Projektarbeit:

Design Patterns in Cross Platform Apps für B2B App

von

Furkan Karayel 36372

Studiengang: Angewandte Informatik

RWU Hochschule Ravensburg-Weingarten

Inhaltsverzeichnis

1 Einführung **1**

2 Grundlagen **2**
2.1 Flutter 2
2.2 State-Management: Riverpod 2
2.3 Model-Generierung: Freezed 2

3 Projekt: MVC+S und Riverpod **4**
3.1 Wie funktioniert der Techstack? 4
3.1.1 ProviderScope 5
3.1.2 StateNotifierProvider 5
3.1.3 Anwendung von StateNotifierProvider 5
3.1.4 Provider auslesen 5
3.2 Riverpod vs. Provider 6
3.2.1 Compile Safe 7
3.2.2 Unabhängig von Flutter 7
3.2.3 Unabhängig von Widget 7
3.2.4 Memory Leaks vereinfacht verhindern 8
3.2.5 FutureProvider 9
3.2.6 StreamProvider 10

4 Riverpod: Bedeutung für das Projekt **11**
4.1 Wie wird ein State Change durchgeführt? 12
4.1.1 Controller 12
4.1.2 View-Provider Überwachung für die Aktualisierung 13
4.2 Was muss beachtet werden wenn eine Page programmiert wird? 13

5 Fazit **17**

6 Literaturverzeichnis **18**

Abbildungsverzeichnis

3.1 MVC+S Funktionsweise Flutter 4

4.1 Providers Definition für das Projekt 11
4.2 SearchView Widget 13
4.3 WeatherHomeModel 14
4.4 Beispiel: isLoading im WeatherHome Widget 15
4.5 onSelected in der SearchView 16

1. Einführung

Die Entwicklung moderner, interaktiver Anwendungen erfordert einen robusten Ansatz für das Zustandsmanagement, um eine effiziente Datenmanipulation und eine klare Trennung der logischen Schichten zu gewährleisten. Flutter, ein von Google entwickeltes Framework, hat sich in der mobilen und Web-Entwicklung durch seine Flexibilität und Leistungsfähigkeit etabliert. Im Flutter-Ökosystem gibt es verschiedene Lösungen für das Zustandsmanagement, darunter Riverpod und Provider. Diese Technologien ermöglichen es Entwicklern, Benutzeroberflächen dynamisch und reaktionsschnell zu gestalten und gleichzeitig Wert auf Wartbarkeit und Erweiterbarkeit zu legen.

Das Ziel dieses Projekts ist es, Riverpod und Provider, zwei führende Ansätze für das Zustandsmanagement in Flutter, detailliert gegenüberzustellen. Es wird untersucht, wie diese Technologien die Herausforderungen der Datenmanipulation zwischen den Benutzeroberflächen adressieren und welche Vorteile sie in Bezug auf die Architektur und die Entwicklungserfahrung bieten. Dazu wird eine Wetteranzeige-Applikation als praktisches Beispiel entwickelt. Der Fokus liegt auf der Untersuchung der Fähigkeiten, eine solide Grundlage für die Trennung der logischen Schichten zu schaffen. Dadurch können Entwickler wartbare und erweiterbare Anwendungen effizient implementieren.

Diese Untersuchung hat zum Ziel, Erkenntnisse darüber zu sammeln und zu vermitteln, wie das Zustandsmanagement optimiert werden kann, um eine nahtlose und intuitive Interaktion mit der Benutzeroberfläche zu ermöglichen. Dabei sollen die Stärken und Schwächen jedes Ansatzes analysiert werden, um ein tieferes Verständnis für die Auswahl des geeigneten Zustandsmanagement-Tools für verschiedene Projektanforderungen zu entwickeln. Dieser Text beleuchtet nicht nur die technischen Aspekte, sondern zeigt auch auf, wie diese Werkzeuge die Entwicklung von nachhaltigen und zukunftssicheren Flutter-Anwendungen unterstützen können.

2. Grundlagen

In diesem Kapitel werden Begriffe und im Rahmen des Projekts genutzte Technologien erläutert, um das Verständnis des Einsatzes besser zu vermitteln.

2.1 Flutter

Flutter hat sich als führendes Framework für die plattformübergreifende Entwicklung etabliert. Mit dem Framework können Entwickler aus einer einzigen Codebasis einheitliche native Anwendungen für Mobile, Web und Desktop erstellen. Flutter vereinfacht den Entwicklungsprozess durch seine reaktive Architektur und vorgefertigten Widgets. Es ermöglicht eine schnelle Anpassung an unterschiedliche Bildschirmgrößen und Betriebssysteme.

Besonders hervorzuheben ist das Hot Reload-Feature, das Codeänderungen in Echtzeit anzeigt, ohne dass die App neu gestartet werden muss. Dies beschleunigt nicht nur den Entwicklungsprozess, sondern fördert auch ein experimentelles Umfeld für UI-Design und Funktionalität.

