyHomePageBuilder(
        props: InheritedValueWidget.of<MyHomePageProps>(context),
      );
}
```

Die Binder-Klasse setzt also in der abgebildeten Implementierung voraus, dass das InheritedWidget `InheritedValueWidget` die aktuellen Props bereitstellt und dass ein StatefulWidget das InheritedWidget mit dem aktuellen App-Zustand versorgt. Unter diesen Voraussetzungen ist die Binder-Klasse seht einfach.

AppState

Der AppState beinhaltet das Property `counter`. Da im Flutter-Counter-App-Projekt auch der `title` von außen an das _MyHomePageState-Widget übergeben wird, habe ich es ebenfalls in den AppState mit aufgenommen, obwohl der `title` nie geändert wird.

```
class MyAppState {
  const MyAppState({required this.title, required this.counter});

  final String title;
  final int counter;

  MyAppState copyWith({String? title, int? counter}) => MyAppState(
        title: title ?? this.title,
        counter: counter ?? this.counter,
      );

  @override
  int get hashCode => hash2(title, counter);

  @override
  bool operator ==(Object other) =>
      other is MyAppState &&
      title == other.title &&
      counter == other.counter;
}
```

Reducer

Die Klasse _MyHomePageState aus dem Flutter-Counter-App-Projekt hat für Änderungen

am App-Zustand nur die eine Methode `incrementCounter`. Da es sich hier um App-Logik in einer Klasse mit UI-Aufgaben handelt, wird der Code in eine neue Klasse `IncrementCounterReducer` ausgelagert. Von der Basisklasse `Reducer` wird für alle Reducer-Implementierungen die reducer-Methode `call` mit der Signatur `S call(S)` vorgegeben. Als konkrete App-Logik wird in der Methode `call` in der Klasse `IncrementCounterReducer` das Property `counter` im AppState inkrementiert. Einige AppState-Verwaltungs-Frameworks entkoppeln die UI von der direkten Zuordnung eines Ereignisses (meist eine erkannte Geste) zu einer App-Logik-Operation, indem die UI Event-Instanzen versendet, um eine Anforderung über eine Operation am AppState auszulösen. In solchen Frameworks mit Event-Klassen können die Reducer-Instanzen als Events verwendet werden. Es gibt in der hier vorgestellten Code-Struktur trotzdem kein Kopplungsproblem zwischen UI und App-Logik, weil diese Entkopplung schon durch die Callback-Properties in den Props-Klassen erreicht werden. Wegen der potenziellen Verwendung als Events müssen die Reducer-Instanzen Wertsemantik haben. Wenn sichergestellt ist, dass es für eine Reducer-Klasse nur eine Instanz gibt, genügen statt Wertsemantik die geerbten hashCode und operator== Methoden.

```
class IncrementCounterReducer extends Reducer<MyAppState> {
  IncrementCounterReducer._();
  factory IncrementCounterReducer() => instance;

  static final instance = IncrementCounterReducer._();

  @override
  MyAppState call(state) =>
      state.copyWith(counter: state.counter + 1);
}
```

App-Zustandsverwaltung

Die Umsetzung der App-Zustands-Verwaltung für die Beispiel-App ist stark vom verwendeten App-Zustands-Verwaltungs-Framework abhängig. Zunächst soll eine Umsetzung mit StatefulWidget und InheritedWidget, also ohne externes Framework gezeigt werden. Für die Implementierung der App-Zustands-Verwaltung mit einer Kombination aus StatefulWidget und InheritedWidget werden die Hilfsklassen `AppStateBinder` und `InheritedValueWidget` eingeführt.

AppStateBinder

Die Klasse `AppStateBinder` erbt von StatefulWidget und hält im Feld _state den veränderlichen App-Zustand. Sie stellt eine get-state-Methode für den App-Zustand sowie eine reduce-Methode, um den App-Zustand von außen mit einem Reducer verändern zu

können, bereit. Die beiden Methoden `get state` und `reduce` werden in ein `Reduceable` verpackt und sind in der hier vorgestellten Code-Struktur die generelle Schnittstelle der App-Zustands-Verwaltung nach außen. Das `Reduceable` wird von `AppStateBinder` an den im Konstruktor hereingereichten `builder` zum Bau des child-Widgets übergeben. Dieser `builder` wird im später vorgestellten `MyAppStateBinder` dazu genutzt werden, InheritedWidget-Kinder zu erzeugen und sie dabei mit dem `Reduceable` zu versorgen.

