Projektarbeit:

Design Patterns in Cross Platform Apps für B2B App

von

Furkan Karayel 36372

Studiengang: Angewandte Informatik

RWU Hochschule Ravensburg-Weingarten

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Dieses Projekt ist im gesamten eine Zusammenstellung bzw. eine Demonstration von vielen möglichen guten Praktiken bei der Umsetzung von Applikationen für Cross Platform Devices. Für das Projekt wurde die konkrete Programmiersprache Dart mit dem Entwicklungs-Kit Flutter verwendet, welche von Google ist. Die Projektidee ist eine Wetteranzeige Applikation, welche auf den Daten der Openweathermap.org basiert, zu programmieren und dabei eine Sammlung von praxisnahen Tipps festzuhalten welche für das Erstellen von weiteren Flutter Applikationen als Guide dienen kann.

Kapitel 2

Projekt Architektur

2.1 Was ist MVC?

Model-View-Controller (MVC) ist ein Architektur-Pattern, das eine Anwendung in drei logische Hauptkomponenten unterteilt: das Model, den View und den Controller. Jede dieser Komponenten ist so konzipiert, dass sie bestimmte Entwicklungsaspekte einer Anwendung handhabt.

2.2 Was ist MVC+S?

Bei der Umsetzung dieses Projekts wurde die MVC+S Architektur verwendet. Der Zweck des Einsatzes dieses Patterns ist Applikationen mit einer besser strukturierten, kontrollierbaren und lesbaren Architektur zu erschaffen. Der grundsätzliche Unterschied zum MVC Pattern, ist die Separierung der Service Ebene, welche die Applikation mit der Außenwelt kommunizieren lässt.

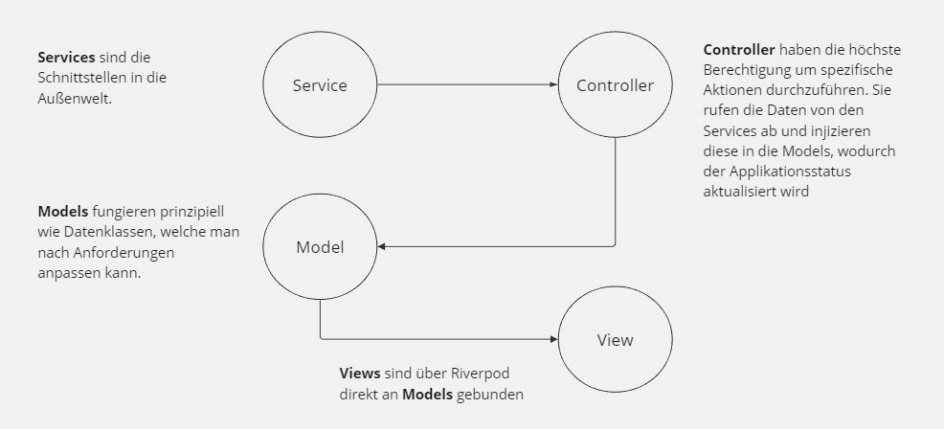


Figure 2.1: MVC+S Funktionsweise Flutter

2.3 Ordner Struktur

Die Ordner Struktur ist sehr individuell und variiert je nach Programmiersprache. Während des Projekts wurde diese Art und Weise der Aufteilung von Dateien in Ordnern bevorzugt, da es übersichtlich ist und eine saubere Aufteilung der genutzten Komponente mit sich bringt. Die gesamten Projektdateien befinden sich im Lib-Ordner. Unter Common können beispielsweise Dateien eingefügt werden, welche eine Sammlung von allgemeineren Definitionen beinhalten. Pages beinhaltet alle Dateien welche zur Anzeige einer UI Seite gehören, bei der spezifischen Page selbst gibt es eine MVC Ordnerstruktur, die jeweilige Schichten logisch trennt. Services haben eine allgemeinere Bedeutung und können global eingesetzt werden um Teile der Applikation mit der Außenwelt kommunizieren zu lassen, daher ist der Ort im Lib Ordner. UI-Kit, sind visuelle wiederverwendbare Komponenten, die nicht spezifisch verwendet werden und in verschiedenen View-Pages genutzt werden können, daher werden diese speziell in den Order separiert.