Flutter nutzt Dart als Programmiersprache, die für ihre Effizienz auf diversen Plattformen bekannt ist. Dart ermöglicht es Flutter-Anwendungen, mit der Geschwindigkeit und Reaktionsfähigkeit nativen Codes zu laufen, was eine reibungslose Benutzererfahrung garantiert. [9]

2.2 State-Management: Riverpod

Riverpod ist ein vielseitiges Framework für reaktives Caching und Datenbindung. Es handelt sich um eine Erweiterung des Provider Package, dem Standard-Statemanagement von Flutter. Das Framework verfügt über ein einziges Widget, das den Zustand aller Provider in Ihrer Flutter-App speichert. Dadurch wird die Abhängigkeit vom Buildcontext von Flutter bewusst vermieden. Riverpod fördert die ordnungsgemäße Trennung von Belangen, indem es die Benutzeroberfläche von der Logik trennt und das Testen verbessert.[1]

2.3 Model-Generierung: Freezed

In Flutter-Anwendungen ist die Verwaltung von Zuständen ein wichtiger Aspekt der Entwicklung. Immutable States, die nach der Erstellung nicht mehr verändert werden können, ermöglichen eine effizientere Verwaltung der Applikationszustände und reduzieren die Wahrscheinlichkeit von Fehlern durch unerwartete Änderungen.

Freezed ist eine einfache Lösung, um unveränderliche Datenklassen durch Codegenerierung zu erzeugen. Die Verwendung von Freezed erspart den Entwicklern einen großen Teil der Codierungsarbeit und gibt ihnen mehr Zeit, sich mit der Geschäftslogik zu beschäftigen.

3. Projekt: MVC+S und Riverpod

3.1 Wie funktioniert der Techstack?

Bei der Umsetzung dieses Projekts wurde die MVC+S Architektur verwendet. Im klassischen MVC-Pattern wird die Geschäftslogik im Model implementiert.

Die Models sind über das Paket "freezed" generiert worden, wodurch sie unveränderliche Datenklassen sind. Daher musste im Projekt die Geschäftslogik innerhalb des Controllers definiert werden. Diese Spaltung von Logik bringt eine klare Separierung von Logik in die Projektstruktur und erleichtert somit die Wartbarkeit sowie das Testing.

Der Controller enthält die gesamte Geschäftslogik und verarbeitet Nutzeraktionen oder Ereignisse aus der View-Schicht. Die Services werden per Dependency Injection dem Controller eingespeist, wodurch die Kommunikation in die Außenwelt ermöglicht wird.

Mit der Nutzung des StateNotifierProvider des Riverpod Pakets wird das Statemanagement ermöglicht. Dabei wird eine Reaktivität ermöglicht wodurch in der View die Model States belauscht/observiert werden um somit die Benutzeroberfläche enstprechend zu aktualisieren. Der State eines StateNotifierProvider wird überwacht, wobei der State das Model und der Notifier der Controller ist.

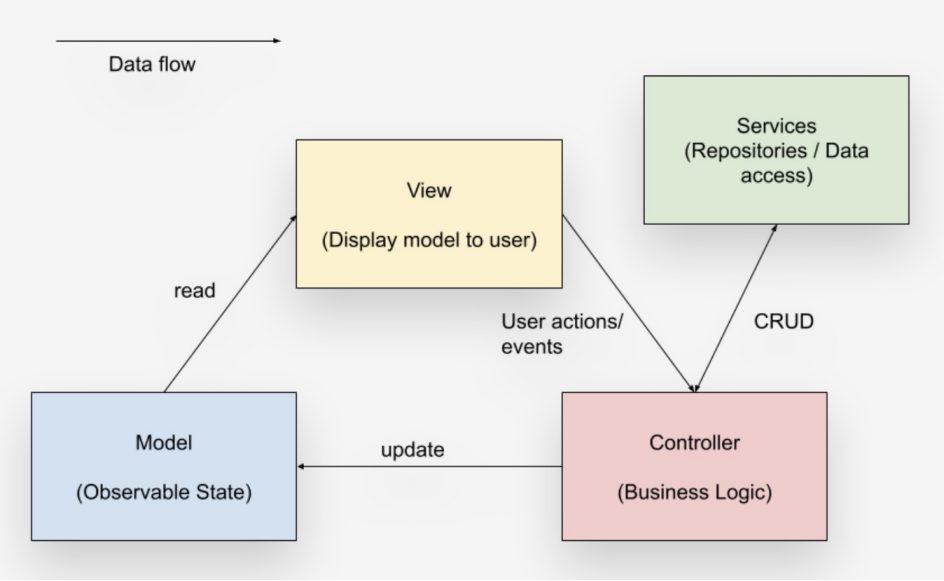

Figure 3.1: MVC+S Funktionsweise Flutter

3.1.1 ProviderScope

Dieses Widget dient zum Zweck, alle Zustände der Provider innerhalb des Projekts zu speichern und ist für die Nutzung Riverpod Pakets relevant. An der Wurzel des Widget-Baums des Projekts muss dieses Widget enthalten sein. Dies sieht aus wie folgt:

```
1 void main() {
2   runApp(ProviderScope(child: MyApp()));
3 }
4
```

3.1.2 StateNotifierProvider

StateNotifierProvider ist ein Provider, der einen StateNotifier zur Verfügung stellt und zum Lauschen verwendet wird. Riverpod empfiehlt die Verwendung von StateNotifierProvider zusammen mit StateNotifier zur Verwaltung von Zuständen, die sich in Reaktion auf eine Benutzerinteraktion ändern können.
Im Projekt wird ein StateNotifierProvider pro Widget (Anzeige) unter der Datei *providers.dart* festgelegt, wobei der State das spezifische Model ist und der Controller der Notifier.