```
typedef ReduceableWidgetBuilder<S> = Widget Function(
  Reduceable<S> value,
  Widget child,
);
```

```
class AppStateBinder<S> extends StatefulWidget {
  const AppStateBinder({
    super.key,
    required this.initialState,
    required this.child,
    required this.builder,
  });

  final S initialState;
  final Widget child;
  final ReduceableWidgetBuilder<S> builder;

  @override
  State<AppStateBinder> createState() =>
      _AppStateBinderState<S>(initialState);
}
```

```
class _AppStateBinderState<S> extends State<AppStateBinder<S>> {
  _AppStateBinderState(S initialState) : _state = initialState;

  S _state;

  S getState() => _state;

  late final reduceable = Reduceable(getState, reduce);

  void reduce(Reducer<S> reducer) =>
      setState(() => _state = reducer(_state));

  @override
  Widget build(BuildContext context) => widget.builder(
        reduceable,
        widget.child,
      );
}
```

InheritedValueWidget

Die Klasse `InheritedValueWidget` erbt von InheritedWidget. Sie bekommt im Konstruktor einen Wert `value`, speichert ihn in einem gleichnamigen finalen Feld und stellt diesen Wert nachfolgenden Kind- und Kindeskind-Widgets über eine of-Methode zur Verfügung.

```
class InheritedValueWidget<V> extends InheritedWidget {
  const InheritedValueWidget({
    super.key,
    required super.child,
    required this.value,
  });

  final V value;

  static V of<V>(BuildContext context) =>
      _widgetOf<InheritedValueWidget<V>>(context).value;

  static W _widgetOf<W extends InheritedValueWidget>(
          BuildContext context) =>
      context.dependOnInheritedWidgetOfExactType<W>()!;

  @override
  bool updateShouldNotify(InheritedValueWidget oldWidget) =>
      value != oldWidget.value;
}
```

MyAppStateBinder

Die Klasse `MyAppStateBinder` kapselt in der hier vorgestellten Code-Struktur die App-Zustands-Verwaltung, bei der es sich entweder um ein Framework, wie Riverpod oder Bloc, oder um eine eigene Implementierung handeln kann. Als Beispiel implementieren wir Die Klasse `MyAppStateBinder` mit Hilfe von `AppStateBinder` und `InheritedValueWidget` selbst.

```
class MyAppStateBinder extends StatelessWidget {
  const MyAppStateBinder({super.key, required this.child});

  final MyAppState state = const MyAppState(
    title: 'Flutter Demo Home Page',
    counter: 0,
  );
  final Widget child;

  @override
  Widget build(context) => AppStateBinder(
        initialState: state,
        child: child,
        builder: (value, child) => InheritedValueWidget(
          value: MyHomePageProps.reduceable(value),
          child: InheritedValueWidget(
            value: MyCounterWidgetProps.reduceable(value),
            child: child,
          ),
        ),
      );
}
```

Das `MyAppStateBinder`-Widget bildet die Wurzel der Widget-Hierarchie der App und ist für die Bindung der App-Zustands-Verwaltung an die UI verantwortlich. Es stellt der nachfolgenden Widget-Hierarchie den Zugang zur App-Zustands-Verwaltung in Form des Interfaces `Reduceable` bereit.
In diesem Fall haben wir die App-Zustands-Verwaltung einer Kombination aus StatefulWidget und InheritedWidget selbst implementiert und stellen das Interface `Reduceable`, wie bei InheritedWidgets üblich, mit einer statischen Methode `T of<T>(BuildContext context)` bereit.

main

In der main-Funktion werden Instanzen der Klassen `MyAppStateBinder` für die App-Zustands-Verwaltung und `MyAppBuilder` für die UI erzeugt, miteinander verknüpft und die App gestartet.