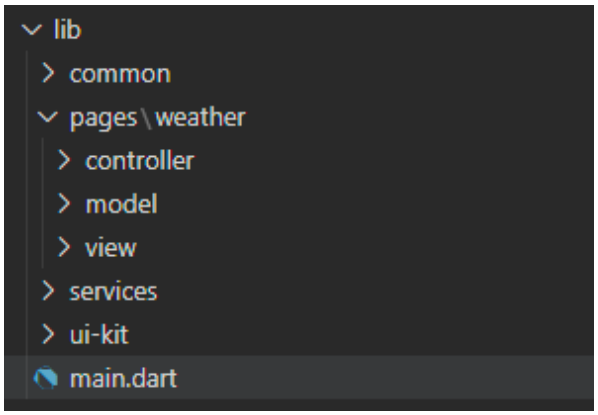


Figure 2.2: Ordner Struktur der Wetter Applikation

2.4 Model

Modelle sind Datenklassen, die Sie nach Ihren Bedürfnissen erstellen. Mit Hilfe des Controllers können Sie die Modelldaten manipulieren.

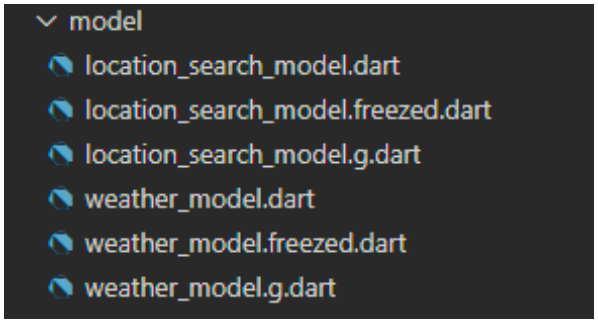


Figure 2.3: Models der Wetter Applikation

Die angezeigten Models gehören zur Weather Page, daher sind diese in dem Ordner

lib/pages/Weather/model zu finden. In der Flutter Welt ist es ein weit verbreiteter Ansatz die Models über @freezed zu generieren.

Der vom Benutzer generierte Inhalt, also die Vorgabe wie das Datenmodell auszuschauen hat, wird in der Hauptdatei mit der Endung Bsp.: `weather_model.dart` definiert.

2.4.1 Freezed

In Abbildung unten ist zu sehen wie eine Beispielsdatei für die Nutzung von Freezed aussieht. Hierfür muss zunächst einmal die Abhängigkeit `Freezed` und `Freezed_annotation` für den Code Generator innerhalb des Projekts installiert werden. Dafür werden im Terminal folgender Befehle ausgeführt:

- **flutter pub add freezed**
- **flutter pub add freezed_annotation**

Wie in der Abbildung zu sehen ist, wurde eine Datenklasse namens **WeatherHomeModel** definiert, die vier Felder enthält: currentTemperature, forecasts, isLoading und hasError. Ein Fabrikkonstruktor wurde definiert, der alle Felder als Parameter übernimmt und sie den entsprechenden Eigenschaften zuweist. Das Schlüsselwort @required wurde verwendet, um anzugeben, dass diese Felder obligatorisch sind und nicht leer sein dürfen.

Es wurde ein weiterer Factory-Konstruktor definiert, der eine JSON-Map als Argument nimmt und sie mit dem von Freezed generierten Code in ein User-Objekt umwandelt. Dies ist nützlich für das Parsen von Daten aus einer API oder einem lokalen Speicher.