3.1.3 Anwendung von StateNotifierProvider

Der StateNotifierProvider wird in der View Schicht angewendet wodurch ermöglicht wird, Controller Funktionen zur Verarbeitung von Nutzeraktionen zu verarbeiten und beim State Change sprich das Model eine Änderung bekommt die Anzeige neu gebaut wird.

Umstellung: ConsumerWidget statt StatelessWidget

Durch die Verwendung von ConsumerWidget, welcher Stateless ist, kann der Widget-Baum auf Änderungen des Providers lauschen, so dass die Benutzeroberfläche bei Bedarf automatisch aktualisiert wird.
Dabei entsteht in der build-Methode des Widgets eine Änderung; als Übergabeparameter kommt ref des Typs WidgetRef dazu.
Durch diese Änderung, wird es möglich auf beliebige Provider innerhalb der build-Methode zu lauschen. [4]

3.1.4 Provider auslesen

Riverpod bietet eine flexible und vielseitige Herangehensweise an das Zustandsmanagement, die es Entwicklern ermöglicht, Zustände auf verschiedene Arten auszulesen und darauf zu reagieren. Hier sind die drei Hauptmethoden, um die Werte eines Providers über Riverpod auszulesen:

- **ref.watch** Diese Methode wird verwendet, um auf Änderungen eines Zustands zu reagieren und die Benutzeroberfläche entsprechend zu aktualisieren. Wenn sich der Wert des beobachteten Zustands ändert, sorgt ref.watch dafür, dass das Widget, in dem es aufgerufen wird, neu gebaut wird, um die neuesten Daten

anzuzeigen. Dies ist besonders nützlich, um dynamische Daten in der UI darzustellen, die sich im Laufe der Zeit ändern können. [7]

Für die Wetterapplikation wird dieser Ansatz genutzt, um den State (Model) auszulesen und dessen Änderung zu erkennen.

Beispiel:

```
1         final WeatherHomeModel model = ref.watch(providers.weatherControllerProvider);
```

- **ref.listen** Ähnlich wie ref.watch, aber mit dem Unterschied, dass anstatt eines Neubaus der Benutzeroberfläche eine spezifische Aktion oder Funktion ausgeführt wird, wenn sich der beobachtete Zustand ändert. ref.listen ist leistungseffizient für Fälle, in denen eine Zustandsänderung nicht unbedingt eine Aktualisierung der UI erfordert, sondern beispielsweise eine Benachrichtigung oder einen Log-Eintrag auslösen soll.[7]

  In der Wetterapplikation hat dieser Ansatz keine Verwendung stattgefunden.

- **ref.read** Diese Methode wird verwendet, um den aktuellen Wert eines Zustands einmalig auszulesen, ohne dabei auf zukünftige Änderungen zu reagieren. ref.read eignet sich für Fälle, in denen man sicher ist, dass der Wert des Zustands sich nicht ändern wird, während das Widget angezeigt wird, oder wenn man den Wert außerhalb des Build-Prozesses benötigt, etwa in einem Event-Handler. [7]

  In der Wetterapplikation wird über Read der Controller eingelesen.

  Beispiel:

```
1     void onSelected(String location) {
2       final weatherController = ref.read(providers.weatherControllerProvider.notifier);
3       weatherController.fetchWeatherData(location);
4       Navigator.of(context).pop();
5     }
```

3.2 Riverpod vs. Provider

In der Welt der Flutter-Entwicklung spielt die Verwaltung von Anwendungszuständen eine zentrale Rolle bei der Erstellung reaktionsfähiger und benutzerfreundlicher Anwendungen. Obwohl das Flutter-Framework keine eingebaute Lösung für das Zustandsmanagement bietet, hat die Community im Laufe der Zeit mehrere Ansätze entwickelt, um diese Herausforderung zu meistern.
Das Provider Package hat sich aufgrund seiner Einfachheit und Effizienz als Standard durchgesetzt. Mit dem Erscheinen von Riverpod, einem neueren Paket, das vom selben Autor wie Provider entwickelt wurde, haben Entwickler jedoch die Wahl zwischen zwei leistungsstarken Zustandsverwaltungswerkzeugen. Beide Pakete versprechen, das Zustandsmanagement in Flutter-Anwendungen zu vereinfachen, unterscheiden sich jedoch in Design, Funktionalität und Einsatzmöglichkeiten. In diesem Abschnitt werden wir einen detaillierten Vergleich zwischen Riverpod und Provider anstellen, um ihre Unterschiede, Vorteile und potentiellen Anwendungsfälle zu beleuchten. Ziel ist es,

Entwicklern eine klare Orientierung zu geben, welche Lösung für ihre spezifischen Projektanforderungen am besten geeignet ist.

