Skatify - Dokumentation

Einführung

Aufgabenstellung

Zielstellung war es eine Skat Bewertungsapp zu erstellen. Die App sollte verschiedene Funktionen integrieren. So sollten nicht nur die Punkte der einzelnen Spieler angezeigt, sondern auch mittels verschiedener Eingaben die Punkte pro Runde ermittelt werden (Prinzipiell ein Punkte "Wizzard").

Im zweiten Schritt sollte es dann möglich sein, ein Turnier zu starten und mehrere Geräte lokal miteinander zu verbinden.

Vorgehensweise

Das Vorgehen kann in zwei Phasen geteilt werden. In der ersten wurde das Vorgehen und die Rahmenbedingungen festgelegt. In der zweiten ging es an die praktische Umsetzung.

Für die Planung wurde mit Hilfe von Trello ein Storyboard und ein Kanbanboard angelegt. Auf dem Storyboard wurden die einzelnen Aufgaben in verschiedene Themengebiete eingeteilt. Die Funktionen sind in die Bereiche "Konfiguration", "Elemente erzeugen", "Daten anzeigen" und "Speicher" unterteilt. Des Weiteren wurde definiert, welche Aufgaben in MVP 1 und welche in MVP 2 fallen.

Alle Funktionen bezüglich "Turnier erstellen und abhalten" wurden als MVP 2 deklariert.

Nach dem Verteilen der größeren Aufgabenblöcke wurde die theoretische UI-Struktur (User Interface) der App betrachtet. Mittels der Aufgabenblöcke (User Stories und Storyboard) und der UI wurde das Kanban mit Tickets versehen.

Generell beinhaltet das Kanban vier verschiedene Spalten.

1. Todo - Sammeln der Aufgaben, welche noch nicht begonnen wurden
2. Doing - alle momentan bearbeiteten Aufgaben (max. 2, da nur zwei Personen)

3. Ongoing - Tickets welche dauerhaft verändert werden, während andere Tickets bearbeitet werden
4. Resolved - Entwicklung prinzipiell abgeschlossen -> anderer Entwickler schaut nochmal drüber
5. Done - abgeschlossene Aufgaben

Es wurden Tickets für verschiedene Felder hinzugefügt. So gab es hauptsächlich UI-Tickets, welche das Erstellen der verschiedenen Bildschirme beschrieben. Aber auch technische Tickets, wie das Erstellen eines Spielobjekts für die Spiellogik.

Nach der theoretischen Überlegung erfolgt die praktische Umsetzung. Dazu wurde ein Github Repository erstellt, um die Dokumentation und den Code zugänglich zu machen. Die Dokumentation ist auch über maximilian-skowron.github.io erreichbar. — layout: default title: "02 - User Interface" nav_order: 2 permalink: /ui —

UI Planung

Während der Planung der Nutzeroberfläche wurde folgender Pretotype erstellt:

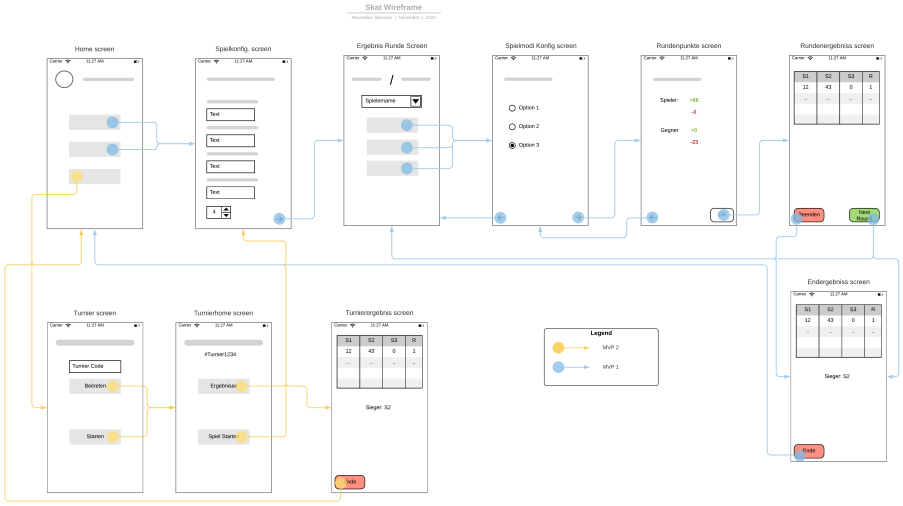

Figure 1: Skatify Pretotype

Der Pretotype zeigt die einzelnen Bildschirme der App und wie man zwischen ihnen wechselt. Die Pfeile fungieren dabei als Verlaufselemente und weisen einzelne Bildschirme bzw. Funktionen einem MVP zu.