```
void main() {
  runApp(const MyAppStateBinder(child: MyAppBuilder()));
}
```

Zugabe

Mit der main-Funktion könnte das Tutorial und dieser Artikel enden. Doch es gibt noch eine Verlängerung, und auf die könnte der Titel des Films "Das Beste kommt zum Schluss" passen:

1. **Skalierbarkeit**
   Um zu zeigen, dass die neue Code-Struktur einfach skalierbar ist, extrahieren wir aus der Klasse `MyHomePageBuilder` eine Klasse `MyCounterBuilder`, die nur die Anzeige des Zählerwerts enthält, und implementieren für beide Klassen individuelle selektive Bindungen an den App-Zustand, so dass die build-Methoden von `MyHomePageBuilder` und `MyCounterBuilder` nur ausgeführt werden, wenn sich ihre Props tatsächlich ändern.

2. **Testbarkeit**
   Um zu zeigen, wie die Trennung der Verantwortlichkeiten im Code für eine Vereinfachung der Tests genutzt werden kann, schreiben wir für die umgestellte Counter-Demo-App einen UI-Test und einen App-Logik-Test.

3. **Portierbarkeit**
   Um zu zeigen, dass die neue Code-Struktur die Abhängigkeit vom verwendeten Framework tatsächlich veringert und sich auf die Binder-Klassen beschränkt, portieren wir die umgestellte Counter-Demo-App von der eigenen App-Zustands-Verwaltungs-Implementierung nacheinander auf die Nutzung der Frameworks Riverpod und Bloc.

Skalierung

Beim bisher erreichten Stand der Umgestaltung der Flutter-Counter-App hat sich eines gegenüber dem Original nicht geändert: Bei jeder Änderung des App-Zustandes wird der Widget-Baum der gesamten App-Seite neu gebaut.
Für komplexere Apps kann dies zu einem Performance-Problem werden. Darum ist es essenziell, bei Änderungen des App-Zustandes nur die Teil-Bäume der App-Seite neu zu bauen, die tatsächlich von der Änderung betroffen sind.
Im Widget-Baum der Flutter-Counter-App-Seite ist die AppBar vom Property `title` abhängig, der FloatingActionButton vom Property `onIncrementPressed` und ein Text-Widget vom Property `counterText`.
Wir wollen nun die App so refaktorisieren, dass bei Änderungen des Properties `counterText` nur noch das betroffene Text-Widget neu erzeugt wird und die anderen Widgets des Widget-Baums der App-Seite weiter genutzt werden. Der Umbau erfolgt in 5 Schritten:

1. **Props**
Dazu extrahieren wir aus der Klasse `MyHomePageProps` das Property `counterText` in eine neue Klasse `MyCounterWidgetProps` und definieren den benannten Konstruktor `MyCounterWidgetProps.reduceable`.

```
class MyHomePageProps {
  final String title;
  final Callable<void> onIncrementPressed;

  MyHomePageProps({
    required this.title,
    required this.onIncrementPressed,
  });

  MyHomePageProps.reduceable(Reduceable<MyAppState> reduceable)
      : title = reduceable.state.title,
        onIncrementPressed = VoidCallable(
          reduceable,
          IncrementCounterReducer(),
        );

  @override
  int get hashCode => hash2(title, onIncrementPressed);

  @override
  bool operator ==(Object other) =>
      other is MyHomePageProps &&
      title == other.title &&
      onIncrementPressed == other.onIncrementPressed;
}
```

```
class MyCounterWidgetProps {
  final String counterText;

  MyCounterWidgetProps({
    required this.counterText,
  });

  MyCounterWidgetProps.reduceable(Reduceable<MyAppState> reduceable)
      : counterText = '${reduceable.state.counter}';

  @override
  int get hashCode => counterText.hashCode;

  @override
  bool operator ==(Object other) =>
      other is MyCounterWidgetProps &&
      counterText == other.counterText;
}
```

2. **Änderungs-Selektor**

In der Klasse `MyAppStateBinder` fügen wir ein `InheritedValueWidget` mit einem Template-Parameter vom Typ `MyCounterWidgetProps` ein.