Das Mixin _$WeatherHomeModel wird genutzt, um einige Funktionen von Freezed zu aktivieren, wie zum Beispiel copyWith und toString.

```
1  import 'package:freezed_annotation/freezed_annotation.dart';
2
3  part 'weather_model.freezed.dart';
4  part 'weather_model.g.dart';
5
6  @freezed
7  abstract class WeatherHomeModel with _$WeatherHomeModel {
8    factory WeatherHomeModel({
9      required WeatherModel currentTemperature,
10     required List<WeatherModel> forecasts,
11     required bool isLoading,
12     required bool hasError,
13   }) = _WeatherHomeModel;
14
15   factory WeatherHomeModel.fromJson(Map<String, dynamic> json) =>
16       _$WeatherHomeModelFromJson(json);
17 }
18
```

Figure 2.4: Weather Freezed Beispiel

In den Zeilen 3-4 ist die Anweisung definiert, welche die Ausgabedateien für den Code Generator bestimmt.

part 'weather_model.g.dart' Diese Zeile zeigt an, dass die aktuelle Datei Teil des Codegenerierungsprozesses für die Datei weather_model.dart ist, die von Freezed generiert wird. Das Suffix .freezed.dart zeigt an, dass die generierte Datei Code für die Funktionalität von Freezed enthalten wird.

part 'weather_model.g.dart Diese Zeile zeigt an, dass die aktuelle Datei auch Teil des Codegenerierungsprozesses ist, jedoch für den entsprechenden Serialisierungscode. Das Suffix .g.dart zeigt normalerweise an, dass die generierte Datei Code für die Codegenerierung (in diesem Fall für die Serialisierung) mit einem Tool wie json_serializable oder built_value enthält.

Zum Erstellen des Modells wird im Terminal folgender Befehl ausgeführt:

- **dart run build_runner build**

Nach der Ausführung des Befehls werden die Dateien mit der Endung model.freezed.dart und model.g.dart generiert, deren Bedeutung bereits oben erwähnt wurde.

2.5 View

Views sind alle Widgets und Seiten innerhalb der Flutter-Anwendung.

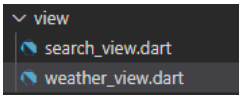


Figure 2.5: View Pages des Projekts

2.5.1 UI-Kit

Unter UI-Kits wurden Teile des Views zwecks Wiederverwendbarkeit ausgelagert. Die in der folgenden Abbildung angezeigten Kits, wurden in der Weather View, also der Hauptanzeige verwendet.

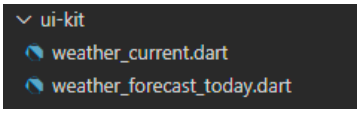
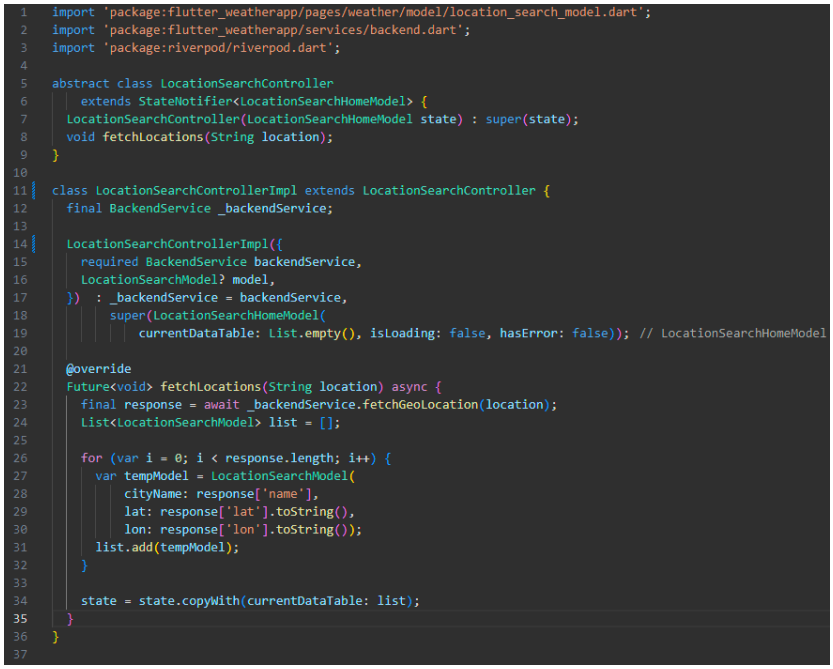


