Projektarbeit:

Design Patterns in Cross Platform Apps für B2B App

von

Furkan Karayel 36372

Studiengang: Angewandte Informatik

RWU Hochschule Ravensburg-Weingarten

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einführung

In der schnelllebigen Welt der Softwareentwicklung ist es entscheidend, robuste und flexible Anwendungen zu erstellen. Insbesondere im Bereich mobiler und webbasierter Anwendungen hat sich das Framework Flutter von Google als leistungsstarke Lösung etabliert. Es bietet Entwicklern die Werkzeuge, um effiziente, plattformübergreifende Anwendungen zu entwickeln. Ein zentraler Aspekt bei der Entwicklung von Anwendungen ist das Zustandsmanagement. Es spielt eine entscheidende Rolle bei der Gewährleistung einer reibungslosen Benutzererfahrung.

Das vorliegende Projekt untersucht und vergleicht zwei prominente Zustandsmanagement-Lösungen innerhalb des Flutter-Ökosystems: Provider und Riverpod. Beide Ansätze bieten unterschiedliche Herangehensweisen und Werkzeuge zur Verwaltung des Anwendungszustands. Riverpod gilt als eine Weiterentwicklung von Provider und verspricht, einige der Einschränkungen und Herausforderungen von Provider zu überwinden. Das Ziel dieses Projekts ist es, die praktische Anwendung, Vor- und Nachteile sowie Unterschiede zwischen den beiden Zustandsmanagement-Strategien zu erforschen.

Als Fallstudie wurde eine Wetteranzeige-Applikation gewählt, da sie eine überschaubare Komplexität aufweist und gleichzeitig genügend Herausforderungen in Bezug auf das Zustandsmanagement und die Datenverarbeitung bietet. Diese Anwendung dient als Mittel, um die Konzepte und Techniken des Zustandsmanagements mit Provider und Riverpod zu demonstrieren und zu vergleichen. Sie verwendet die MVC+S-Architektur, eine Variation des klassischen Model-View-Controller (MVC)-Musters, die speziell für dieses Projekt angepasst wurde, um die Integration von Riverpod zu erleichtern und zu veranschaulichen.

Das Hauptziel dieses Projekts ist es, ein tiefgreifendes Verständnis für die Anwendung und das Management von Anwendungszuständen in Flutter zu entwickeln. Es wird untersucht, wie Riverpod im Vergleich zu Provider die Entwicklung von Anwendungen beeinflusst, die nicht nur funktional und benutzerfreundlich sind, sondern auch eine solide Architektur aufweisen, die Wartbarkeit, Testbarkeit und Erweiterbarkeit unterstützt. Die Analyse der Implementierung der Wetteranzeige-Applikation beleuchtet die Stärken und Schwächen beider Zustandsmanagement-Lösungen. Dadurch erhalten Entwickler eine fundierte Entscheidungsgrundlage für die Auswahl des geeigneten Tools für ihre Projekte.

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, worauf im Kapitel 3.2.2 eingegangen wird.

Der Controller enthält die gesamte Geschäftslogik und verarbeitet Nutzeraktionen oder Ereignisse aus der View-Schicht wie in der folgenden Abbildung 3.1 zu sehen ist. Die Services werden per Dependency Injection dem Controller eingespeist, worauf im Kapitel **??** eingegangen wird. Hier besteht die Möglichkeit auf die Nutzung von externen Backend Schnittstellen oder die Implementierung einer Persistenzschicht.

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.