```
class MyAppStateBinder extends StatelessWidget {
  const MyAppStateBinder({super.key, required this.child});

  final MyAppState state = const MyAppState(
    title: 'Flutter Demo Home Page',
    counter: 0,
  );
  final Widget child;

  @override
  Widget build(context) => AppStateBinder(
        initialState: state,
        child: child,
        builder: (value, child) => InheritedValueWidget(
          value: MyHomePageProps.reduceable(value),
          child: InheritedValueWidget(
            value: MyCounterWidgetProps.reduceable(value),
            child: child,
          ),
        ),
      );
}
```

3. **Builder**

Aus der build-Methode der Klasse `MyHomePageBuilder` extrahieren wir das relevante Text-Widget in eine neue Klasse `MyCounterWidgetBuilder` und verwenden die Klasse `MyCounterWidgetProps` als Konstruktor-Parameter.

```
class MyCounterWidgetBuilder extends StatelessWidget {
  const MyCounterWidgetBuilder({super.key, required this.props});

  final MyCounterWidgetProps props;

  @override
  Widget build(context) => Text(
        props.counterText,
        style: Theme.of(context).textTheme.headline4,
      );
}
```

4. **Binder**

Nach dem Muster von `MyHomePageBinder` erzeugen wir eine neue Klasse

`MyCounterWidgetBinder` und holen ins in deren build-Methode vom `InheritedValueWidget` die `MyCounterWidgetProps` und bauen damit den `MyCounterWidgetBuilder`.

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

  @override
  Widget build(context) => MyCounterWidgetBuilder(
        props: InheritedValueWidget.of<MyCounterWidgetProps>(context),
      );
}
```

5. **Scharfschaltung**

Schließlich ersetzen wir in build-Methode der Klasse `MyHomePageBuilder` das Text-Widget durch `MyCounterWidgetBuilder`.

```
class MyHomePageBuilder extends StatelessWidget {
  const MyHomePageBuilder({super.key, required this.props});

  final MyHomePageProps props;

  @override
  Widget build(context) => Scaffold(
        appBar: AppBar(
          title: Text(props.title),
        ),
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: const <Widget>[
              Text(
                'You have pushed the button this many times:',
              ),
              MyCounterWidgetBinder(),
            ],
          ),
        ),
        floatingActionButton: FloatingActionButton(
          onPressed: props.onIncrementPressed,
          tooltip: 'Increment',
          child: const Icon(Icons.add),
        ),
      );
}
```

Damit ist die Einführung und Nutzung einer neuen Builder-Binder-Props-Klassen-Kombination abgeschlossen und kann genutzt werden.

Test

Die App ist sauber nach UI-Code und App-Logik-Code getrennt. Diese Trennung wollen wir nun auch in den Tests vornehmen.

App-Logik-Tests

In der hier vorgestellten Code-Struktur befindet sich die App-Logik in den call-Methoden der Reducer-Klassen und in den reduceable-Konstruktoren der Props-Klassen. In unserem Beispiel-Projekt sind das die Klassen `IncrementCounterReducer`, `MyHomePageProps` und `MyCounterWidgetProps`. Da diese Klassen keine UI-Ablaufumgebung benötigen, können sie mit einfachen Unit-Tests getestet werden.

Hier ein Beispiel-Test für den call-Methode der Klasse `IncrementCounterReducer`:

```
test('testIncrementCounterReducer', () {
  final objectUnderTest = IncrementCounterReducer();
  final state = objectUnderTest.call(
    const MyAppState(title: '', counter: 0),
  );
  expect(state.counter, equals(1));
});
```

Hier ein Beispiel-Test für den reduceable-Konstruktor der Klasse `MyCounterWidgetProps`:

```
test('testMyCounterWidgetProps', () {
  Reduceable<MyAppState> reduceable = Reduceable(
    () => const MyAppState(counter: 0, title: ''),
    (_) {},
  );
  final objectUnderTest =
      MyCounterWidgetProps.reduceable(reduceable);
  expect(objectUnderTest.counterText, equals('0'));
});
```

Hier ein Beispiel-Test für den reduceable-Konstruktor der Klasse `MyHomePageProps`:

```
  test('testMyHomePageProps', () {
    const title = 'Flutter Demo App';
    final incrementReducer = IncrementCounterReducer();
    final decrementReducer = DecrementCounterReducer();
    final reduceable = Reduceable(
      () => const MyAppState(counter: 0, title: title),
      (_) {},
    );
    final onIncrementPressed =
        VoidCallable(reduceable, incrementReducer);
    final onDecrementPressed =
        VoidCallable(reduceable, decrementReducer);
    final objectUnderTest = MyHomePageProps.reduceable(reduceable);
    final expected = MyHomePageProps(
      title: title,
      onIncrementPressed: onIncrementPressed,
    );
    final withUnexpectedTitle = MyHomePageProps(
      title: '_$title',
      onIncrementPressed: onIncrementPressed,
    );
    final withUnexpectedCallback = MyHomePageProps(
      title: title,
      onIncrementPressed: onDecrementPressed,
    );
    expect(objectUnderTest, equals(expected));
    expect(objectUnderTest, isNot(equals(withUnexpectedTitle)));
    expect(objectUnderTest, isNot(equals(withUnexpectedCallback)));
  });
```