Figure 2.6: UI-Kits des Projekts

2.6 Controller

Die Controller-Schicht besteht aus High-Level-Funktionen, die eine bestimmte Art von Aufgabe ausführen. In der Regel rufen sie Daten der Services ab welche in die Modelle eingespeist werden, um so den Zustand der Anwendung zu kontrollieren und zu aktualisieren. Um dieses Ziel besser und flexibler als mit den Standardwerkzeugen Flutters zu realisieren, wird das Paket **Riverpod** verwendet, auf das im folgenden Kapitel näher eingegangen wird.



```dart
import 'package:flutter_weatherapp/pages/weather/model/location_search_model.dart';
import 'package:flutter_weatherapp/services/backend.dart';
import 'package:riverpod/riverpod.dart';

abstract class LocationSearchController
    extends StateNotifier<LocationSearchHomeModel> {
  LocationSearchController(LocationSearchHomeModel state) : super(state);
  void fetchLocations(String location);
}

class LocationSearchControllerImpl extends LocationSearchController {
  final BackendService _backendService;

  LocationSearchControllerImpl({
    required BackendService backendService,
    LocationSearchModel? model,
  })  : _backendService = backendService,
        super(LocationSearchHomeModel(
            currentDataTable: List.empty(), isLoading: false, hasError: false)); // LocationSearchHomeModel

  @override
  Future<void> fetchLocations(String location) async {
    final response = await _backendService.fetchGeoLocation(location);
    List<LocationSearchModel> list = [];

    for (var i = 0; i < response.length; i++) {
      var tempModel = LocationSearchModel(
          cityName: response['name'],
          lat: response['lat'].toString(),
          lon: response['lon'].toString());
      list.add(tempModel);
    }

    state = state.copyWith(currentDataTable: list);
  }
}
```

Figure 2.7: Beispiel: LocationSearchController

2.7 Riverpod

Riverpod ist eine leistungsstarke Zustandsverwaltungsbibliothek für Flutter, die auf Provider aufbaut und eine verfeinerte und flexible Möglichkeit zur Verwaltung des Anwendungszustands bietet. Im Vergleich zu Provider, welche die Standardbibliothek für Zustandsverwaltung ist, bietet Riverpod eine leichtgewichtige und robuste Lösung für die Handhabung von Zuständen und Dependency Injection in Flutter-Anwendungen.

Kapitel 3

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 3.1: Providers Definition für das Projekt

Im Grunde bringt Riverpod folgende Vorteile mit sich, welche auch im Projekt zu erkennen sind:

- **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.

3.1 Wie wird ein State Change durchgeführt?

Der State Change einer Page ist ausschließlich möglich über die Controller. In der Abbildung 2.7 ist zu sehen, wie der Controller definiert ist.

3.1.1 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(),
```

```
            lon: response['lon'].toString());
        list.add(tempModel);
      }

      state = state.copyWith(currentDataTable: list);
    }
  }

```

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.

3.1.2 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.

```
class SearchView extends ConsumerWidget {
  SearchView({Key? key, required this.current}) : super(key: key);
  TextEditingController textController = TextEditingController();
  final WeatherModel current;

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    List<LocationSearchModel> selectedData = [];
    final LocationSearchController controller =
        ref.read(providers.searchControllerProvider.notifier);
    final LocationSearchHomeModel model =
        ref.watch(providers.searchControllerProvider);
```

Figure 3.2: 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.

3.2 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 3.3: 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 3.4: 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 3.5: 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.