3.2.1 Compile Safe

Riverpod verbessert die Zustandsverwaltung in Flutter-Anwendungen durch das Konzept der Compile Safe"- oder Compile-Time"-Sicherheit. Dies bedeutet, dass Riverpod hilft, viele Fehler zu erkennen und zu verhindern, bevor die Anwendung überhaupt ausgeführt wird.

Im Vergleich dazu basiert das Standard State Management über den Provider-Ansatz mehr auf Laufzeitprüfungen, was bedeutet, dass Fehler oder Probleme in der Anwendung oft erst entdeckt werden, wenn sie bereits ausgeführt wird. [2]

3.2.2 Unabhängig von Flutter

Der Provider-Ansatz ist eng mit Flutter verwoben, da er Widgets verwendet, um Zustände zu verwalten und abhängige Objekte im Widget-Baum bereitzustellen. Dies führt dazu, dass die Verwendung von Provider stark an die Präsenz eines Flutter-UI-Frameworks gebunden ist. Zustände und Dienste werden durch Provider-Widgets bereitgestellt, die in den Flutter-Widget-Baum integriert werden müssen.

Riverpod hingegen ist so konzipiert, dass es unabhängig von Flutter funktioniert. Es nutzt reine Dart-Objekte für das Zustandsmanagement und die Abhängigkeitsinjektion. Diese Unabhängigkeit von Flutter macht Riverpod zu einem vielseitigeren Tool, das auch außerhalb des Kontexts von UI-Entwicklungen verwendet werden kann. Beispielsweise ist es möglich, Riverpod in Dart-basierten Serveranwendungen, CLI-Programmen oder anderen Dart-Projekten einzusetzen, die kein UI benötigen. [6]

Riverpod Vorteile

- **Wiederverwendbarkeit**: Logik und Zustandsmanagement, die mit Riverpod entwickelt wurden, können leicht zwischen Flutter-Anwendungen und anderen Dart-Projekten geteilt werden.
- **Flexibilität**: Entwickler können die leistungsstarken Funktionen von Riverpod in einer Vielzahl von Projekten nutzen, nicht nur in denen, die eine Benutzeroberfläche haben.
- **Testbarkeit**: Da Riverpod nicht von Flutter abhängt, ist es einfacher, das Zustandsmanagement und die Logik unabhängig vom UI-Code zu testen, was zu robusterem Code führt.

3.2.3 Unabhängig von Widget

Der traditionelle Provider-Ansatz in Flutter verwendet den Kontext, um auf Zustände (States) zuzugreifen. Das bedeutet, dass Widgets auf Zustände zugreifen, indem sie sich auf ihre Position im Widget-Baum beziehen. Zustände sind also an den Kontext

gebunden und erfordern, dass ein Widget einen direkten oder indirekten Vorfahren im Baum hat, der den gewünschten Zustand bereitstellt. Diese Struktur macht den Zugriff auf Zustände ortsabhängig und kann zu Komplikationen führen, wenn Zustände in weit verzweigten oder tief verschachtelten Widget-Bäumen benötigt werden.

Riverpod löst dieses Problem, indem es den Zugriff auf Zustände von der Widget-Struktur entkoppelt. Provider werden als globale Variablen deklariert, auf die von überall in der Anwendung zugegriffen werden kann, ohne auf den context angewiesen zu sein. [6] [7]

Riverpod Vorteile

- **Ortsunabhängiger Zugriff**: Da Provider global verfügbar sind, können Zustände leicht von jedem Punkt in der App aus abgerufen oder geändert werden, unabhängig von der Struktur des Widget-Baums.
- **Vereinfachte Zustandsverwaltung**: Die Notwendigkeit, den context zu durchlaufen, um auf Zustände zuzugreifen, entfällt, was den Code vereinfacht und die Fehleranfälligkeit reduziert.
- **Verbesserte Komposition**: Riverpod ermöglicht es, auf einfache Weise Zustände in anderen Zuständen zu verwenden oder Zustände miteinander zu kombinieren, da der Zugriff auf Provider nicht von der Widget-Hierarchie abhängt.
- **Verbesserte Testbarkeit**: Da Zustände nicht mehr direkt an Widget-Bäume gebunden sind, ist es einfacher, sie in Tests zu isolieren und zu mocken.

3.2.4 Memory Leaks vereinfacht verhindern

Bei der Verwendung des Provider-Pakets in Flutter werden Zustände oft im Speicher gehalten, selbst wenn sie nicht mehr benötigt werden. Provider ist primär auf die Bereitstellung von Zuständen und Abhängigkeiten innerhalb des Widget-Baums ausgerichtet, ohne automatische Mechanismen zur Erkennung und Freigabe nicht mehr benötigter Zustände.Entwickler müssen ungenutzte Zustände und Ressourcen ordnungsgemäß verwalten und freigeben, um Memory Leaks zu vermeiden.