Um den `operator==` testen zu können (notwendig für selektive Änderungsbenachrichtigungen), wurde für den Test `testMyHomePageProps` eine weitere Reducer-Klasse `DecrementCounterReducer` angelegt.

```
class DecrementCounterReducer extends Reducer<MyAppState> {
  DecrementCounterReducer._();
  factory DecrementCounterReducer() => instance;

  static final instance = DecrementCounterReducer._();

  @override
  MyAppState call(state) =>
      state.copyWith(counter: state.counter - 1);
}
```

UI-Tests

Ensprechend der Code-Struktur können wir die Tests in Builder-Tests, Binder-Tests und Tests der kompletten App einteilen:

Builder-Tests

Die Builder-Klassen haben die Verantwortlichkeiten Layout, Rendering und Gestenerkennung und dies sollte sich in den Builder-Tests wiederfinden. Es kann einen gewissen Aufwand verursachen, eine App in den jeweiligen Zustand zu bringen, in welchem Layout, Rendering oder Gestenerkennung getestet werden sollen. Wenn für eine hohe Testabdeckung viele Zustände benötigt werden, ist jede Erleichterung wünschenswert. Mit der hier vorgestellten Code-Struktur ist eine Erleichterung für das Einstellen von Zuständen möglich: Dazu erstellen wir zunächst einige Utility-Klassen:

```
class MyMockProps {
  final MyHomePageProps myHomePageProps;
  final MyCounterWidgetProps counterWidgetProps;

  MyMockProps({
    required String title,
    required Callable<void> onIncrementPressed,
    required String counterText,
  })  : myHomePageProps = MyHomePageProps(
          title: title,
          onIncrementPressed: onIncrementPressed,
        ),
        counterWidgetProps = MyCounterWidgetProps(
          counterText: counterText,
        );
}
```

```
extension MockPropsOnBuildContext on BuildContext {
  MyMockProps get mock => InheritedValueWidget.of<MyMockProps>(this);
}
```

```
class MyMockPropsBinder extends StatelessWidget {
  const MyMockPropsBinder({
    super.key,
    required this.child,
    required this.props,
  });

  final Widget child;
  final MyMockProps props;

  @override
  Widget build(context) => InheritedValueWidget(
        value: props,
        child: child,
      );
}
```

```
class MockCallable extends Callable<void> {
  int count = 0;

  @override
  void call() => ++count;
}
```

Mit den Utility-Klassen erstellen wir Mock-Builder-Klassen, mit denen man die Props für die Builder-Klassen vorgeben kann:

```
class MyAppStateBinder extends StatelessWidget {
  const MyAppStateBinder({super.key, required this.child});

  final Widget child;

  @override
  Widget build(context) => child;
}
```

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

  @override
  Widget build(context) => MyHomePageBuilder(
        props: context.mock.myHomePageProps,
      );
}
```

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

  @override
  Widget build(context) => MyCounterWidgetBuilder(
        props: context.mock.counterWidgetProps,
      );
}
```

Nach diesen Vorbereitungen können wir mit dem MyMockProps-Konstruktor nun einfach Builder-Tests für jeden Zustand gewünschten implementieren. Für die überschaubaren Zustände unserer Beispiel-App mag der Vorbereitungsaufwand übertrieben sein, bei größeren Apps kann er sich lohnen.
Hier nun ein Builder-Test:

```
  testWidgets('testBuilder', (tester) async {
    const title = 'title';
    const counterText = '0';
    final onIncrementPressed = MockCallable();
    final app = MyMockPropsBinder(
      props: MyMockProps(
        title: title,
        counterText: '0',
        onIncrementPressed: onIncrementPressed,
      ),
      child: const MyAppBuilder(),
    );
    await tester.pumpWidget(app);
    expect(find.widgetWithText(AppBar, title), findsOneWidget);
    expect(
      find.widgetWithText(MyCounterWidgetBuilder, counterText),
      findsOneWidget,
    );
    await tester.tap(find.byIcon(Icons.add));
    expect(onIncrementPressed.count, equals(1));
  });
```