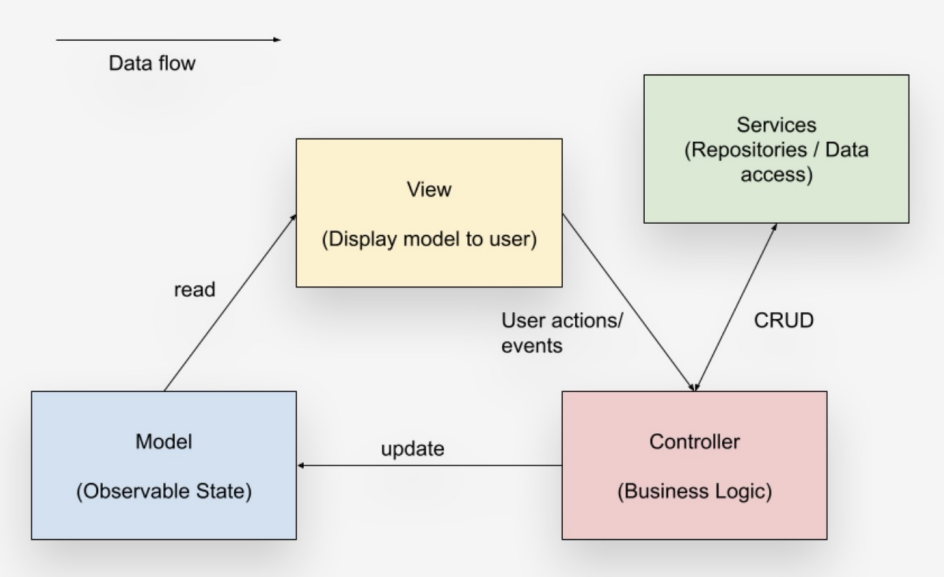

Figure 3.1: MVC+S Funktionsweise Flutter

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 durch striktere Typisierung und explizite Deklaration von Abhängigkeiten zwischen Komponenten hilft, viele Fehler bereits in der Entwicklungsphase zu erkennen und zu vermeiden.

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 erst ausgeführt wird. [2]

Riverpod Vorteile

- **Compile-Time-Sicherheit**: Frühzeitige Fehlererkennung während der Entwicklung noch vor der Code Ausführung.
- **Striktere Typisierung**: Verbesserte Code-Qualität und -Sicherheit.
- **Vermeidung von Laufzeitfehlern**: Im Gegensatz zu Provider, wo Fehler oft erst bei Ausführung entdeckt werden.

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. Beispiel: Dart-Backend.
- **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 bis zur Beendigung der App 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 manuell 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. Bei dauerhaft oder häufig genutzten Providern wäre es eine bessere Möglichkeit die Zustände manuell zu verwalten. Diese Option sollte mit Bedacht eingesetzt werden um unbeabsichtigte Effekte zu vermeiden. [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 animalFutureProvider = FutureProvider<Animal>((ref) async {
  return await AnimalService().getRandomAnimal();
});

class AnimalWidget extends ConsumerWidget {
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final animalFuture = ref.watch(animalFutureProvider);

    return animalFuture.when(
          loading: () => const Text("Loading animal..."),
          error: (err, stack) => const Text("Error loading animal"),
          data: (animal) => Column(children: [
                Text(animal.name),
                Text(animal.scientificName),
```

```
16                      Text(animal.type),
17                      Text(animal.weight),
18                      Text(animal.origin),
19                    ]));
20        }
21      }
22
```

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. Implementierung der Wetteranwendung mit Riverpod

In diesem Kapitel wird die Implementierung einer Wetteranwendung mit Riverpod detailliert beschrieben. Die Anwendung besteht aus zwei Hauptbildschirmen: dem Startbildschirm (Home Screen) und dem Standortsuchbildschirm (Location Search Screen). Beide Bildschirme nutzen die Riverpod-Bibliothek für das Zustandsmanagement und die Architektur der Anwendung folgt dem MVC+S-Muster, wobei Riverpod zur Verwaltung des Zustands eingesetzt wird.

4.1 Startbildschirm (Home Screen)

Der Startbildschirm ist die Hauptansicht der Anwendung, die aktuelle Wetterdaten für einen voreingestellten oder vom Benutzer ausgewählten Standort anzeigt. Die Anzeige umfasst die aktuelle Temperatur, Wetterbedingungen und eine stündliche Vorhersage für den aktuellen Tag.

Figure 4.1: Startbildschirm

4.1.1 Zustandsmanagement

Für das Zustandsmanagement des Startbildschirms wird ein ‘StateNotifierProvider‘ verwendet, der eine Instanz von ‘WeatherController‘ bereitstellt. Der ‘WeatherController‘ ist für das Abrufen der Wetterdaten vom Backend-Service zuständig und aktualisiert den Zustand der Anwendung entsprechend. Der Zustand wird in einem ‘WeatherHomeModel‘ gespeichert, das die aktuellen Wetterdaten und den Ladestatus enthält.

4.1.2 UI-Komponenten

Die Benutzeroberfläche des Startbildschirms besteht aus mehreren wiederverwendbaren Widgets, darunter ‘WeatherCurrent‘ für die Anzeige der aktuellen Wetterdaten und ‘WeatherForecastToday‘ für die stündliche Vorhersage. Diese Widgets beobachten den Zustand des ‘WeatherController‘ und aktualisieren sich automatisch, wenn sich die Wetterdaten ändern.

4.1.3 Beispiel: Zustandsmanagement für den Startbildschirm

Der Startbildschirm zeigt die aktuellen Wetterdaten an, einschließlich Temperatur, Bedingungen und stündlicher Vorhersagen. Die Zustandsverwaltung für diesen Bildschirm umfasst das Abrufen von Wetterdaten und die entsprechende Aktualisierung der Benutzeroberfläche.
Im Folgenden wird erklärt, wie die Zustandsverwaltung eines Screens erstellt und verwendet wird. Zunächst werden die Schritte erläutert, worauf immer ein beispielhafter Codeabschnitt folgt. Um eine bessere Verständlichkeit und Übersichtlichkeit zu gewährleisten, wird der verwendete Code teilweise als Pseudocode dargestellt.

Zustandsdefinition

Zunächst sollte der Zustand festgelegt werden, in dem die Wetterdaten gespeichert werden sollen. Dies beinhaltet sowohl das aktuelle Wetter, die Vorhersagen den Ladestatus als auch den Fehlerstatus.