Riverpod führt das leistungsstarke Feature .autoDispose ein, um mit diesem Problem umzugehen. Wenn ein Provider mit .autoDispose markiert ist, wird sein Zustand automatisch zerstört, sobald er nicht mehr beobachtet wird. Das bedeutet, dass Riverpod den Speicher, der von diesem Zustand belegt wird, automatisch freigibt, sobald kein Widget oder anderer Zustand mehr auf den mit .autoDispose markierten Provider angewiesen ist.[5]

Riverpod Vorteile

- **Automatisierte Speicherverwaltung**: Entwickler müssen sich weniger Sorgen um die manuelle Bereinigung von Zuständen machen, da Riverpod nicht mehr benötigte Zustände automatisch erkennt und entfernt.

- **Verbesserte Performance**: Durch die Bereinigung der ungenutzten Daten kann die Gesamtperformance der Anwendung verbessert werden, insbesondere bei lang laufenden oder ressourcenintensiven Apps.
- **Einfachere Codebasis**: Da die Notwendigkeit der manuellen Verwaltung von Lebenszyklen und der Freigabe von Ressourcen verringert wird, kann der Code einfacher und sauberer gestaltet werden.

3.2.5 FutureProvider

Riverpod bringt mit dem FutureProvider eine leistungsfähige Lösung für das Management asynchroner Datenoperationen, wie das Laden von Daten über Netzwerkanfragen. Dieser Ansatz bietet eine elegante Handhabung von Ladezuständen, Fehlerbehandlungen und Caching, was in traditionellen Provider-Ansätzen nicht so ohne Weiteres möglich ist. [3] Hier ist, wie der FutureProvider in Riverpod funktioniert und welche Vorteile er bietet:

Ladezustand

Mit dem FutureProvider kann der Ladezustand von asynchronen Operationen einfach gehandhabt werden. Riverpod stellt einen Mechanismus bereit, um den aktuellen Zustand des Futures zu überwachen - ob er noch lädt, erfolgreich abgeschlossen wurde oder einen Fehler erlitten hat. Dies ermöglicht es Entwicklern, Benutzeroberflächen zu erstellen, die auf diese Zustände reagieren und beispielsweise Ladeanimationen anzeigen, während auf die Antwort einer Netzwerkanfrage gewartet wird.

Fehlerzustand

Wenn während der Ausführung des Futures ein Fehler auftritt, kann dieser Zustand ebenfalls über den FutureProvider abgefangen und behandelt werden. Das ermöglicht es, Fehlermeldungen oder -ansichten benutzerfreundlich anzuzeigen, ohne dass zusätzlicher Code zur Fehlerüberprüfung erforderlich ist.

Caching

Ein weiterer signifikanter Vorteil des FutureProvider ist das eingebaute Caching. Sobald ein Future erfolgreich abgeschlossen wurde, speichert der FutureProvider das Ergebnis. Wenn das Widget, das den FutureProvider nutzt, neu gebaut wird, wird der gecachte Wert sofort zurückgegeben, anstatt den Future erneut auszuführen. Dies verbessert die Leistung und die Benutzererfahrung, da Daten nicht jedes Mal neu geladen werden müssen, wenn ein Benutzer zu einem bereits besuchten Bildschirm zurückkehrt. Die Verwendung von .autoDispose mit dem FutureProvider löscht den Cache, wenn der Provider nicht mehr beobachtet wird, was hilfreich ist, um den Speicherverbrauch zu optimieren und sicherzustellen, dass veraltete Daten nicht gehalten werden.

Beispiel:

```
final myFutureProvider = FutureProvider.autoDispose<String>((ref) async {
    // Simuliere das Laden von Daten
    await Future.delayed(Duration(seconds: 2));
    return "Daten geladen";
});

// In einem Widget kann man nun auf den Zustand des Futures reagieren:
Consumer(builder: (context, ref, child) {
    final asyncValue = ref.watch(myFutureProvider);

    return asyncValue.when(
        data: (data) => Text(data),
        loading: () => CircularProgressIndicator(),
        error: (e, stack) => Text('Fehler: $e'),
    );
});

```

3.2.6 StreamProvider

Der StreamProvider in Riverpod ist eine spezielle Form des Providers, die für den Umgang mit Streams konzipiert wurde. Ähnlich dem FutureProvider für asynchrone Einzeloperationen ermöglicht der StreamProvider die Integration von kontinuierlichen Datenströmen in Flutter-Anwendungen. Er ist ideal für Szenarien, in denen Daten in Echtzeit empfangen oder überwacht werden müssen, wie z.B. bei Firebase-Updates, Websocket-Verbindungen oder anderen stetigen Datenquellen. [8]