Binder-Tests

Die Binder-Klassen müssen bei relevanten App-Zustands-Änderungen (und möglichst nur dann) dafür sorgen, dass ihr zugehöriges Builder-Widget mit den richtigen Props neu gebaut wird.
Im `testSelectiveRebuild` wird geprüft, dass beim Drücken des Plus-Button das MyCounterWidgetBuilder-Widget neu gebaut wird, aber das MyHomePageBuilder-Widget

nicht.

```
  testWidgets('testSelectiveRebuild',
      (WidgetTester tester) async {

    const app = MyAppBuilder();
    const binder = MyAppStateBinder(child: app);
    await tester.pumpWidget(binder);

    final homePage0 = find.singleWidgetByType(MyHomePageBuilder);
    final counterWidget0 =
        find.singleWidgetByType(MyCounterWidgetBuilder);

    await tester.tap(find.byIcon(Icons.add));
    await tester.pump();

    final homePage1 = find.singleWidgetByType(MyHomePageBuilder);
    final counterWidget1 =
        find.singleWidgetByType(MyCounterWidgetBuilder);

    expect(identical(counterWidget0, counterWidget1), isFalse);
    expect(identical(homePage0, homePage1), isTrue);
  });
```

```
extension SingleWidgetByType on CommonFinders {
  T singleWidgetByType<T>(Type type) =>
      find.byType(type).evaluate().single.widget as T;
}
```

Portierung auf Riverpod

Zum Abschluss des Tutorials soll die App von der selbstgebauten App-Zustands-Verwaltung nacheinander auf die bekannten App-Zustands-Verwaltungs-Frameworks Riverpod und Bloc portiert werden. Wir beginnen mit der Portierung auf Riverpod.

Dazu legen wir im Ordner `lib` einen Unterordner `riverpod` an und darin die Dateien `riverpod.dart` und `riverpod_binder.dart`.

In der Datei `riverpod.dart` definieren wir die Klasse `MyAppStateNotifier` und die finale Variable `appStateProvider` für den App-Zustand.

```dart
class MyAppStateNotifier extends StateNotifier<MyAppState> {
  MyAppStateNotifier()
      : super(const MyAppState(
          title: 'Flutter Demo Home Page',
          counter: 0,
        ));

  late final reduceable = Reduceable(getState, reduce);

  MyAppState getState() => super.state;

  void reduce(Reducer<MyAppState> reducer) => state = reducer(state);
}
```

```dart
final appStateProvider =
    StateNotifierProvider<MyAppStateNotifier, MyAppState>(
  (ref) => MyAppStateNotifier(),
);
```

In der Datei `riverpod.dart` definieren außerdem die finalen Variablen `homePagePropsProvider` und `counterWidgetPropsProvider` zur Bereitstellung der Props.

```dart
final counterWidgetPropsProvider = StateProvider(
  (ref) {
    final appStateNotifier = ref.watch(appStateProvider.notifier);
    return ref.watch(
      appStateProvider.select(
        (state) => MyCounterWidgetProps.reduceable(
          appStateNotifier.reduceable,
        ),
      ),
    );
  },
);
```

```
final homePagePropsProvider = StateProvider(
  (ref) {
    final appStateNotifier = ref.watch(appStateProvider.notifier);
    return ref.watch(
      appStateProvider.select(
        (state) => MyHomePageProps.reduceable(
          appStateNotifier.reduceable,
        ),
      ),
    );
  },
);
```