```
@freezed
abstract class WeatherHomeState with _$WeatherHomeState {
  factory WeatherHomeState({
    required WeatherModel currentWeather,
    required List<WeatherModel> hourlyForecast,
    required bool isLoading,
    required bool hasError,
  }) = _WeatherHomeState;
}
```

StateNotifier für die Wetterdaten

Darauf folgt die Erstellung eines 'StateNotifier', der die Wetterdaten abruft und verwaltet. Dieser Controller aktualisiert den Zustand basierend auf den Ergebnissen der API-Aufrufe.

```
class WeatherController extends StateNotifier<WeatherHomeState> {
  final BackendService _backendService;

  Future<void> fetchWeather(String city) async {
    state = state.copyWith(isLoading: true);
    try {
      final currentWeather = await _backendService.fetchCurrentWeather(city);
      final hourlyForecast = await _backendService.fetchHourlyForecast(city);
      state = state.copyWith(
        isLoading: false,
        hasError: false,
        currentWeather: currentWeather,
        hourlyForecast: hourlyForecast,
      );
    } catch (e) {
      state = state.copyWith(isLoading: false, hasError: true);
    }
  }
}
```

Verwendung des StateNotifierProvider

In diesem Schritt wird der ‘StateNotifierProvider‘ erzeugt, um eine Instanz von ‘WeatherController‘ bereitzustellen.

```
final weatherControllerProvider = StateNotifierProvider<WeatherController, WeatherHomeState>((ref) {
  return WeatherController(ref.watch(weatherRepositoryProvider));
});
```

UI Integration

Im Widget ’Location Search Screen’ sollte der ’LocationSearchControllerProvider’ verwendet werden, um die Suchanfrage zu verwalten und die Ergebnisse anzuzeigen.

```
class LocationSearchScreen extends ConsumerWidget {
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final state = ref.watch(locationSearchControllerProvider);
    final controller = ref.read(locationSearchControllerProvider.notifier);