4. Riverpod: Bedeutung für das Projekt

Es gibt verschiedene Möglichkeiten, Provider innerhalb des Projekts zu definieren. Während der Umsetzung dieses Projekts wurde der Ansatz der zentralen Verwaltung der Provider gewählt, wie in der folgenden Abbildung zu sehen ist. Die Verwendung dieser Methode bietet den Vorteil der Übersichtlichkeit, der Möglichkeit der Dependency Injection bei der Erzeugung des Controllers und einer einfachen Möglichkeit, den Status der Applikation über die Controller-Methoden zu manipulieren.

```dart
import 'package:flutter_weatherapp/pages/weather/model/location_search_model.dart';
import 'package:flutter_weatherapp/services/api_service.dart';
import 'package:flutter_weatherapp/pages/weather/controller/search_controller.dart';
import 'package:flutter_weatherapp/pages/weather/controller/weather_controller.dart';
import 'package:flutter_weatherapp/pages/weather/model/weather_model.dart';
import 'package:riverpod/riverpod.dart';

final Providers providers = Providers();

class Providers {
  final StateNotifierProvider<WeatherController, WeatherHomeModel>
      weatherControllerProvider =
      StateNotifierProvider<WeatherController, WeatherHomeModel>(
          (StateNotifierProviderRef ref) =>
              WeatherControllerImpl(backendService: ApiClientImpl())); // StateNotifierProvider

  final StateNotifierProvider<LocationSearchController, LocationSearchHomeModel>
      searchControllerProvider =
      StateNotifierProvider<LocationSearchController, LocationSearchHomeModel>(
          (StateNotifierProviderRef ref) =>
              LocationSearchControllerImpl(backendService: ApiClientImpl())); // StateNotifierProvider
}
```

Figure 4.1: Providers Definition für das Projekt

Das Riverpod Setup im Projekt bringt folgende Vorteile mit sich:

- **Trennung von Logik:** Durch die Verwendung von Anbietern ist es möglich die Logik der Anwendung in verschiedene Klassen oder Controller aufteilen. Dies fördert eine modularere und besser wartbare Codebasis, da jede Klasse für einen bestimmten Aspekt der Funktionalität der Anwendung verantwortlich ist.

- **Dependency Injection:** Provider erleichtern die Injektion von Abhängigkeiten, indem sie es ermöglichen, Abhängigkeiten (z. B. API-Dienste) in die jeweiligen Controller oder andere Klassen zu injizieren. Dies fördert die lose Kopplung zwischen den Komponenten und macht den Code besser testbar und einfacher zu refaktorisieren.

- **Reaktive Zustandsverwaltung:** Riverpod bietet reaktive Zustandsverwaltungsfunktionen, mit denen Ihre Benutzeroberfläche automatisch auf Zustandsänderungen

reagieren kann. Dies ermöglicht einen deklarativen und effizienten Programmierstil, da Sie definieren können, wie sich Ihre Benutzeroberfläche basierend auf dem aktuellen Zustand Ihrer Anwendung verhalten soll.

- **Immutable State:** Riverpod fördert die Verwendung unveränderlicher Zustandsobjekte, was dazu beiträgt, häufige Probleme im Zusammenhang mit veränderlichen Zuständen zu vermeiden, wie z. B. unerwartete Seiteneffekte und Race Conditions.

4.1 Wie wird ein State Change durchgeführt?

Der State Change einer Page ist ausschließlich möglich über die Controller. In der Abbildung **??** ist zu sehen, wie der Controller definiert ist.

4.1.1 Controller

Hier die Definition noch einmal:

```
1  abstract class LocationSearchController
2      extends StateNotifier<LocationSearchHomeModel> {
3    LocationSearchController(LocationSearchHomeModel state) : super(state);
4    void fetchLocations(String location);
5  }
```

Im Grunde ist der Controller eine Erweiterung von einem StateNotifier eines bestimmten Typs, welches das Model ist. Der initiale State der Superklasse wird durch die Zeile 3 gesetzt und anschließend die Funktion der abstrakten Klasse definiert.

```
1  class LocationSearchControllerImpl extends LocationSearchController {
2    final BackendService _backendService;
3
4    LocationSearchControllerImpl({
5      required BackendService backendService,
6      LocationSearchModel? model,
7    })  : _backendService = backendService,
8          super(LocationSearchHomeModel(
9              currentDataTable: List.empty(), isLoading: false, hasError: false));
10
11   @override
12   Future<void> fetchLocations(String location) async {
13     final response = await _backendService.fetchGeoLocation(location);
14     List<LocationSearchModel> list = [];
15
16     for (var i = 0; i < response.length; i++) {
17       var tempModel = LocationSearchModel(
18           cityName: response['name'],
19           lat: response['lat'].toString(),
20           lon: response['lon'].toString());
21       list.add(tempModel);
22     }
23
24     state = state.copyWith(currentDataTable: list);
```

```
25    }
26  }
27
```

Die Implementierung (LocationSearchControllerImpl) der abstrakten Klasse ist im oberen Codeabschnitt. Da in den Providers ein Backendservice per Dependency Injection (DI) injiziert wird, ist hier nun die Stelle der Initialisierung des Controllers mit der Service Schnittstelle. In der fetchLocations Methode des Controllers werden die Städte mit den Breiten- und Längengraden per Backendservice Methode abgerufen, die in eine Liste vom Typ **LocationSearchModel** hinzugefügt werden. Diese Liste wird im Anschluss der Befüllung an das View übergeben. Der aktuelle State wird überschrieben, was nach der richtigen Konfiguration auf der View-Seite zu einem Rebuild der Benutzeroberfläche führt.