In der Datei `riverpod_binder.dart` duplizieren wir alle Klassen aus `stateful_binder.dart` und stellen sie auf Riverpod um.

```
class MyAppStateBinder extends StatelessWidget {
  const MyAppStateBinder({super.key, required this.child});

  final MyAppBuilder child;

  @override
  Widget build(context) => ProviderScope(child: child);
}
```

```
class MyHomePageBinder extends ConsumerWidget {
  const MyHomePageBinder({super.key});

  @override
  Widget build(context, ref) => MyHomePageBuilder(
        props: ref.watch(homePagePropsProvider),
      );
}
```

```
class MyCounterWidgetBinder extends ConsumerWidget {
  const MyCounterWidgetBinder({super.key});

  @override
  Widget build(context, ref) => MyCounterWidgetBuilder(
        props: ref.watch(counterWidgetPropsProvider),
      );
}
```

Nun können wir in der Klasse `binder.dart` den Schalter umlegen und anstelle von

`stateful_binder.dart` die Datei `riverpod_binder.dart` exportieren. Damit ist die Portierung auf Riverpod abgeschlossen und die App kann verwendet werden.

Portierung auf Bloc

Den Abschluss des Tutorials bildet die Portierung auf Bloc. Dazu legen wir im Ordner `lib` einen Unterordner `bloc` an und darin die Dateien `bloc.dart` und `bloc_binder.dart`.

In der Datei `bloc.dart` definieren wir die Klasse `MyAppStateBloc` und eine Extension für den bequemen Zugriff auf die Instanz dieser Klasse.

```
class MyAppStateBloc extends Bloc<Reducer<MyAppState>, MyAppState> {
  MyAppStateBloc()
      : super(
          const MyAppState(
            title: 'Flutter Demo Home Page',
            counter: 0,
          ),
        ) {
    on<Reducer<MyAppState>>((event, emit) => emit(event(state)));
  }

  MyAppState getState() => state;

  late final reduceable = Reduceable(getState, add);

}
```

```
extension MyAppStateBlocOnBuildContext on BuildContext {
  MyAppStateBloc get appStateBloc =>
      BlocProvider.of<MyAppStateBloc>(this);
}
```

In der Datei `bloc_binder.dart` duplizieren wir alle Klassen aus `stateful_binder.dart` (oder `riverpod_binder.dart`) und stellen sie auf Bloc um.

```
class MyAppStateBinder extends StatelessWidget {
  const MyAppStateBinder({super.key, required this.child});

  final MyAppState state = const MyAppState(
    title: 'Flutter Demo Home Page',
    counter: 0,
  );
  final Widget child;

  @override
  Widget build(context) => BlocProvider(
        create: (_) => MyAppStateBloc(),
        child: child,
      );
}
```

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

  @override
  Widget build(context) =>
      BlocSelector<MyAppStateBloc, MyAppState, MyHomePageProps>(
        selector: (state) => MyHomePageProps.reduceable(
          context.appStateBloc.reduceable,
        ),
        builder: (context, props) => MyHomePageBuilder(
          props: props,
        ),
      );
}
```

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

  @override
  Widget build(context) =>
      BlocSelector<MyAppStateBloc, MyAppState, MyCounterWidgetProps>(
        selector: (state) => MyCounterWidgetProps.reduceable(
          context.appStateBloc.reduceable,
        ),
        builder: (context, props) => MyCounterWidgetBuilder(
          props: props,
        ),
      );
}
```

Nun können wir in der Klasse `binder.dart` den Schalter umlegen und anstelle von `stateful_binder.dart` oder `riverpod_binder.dart` die Datei `bloc_binder.dart` exportieren. Damit ist die Portierung nach Bloc abgeschlossen und die App kann verwendet werden.

Offene Enden

Grauzonen zwischen UI uns App-Logik

Zur Implementierung einiger UI-Aktionen benötigt man einen BuildContext. Ein prominentes Beispiel ist die Navigation zwischen App-Seiten mit `Navigator.of(BuildContext)`. Die Entscheidung, wann zu welcher App-Seite navigiert wird, ist App-Logik. Die App-Logik sollte möglichst ohne Abhängigkeiten von der UI-Ablaufumgebung bleiben, und ein BuildContext repräsentiert quasi diese Ablaufumgebung.
Ein ähnliches Problem sind UI-Ressourcen wie Bilder, Icons, Farben und Fonts, die eine Abhängigkeit zur UI-Ablaufumgebung besitzen und deren Bereitstellung UI-App-Logik erfordern kann. Zwischen UI-Code und App-Logik-Code gibt es also noch Grauzonen, die in klare Abgrenzungen umgewandelt werden sollten.