    return Scaffold(
      appBar: AppBar(title: Text('Search Location')),
      body: Column(
        children: [
          TextField(
            onChanged: (query) => controller.searchLocation(query),
            decoration: InputDecoration(labelText: 'Search'),
          ),
          state.isLoading
              ? CircularProgressIndicator()
              : state.hasError != null
                  ? Text('Error: ${state.hasError}')
                  : Expanded(child: LocationList(locations: state.searchResults)),
        ],
      ),
    );
  }
}
```

4.2 Standortsuchbildschirm (Location Search Screen)

Der Standortsuchbildschirm ermöglicht es dem Benutzer, nach Städten zu suchen und das Wetter für einen ausgewählten Standort anzuzeigen. Die Suche wird über eine externe API realisiert, die Standortdaten basierend auf dem Suchbegriff des Benutzers liefert.

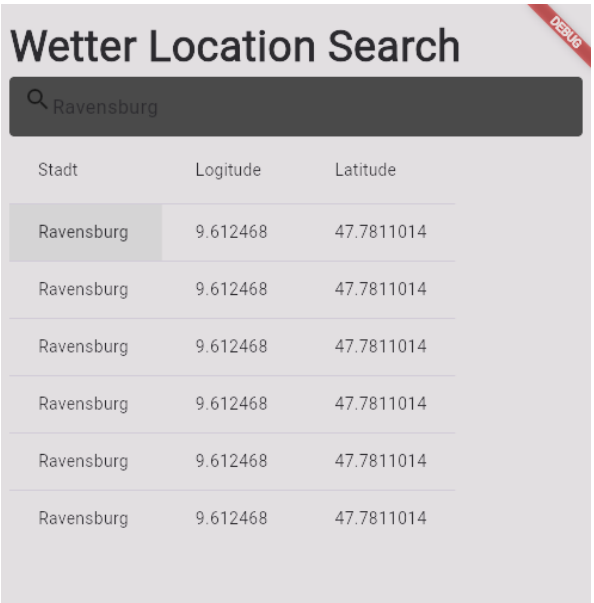

Figure 4.2: Standortsuchebildschirm

4.2.1 Zustandsmanagement

Ähnlich wie beim Startbildschirm wird auch hier ein ‘StateNotifierProvider‘ eingesetzt, der eine Instanz von ‘LocationSearchController‘ bereitstellt. Der ‘LocationSearchController‘ verwaltet den Suchvorgang und den Zustand der Suchergebnisse, die in einem ‘LocationSearchHomeModel‘ gespeichert werden.

4.2.2 UI-Komponenten

Die Benutzeroberfläche des Standortsuchbildschirms besteht aus einem Suchfeld und einer Tabelle, die die Suchergebnisse anzeigt. Benutzer können einen Standort aus den Suchergebnissen auswählen, woraufhin die Anwendung zum Startbildschirm zurückkehrt und das Wetter für den ausgewählten Standort anzeigt.

4.2.3 Beispiel: Zustandsmanagement für den Standortsuchscreen

Auf dem Bildschirm ’Standortsuche’ können Benutzer nach Städten suchen und eine auswählen, um die zugehörigen Wetterdaten anzuzeigen. Die Statusverwaltung für diesen Bildschirm umfasst die Bearbeitung der Suchanfrage und die Anzeige der Suchergebnisse.
Im Folgenden wird erklärt, wie die Zustandsverwaltung eines Screens erstellt und verwendet wird. Zunächst werden die Schritte erläutert, worauf immer ein beispielhafter Codeabschnitt folgt. Um eine bessere Verständlichkeit und Übersichtlichkeit zu gewährleisten, wird der verwendete Code teilweise als Pseudocode dargestellt.

Zustandsdefinition

Der Zustand wird definiert, in dem die Suchergebnisse, der Ladestatus und der Fehlerstatus darin gespeichert werden.

```
@freezed
abstract class LocationSearchHomeState with _$LocationSearchHomeState {
  factory LocationSearchHomeState({
    required List<LocationSearchModel> searchResults,
    @Default(false) bool isLoading,
    @Default(false) bool hasError,
  }) = _LocationSearchHomeState;
}
```

StateNotifier für die Standortsuche

Es wird ein 'StateNotifier' erstellt, der den Suchvorgang verwaltet und den Status mit den Ergebnissen aktualisiert.

```
class LocationSearchController extends StateNotifier<LocationSearchState> {
  final LocationService locationService;

  LocationSearchController(this.locationService) : super(LocationSearchState());

  Future<void> searchLocation(String query) async {
    state = state.copyWith(isLoading: true);
    try {
      final results = await locationService.searchLocation(query);
      state = state.copyWith(isLoading: false,  hasError: false, searchResults: results);
    } catch (e) {
      state = state.copyWith(isLoading: false, hasError: true);
    }
  }
}
```

Verwendung des StateNotifierProvider

In diesem Schritt wird der 'StateNotifierProvider' erzeugt, um eine Instanz von 'WeatherController' bereitzustellen.

```
final searchControllerProvider = StateNotifierProvider<LocationSearchController, LocationSearchHomeSt
  return WeatherController(ref.watch(weatherRepositoryProvider));
});
```

UI Integration

Im Startbildschirm-Widget wird der 'weatherControllerProvider' verwendet, um Wetterdaten abzurufen und die Benutzeroberfläche basierend auf dem Zustand zu erstellen.