4.1.2 View-Provider Überwachung für die Aktualisierung

Damit die Reaktivität ins Spiel kommt, muss bei der in der Wetter Applikation ausgewählten Riverpod Architektur ein **ref.watch()** auf den ControllerProvider innerhalb der build() Methode des Widgets definiert werden, wie in der folgenden Abbildung zu sehen ist.

```
9   class SearchView extends ConsumerWidget {
10    SearchView({Key? key, required this.current}) : super(key: key);
11    TextEditingController textController = TextEditingController();
12    final WeatherModel current;
13
14    @override
15    Widget build(BuildContext context, WidgetRef ref) {
16      List<LocationSearchModel> selectedData = [];
17      final LocationSearchController controller =
18          ref.read(providers.searchControllerProvider.notifier);
19      final LocationSearchHomeModel model =
20          ref.watch(providers.searchControllerProvider);
```

Figure 4.2: SearchView Widget

Normalerweise gibt es in Flutter **StatefulWidget** und **StatelessWidget**, wenn der Zustand der Views ohne weitere Bibliotheken gehandhabt wird. Durch den Einsatz von Riverpod bei der **SearchView** wird ein **ConsumerWidget**, welcher auch zustandslos ist, so geändert um über Riverpod auch Zustände nutzen zu können welche bleibend sind. Der ConsumerWidget hat speziell die Aufgabe die UI bei Provideränderungen gezielt aktualisieren können.

4.2 Was muss beachtet werden wenn eine Page programmiert wird?

Das Architektur Pattern MVC+S in Flutter erfordert primär die Beantwortung von einigen Fragen aus der View und Model Perspektive. Für die Beantwortung folgender Fragestellungen welche aus meinem Standpunkt relevant für die Umsetzung sind, nehme ich als Beispiel das WeatherHomeModel:

1. **Wie muss das Model für meine Anzeige aussehen?**

Für die Anzeige der Temperatur wird zunächst einmal standardmäßig immer die Stadt Berlin genommen. Das Weather Home View besteht aus den UI-Kits, **WeatherCurrent** und **WeatherForecastToday**, welche zwecks Wiederverwendbarkeit ausgelagert wurden. Da das WeatherModel gezielte wetterspezifische Attribute enthält, kann dieses einzelne Model nicht den State des gesamten Weather-Home Widgets beinhalten und eignet sich nur als einziges Wetter Objekt, welches die Informationen zu einem Zeitpunkt beinhaltet.

```dart
@freezed
abstract class WeatherHomeModel with _$WeatherHomeModel {
  factory WeatherHomeModel({
    required WeatherModel currentTemperature,
    required List<WeatherModel> forecasts,
    required bool isLoading,
    required bool hasError,
  }) = _WeatherHomeModel;

  factory WeatherHomeModel.fromJson(Map<String, dynamic> json) =>
      _$WeatherHomeModelFromJson(json);
}
```

Figure 4.3: WeatherHomeModel

Daher wurde das sogenannte Wrapper Model namens WeatherHomeModel zur Nutzung für den State erstellt. Das Model beinhaltet ein WeatherModel **currentTemperature**, für die Anzeige der Daten zum jetztigen Zeitpunkt, eine Liste von WeatherModels namens **forecasts** zur Vorhersage von dem ausgewählten Ort und zwei bool Variablen, welche für den UI-Anzeige-Fortschritt relevant sind; **isLoading** und **hasError**.

Beispiel: isLoading wird in der Controller Funktion bei einem Fetch der Daten über den Backendservice auf true gesetzt, so weiß der User, dass in dem Moment Daten geladen werden. Falls eine Exception während dieses Prozesses eintrifft, kann darauf in der View auch dementsprechend reagiert werden. Für isLoading ist unten ein Ausschnitt von WeatherHome.

```
Padding(
  padding:
      EdgeInsets.all(size.width * 0.005),
  child: SingleChildScrollView(
    scrollDirection: Axis.horizontal,
    child: Row(
      children: [
        model.isLoading
            ? const CircularProgressIndicator()
            : WeatherForecastToday(
                data: model.forecasts[0],
                isDarkMode: isDarkMode,
                size: size), // WeatherForecastToday
        WeatherForecastToday(
            data: model.forecasts[1],
            isDarkMode: isDarkMode,
            size: size), // WeatherForecastToday
        WeatherForecastToday(
            data: model.forecasts[2],
            isDarkMode: isDarkMode,
            size: size), // WeatherForecastToday
        WeatherForecastToday(
            data: model.forecasts[3],
            isDarkMode: isDarkMode,
            size: size), // WeatherForecastToday
        WeatherForecastToday(
            data: model.forecasts[4],
            isDarkMode: isDarkMode,
            size: size) // WeatherForecastToday
      ],
    ), // Row
  ), // SingleChildScrollView
), // Padding
```

Figure 4.4: Beispiel: isLoading im WeatherHome Widget

2. **Welche Abhängigkeiten braucht die Page?**

   Die Frage hier ist, ob die zu programmierende Page externe Daten, und somit ein Backendservice per DI, bekommen muss oder nicht. Gibt es andere zu erfüllende Abhängigkeiten, wie die Verwendung einer Datenbank? Diese kann auch per DI unter den Providers für die jeweiligen Pages verteilt werden.