Lokale App-Zustände

Der Ansatz, den kompletten App-Zustand als unveränderliche Instanz einer einzigen Klasse zu modellieren, wird bei sehr komplexen Datenstrukturen, sehr großen Datenmengen oder sehr häufigen Änderungsaktionen an seine Grenzen kommen [15].

In der Redux-Dokumentation gibt es Hinweise [16], [17], wie man diese Grenzen durch eine gute Strukturierung der App-Zustands-Klasse erweitern kann.

Letztlich kann man versuchen, Performance-kritische Teile aus dem globalen App-Zustand zu extrahieren und mit lokalen Zustands-Verwaltungs-Lösungen umzusetzen.

UI-Code-Strukturierung

In diesem Artikel wurden Code-Struktur und Entwurfsmuster für eine Trennung der Verantwortlichkeiten von UI-Code und App-Logik-Code diskutiert.
Eine separierte App-Logik kann man mit allen verfügbaren Architektur-Ansätzen und Entwurfmustern weiterstrukturiert werden und ist für mich darum kein offenes Ende dieses Artikels über Flutter-App-Code.
Für den separierten Flutter-UI-Code werden allerdings für größere Projekte weitere interne Strukturierungskonzepte benötigt, die die Vorteile aus dem Flutter-Prinzip 'Alles ist ein Widget' nutzen. Hier einige offene Punkte:

- Wie separiere ich das Theming, z.B. Light Mode und Dark Mode?
- Wie separiere ich die Layout-Adaptionen für verschiedene Endgeräte-Gruppen, z.B. Smartphone, Tablet, Desktop ?
- Wie separiere ich den Code für Animationen?

Groß-Projekt-Erprobung

Die in diesem Artikel vorgestellte Code-Struktur ist ein Ergebnis meiner Erfahrungen aus kleinen und mittleren Flutter-Projekten. Eines davon ist das Projekt Cantarei - die Taizé-Lieder-App. Die App ist frei im Apple- [18] und im Google- [19] App-Store verfügbar, so dass jeder Interessierte selbst einen Eindruck gewinnen kann, für welche Projekt-Größen die hier vorgeschlagenen Konzepte bereits praxiserprobt sind. Ob sie sich, so wie sie sind, mit Erfolg auf größere Projekte anwenden lassen, müsste noch geprüft werden.

Fazit

Code-Struktur und Entwurfsmuster sind wichtige Themen der App-Architektur. Mit diesem Artikel wollte ich das Nachdenken und die Diskussion über diese Themen anregen.

Referenzen

1. Entwurfmuster
   en.wikipedia.org/wiki/Software*design*pattern ↩
2. Trennung der Verantwortlichkeiten
   en.wikipedia.org/wiki/Separation*of*concerns ↩
3. Flutter
   flutter.dev ↩
4. Riverpod
   riverpod.dev ↩
5. Bloc
   bloclibrary.dev ↩
6. Flutter State Management Approaches
   docs.flutter.dev/development/data-and-backend/state-mgmt/options ↩
7. Humble Object Pattern
   xunitpatterns.com/Humble Object.html ↩

8. Reducer Pattern
   redux.js.org/tutorials/fundamentals/part-3-state-actions-reducers ↩

9. Alles ist ein Widget
   docs.flutter.dev/development/ui/layout ↩

10. Michael Feathers michaelfeathers.silvrback.com/bio ↩

11. Faltungsfunktion
    www.cs.nott.ac.uk/~gmh/fold.pdf ↩

12. Redux
    redux.js.org/ ↩

13. React
    reactjs.org/ ↩

14. Wertsemantik
    en.wikipedia.org/wiki/Value_semantics ↩

15. Reducer Pattern Nachteil
    twitter.com/acdlite/status/1025408731805184000 ↩

16. App-Zustands-Gliederung
    redux.js.org/usage/structuring-reducers/basic-reducer-structure ↩

17. App-Zustands-Normalisierung
    redux.js.org/usage/structuring-reducers/normalizing-state-shape ↩

18. Die App *Cantarei* im Apple-Appstore apps.apple.com/us/app/cantarei/id1624281880 ↩

19. Die App *Cantarei* im Google-Playstore play.google.com/store/apps/details?id=de.partmaster.cantarei ↩