```
class LocationSearchScreen extends ConsumerWidget {
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    final state = ref.watch(locationSearchControllerProvider);
    final controller = ref.read(locationSearchControllerProvider.notifier);
```

```
6
7       return Scaffold(
8         appBar: AppBar(title: Text('Search Location')),
9         body: Column(
10          children: [
11            TextField(
12              onChanged: (query) => controller.searchLocation(query),
13              decoration: InputDecoration(labelText: 'Search'),
14            ),
15            state.isLoading
16                ? CircularProgressIndicator()
17                : state.hasError != null
18                    ? Text('Error: ${state.hasError}')
19                    : Expanded(child: LocationList(locations: state.searchResults)),
20          ],
21        ),
22      );
23    }
24  }
```

4.3 Navigation zwischen den Bildschirmen

Die Navigation zwischen dem Startbildschirm und dem Standortsuchbildschirm wird durch Flutter's Navigator realisiert. Wenn ein Benutzer einen Standort aus den Suchergebnissen auswählt, wird der 'WeatherController' mit dem ausgewählten Standort aktualisiert und der Benutzer wird zum Startbildschirm navigiert, wo die Wetterdaten für den neuen Standort angezeigt werden.

4.4 Zusammenfassung

Die Implementierung der Wetteranwendung mit Riverpod demonstriert die Flexibilität und Effizienz von Riverpod für das Zustandsmanagement in Flutter-Anwendungen. Durch die Verwendung des MVC+S-Musters und die klare Trennung von Zustandsmanagement, Logik und Benutzeroberfläche wird die Wartbarkeit und Erweiterbarkeit der Anwendung verbessert. Die Anwendung bietet eine reaktive Benutzererfahrung, indem sie automatisch auf Zustandsänderungen reagiert und die Benutzeroberfläche entsprechend aktualisiert.

5. Fazit

Die vorliegende Projektarbeit vergleicht zwei führende Zustandsmanagement-Lösungen im Flutter-Ökosystem, nämlich Provider und Riverpod. Dabei wird die Eignung dieser Technologien für die Realisierung einer effizienten und benutzerfreundlichen Wetteranzeige-Applikation evaluiert. Die Implementierung der Applikation mit der MVC+S-Architektur hat gezeigt, dass Riverpod signifikante Vorteile gegenüber dem traditionellen Provider-Ansatz bietet, insbesondere in Bezug auf Flexibilität, Sicherheit und Testbarkeit.

Einer der herausragenden Aspekte von Riverpod ist seine Fähigkeit, eine Compile-Time-Sicherheit zu gewährleisten, die Entwicklern hilft, Fehler frühzeitig im Entwicklungsprozess zu erkennen. Diese Eigenschaft, kombiniert mit der Unabhängigkeit von Flutter und der Entkopplung vom Widget-Baum, macht Riverpod zu einem mächtigen Werkzeug für das Zustandsmanagement in Flutter-Anwendungen. Die Implementierung der Wetteranzeige-Applikation hat zudem die Vorteile einer klaren Trennung zwischen der Geschäftslogik, der Darstellung und dem Zustandsmanagement unter Beweis gestellt, was zu einer verbesserten Wartbarkeit und Erweiterbarkeit der Anwendung führt.

Provider bleibt ein solides Werkzeug für das Zustandsmanagement in Flutter-Anwendungen. Allerdings hat die Untersuchung gezeigt, dass Riverpod in vielen Aspekten überlegen ist, insbesondere bei komplexen Anwendungen mit umfangreichen Zustandsmanagement-Anforderungen. Riverpod ermöglicht es, Zustände effizient zu verwalten, zu testen und zu debuggen, ohne dabei auf die Struktur des Widget-Baums angewiesen zu sein. Dies stellt einen bedeutenden Fortschritt dar.

Zusammenfassend kann festgestellt werden, dass die Wahl des Zustandsmanagement-Tools stark von den spezifischen Anforderungen des Projekts und den Präferenzen des Entwicklerteams abhängt. Riverpod bietet jedoch durch seine fortschrittlichen Funktionen und seine Flexibilität einen deutlichen Mehrwert für die Entwicklung zukunftssicherer Flutter-Anwendungen. Die Erkenntnisse aus diesem Projekt können Entwicklern als Leitfaden dienen, um fundierte Entscheidungen über das Zustandsmanagement in ihren eigenen Projekten zu treffen und die Vorteile von Riverpod voll auszuschöpfen.

Insgesamt hat diese Projektarbeit wertvolle Einblicke in die praktische Anwendung von Zustandsmanagement-Lösungen in Flutter geliefert und die Grundlage für weiterführende Untersuchungen und Entwicklungen in diesem spannenden und dynamischen Bereich der Softwareentwicklung gelegt.

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).