3. **Wird zwischen den Pages auf gemeinsame Daten zugegriffen?**

   In der Wetterapplikation gibt es die Home- und die Search-Ansicht, beide Pages sind voneinander unabhängig. Durch die Auswahl eines Suchergebnisses auf der Search-Seite, wird beispielsweise ein Backend-Fetch mit dem ausgewählten Stadtnamen ausgeführt woraufhin der User zurück auf die Home Seite gebracht wird. Im Folgenden ist die Methode, welche die erwähnten Schritte durchführt:

```
void onSelected(String location) {
  final weatherController =
      ref.read(providers.weatherControllerProvider.notifier);
  weatherController.fetchWeatherData(location);
  Navigator.of(context).pop();
}
```

Figure 4.5: onSelected in der SearchView

Falls jedoch kein Fetch für die Anzeige der Daten auf einer anderen Seite getriggert werden muss sondern statisch überschrieben, kann man sich dazu Gedanken machen ob es sinnvoll ist dafür eine spezifische Controller-Methode der betroffenen Page zu erstellen oder einen passenden Provider zu definieren, welche nur für den gemeinsamen Zugriff einer Ressource bestimmt ist.

5. Fazit

Das Projekt wurde mit einer Riverpod verbundenen MVC+S Architektur umgesetzt, bei der die Service Schicht für die Kommunikation nach außen steht. Durch die Nutzung dieser Architektur werden Aktionen wie die Änderung oder Erweiterung der bestehenden Teile recht einfach. Es hätten mehr Showcases mit reingenommen werden können wie das Testing, Implementierung einer Datenbank oder Verwendung der Cache Funktionalität, welche z.B.: mit den FutureProviders dazugekommen wäre, um eine verbesserte Performance der App bei externen Daten- oder Bildzugriffen zu haben, jedoch hätte dies den Rahmen der Projektarbeit gesprengt.
Im Laufe des Projekts wurden die Funktionalitäten; Backend Service für Wetter und Geographie Daten, Anzeige für Wetterdaten, Suche nach weltweiten Städten und Anzeige der Wetterdaten für die ausgewählte Stadt implementiert. Die vereinfachte Model-Erstellung über Freezed führt zu einer erheblichen Zeiteinsparung. Es wurden im Projekt auch nicht alle Möglichkeiten die aus der Verwendung von Freezed erstellten Models und deren Methoden demonstriert. Es wurde eher ein klares Beispiel mit einer Wetterapplikation genommen, anhand von der ein sauberes MVC+S Pattern mit zwei View-Pages, Start und Suche, welche auf verschiedene Schnittstellen von OpenWeatherAPI basieren implementiert. Dabei wurden wichtige Fragen und Hinweise für die Nutzung dieser Architektur festgehalten die zusammen mit dieser Dokumentation und dem Repository auf Github hilfreich für die Nutzung dieser technischen Herangehensweise sein sollten.

6. Literaturverzeichnis

[1] Nikki Eke. *Mastering Riverpod in Flutter: The Ultimate Guide for Flutter Beginners*. 2023. URL: https://dev.to/nikki_eke/master-riverpod-even-if-you-are-a-flutter-newbie-2m34 (visited on 03/09/2024).

[2] Flutter.dev. *List of state management approaches*. 2024. URL: https://docs.flutter.dev/data-and-backend/state-mgmt/options (visited on 03/19/2024).

[3] Online. *Riverpod and FutureProvider*. 2024. URL: https://fitech101.aalto.fi/device-agnostic-design/09-futures-and-working-with-apis/5-flutter-riverpod-and-futureprovider/ (visited on 03/19/2024).

[4] Pub.dev. *ConsumerWidget class*. 2024. URL: https://pub.dev/documentation/flutter_riverpod/latest/flutter_riverpod/ConsumerWidget-class.html (visited on 03/19/2024).

[5] Riverpod.dev. *.autoDispose*. 2024. URL: https://riverpod.dev/docs/concepts/modifiers/auto_dispose (visited on 03/18/2024).

[6] Riverpod.dev. *Provider vs Riverpod*. 2024. URL: https://riverpod.dev/de/docs/from_provider/provider_vs_riverpod (visited on 03/18/2024).

[7] Riverpod.dev. *Reading a Provider*. 2024. URL: https://riverpod.dev/docs/concepts/reading (visited on 03/19/2024).

[8] Riverpod.dev. *StreamProvider*. 2024. URL: https://riverpod.dev/de/docs/providers/stream_provider (visited on 03/19/2024).

[9] Wikipedia. *Flutter (software)*. 2024. URL: https://en.wikipedia.org/wiki/Flutter_(software) (visited on 03/09/2024).