Besonders wichtig war die Farbkodierung einiger Elemente, wie z.B. der

“Beenden” und “Nächste Runde” Schaltflächen. Dem Nutzer sollte vermittelt werden, dass eine Interaktion mit der “Beenden” Schaltfläche zwangsläufig zu einem Ende des Spielablaufs führte. — layout: default title: “03 - Planung der praktischen Umsetzung” nav_order: 3 permalink: /practical-solution-planing —

Praktische Umsetzung Planung

Bevor mit der Umsetzung begonnen werden kann muss evaluiert werden, welche Tools für mobile Apps verwendet werden könne. Generell gibt es vier goße Tools:

1. QT
2. React Native
3. Flutter
4. native Java/Kotlin/Swift/Object-C

Im folgenden werden kurz die Eigenheiten einer jeden Vorgehensweise kurz beschrieben und eine eigene Meinung dazu gewidmet. In darauf folgenden Kapiteln wird auf den Unterschied zwischen nativer Android Entwicklung (Java/Kotlin) und Flutter genauer eingegangen.

QT

QT ist das älteste Mittel zum erstellen von cross platform Applikationen. Die neuste Version bietet verschiedene Möglichkeiten eine App zu erstellen. Am meisten genutzt wird QT Quick mit QML, eine Javascript ähnlichen Sprache, und QT+ mit C++ als Sprache verwendet.

Bei QT handelt es sich, unabhängig von der gewählten Vorgehensweise, um ein Framework zum erstellen von Nutzeroberflächen und entsprechender Logik für Android, IOS, Linux und Windows.

Für QT sollten gute C++ Kenntnisse vorhanden sein, da QML dazu tendiert “Spagetthi”-Code zu produzieren. Somit bleibt einem, wenn man eine möglichst große Widgetauswahl haben möchte mit lesbarem Code, nur QT+ als Variante.

React Native

React Native ist ein Webframework, welches eine mittels HTML, CSS und Javascript innerhalb eines Browsers nativ auf verschiedenen Geräten laufen kann. So muss wie auch bei QT kein Plattform spezifischer Code geschriebene werden.

React Native ist recht Ressourcen kostspielig und kann nur auf die vom React Team implementierten APIs der Zielplattform zugreifen. Allerdings verwendet

es das React Framework und bietet somit Entwicklern mit Frontend oder React Erfahrung die Möglichkeit schnell eigene Apps zu erstellen.

Hinsichtlich dem MVP 2 wurde React Native ausgeschlossen, da nicht bekannt war wie der Zugriff auf Bluetooth oder ähnliches erfolgen kann.

native Entwicklung

Unter IOS kann entweder mit dem älteren Object C oder Swift entwickelt werden. Bei Android kann Java oder Kotlin vom Entwickler verwendet werden. Innerhalb der Umsetzung wurde sich auf die Androidentwicklung konzentriert.

Generell bieten native Apps die beste Performance. Allerdings sind sie nicht cross plattform kompatible mit Ausnahme von Kotlin was im Februar bekannt gegeben hat Multiplattformsupport zu erhalten.

Anfänglich während der Umsetzung wurde sich für Java entschieden, da ein Java Background vorhanden war. Später wurde aber auf Flutter umgestiegen. Entsprechende Gründe und Erkenntnisse werden in folgenden Kapiteln beschrieben.

Flutter

Flutter ist ein Multiplattform Framework zum erstellen von IOS, Android Apps, Webseiten und Desktopanwendungen für Linux und Windows. Es verwendet die von Google entwickelte Sprache Dart.

Flutter kommt in den Feldern von IOS und Android Apps nah an die Performance nativer Apps heran. Durch seine neue und durchdachte Programmiersprache ermöglicht es einfach und schnell Logik in die App zu integrieren. Es bietet ein riesieges Sammelsorium an Widgets mit denen schnell eine funktionale UI erstellt werden kann.

Entgegen Android Entwicklung mit Java/Kotlin gibt es bei Flutter viele Möglichkeiten UI-State zu managen. Vom Flutter Team direkt wird oft die Nutzung des Provider Packages empfholen. Aber auch dieses hat seine Schwäche und wirkt noch recht unausgereift. Ein genauer Vergleich der Möglichkeiten kann hier nachgelesen werden.

Da unter Android die model-view-viewmodel Archtiketur nicht nur empfholen, sondern auch durch entsprechende Packages gut unterstützt ist wurde sich gegen Flutter entschieden.

Java praktische Erfahrung während der Umsetzung

Einrichtung der Entwicklungsumgebung

Für die native Androidentwicklung muss Android Studio heruntergeladen und installiert werden. Bei der Installation werden alle benötigten Java Versionen automatisch heruntergeladen. Für die Entwicklung sollte ein Emulator eingerichtet werden. Dafür muss eine SDK (spezifische Android Version) während der Installation von Android Studio heruntergeladen werden. Es muss bedacht werden je neuer die SDK umso neuere Funktionen sind vorhanden, allerdings werden weniger Endgeräte abgedeckt. Der Installationswizzard fragt dann noch danach ein AVD (Android Virtual Device) zu erstellen mit einer vorhandenen SDK.

Es kann auch anstelle eines Emulators das eigene Smartphone mittels eines USB Kabels angeschlossen und genutzt werden.

Danach kann in Android Studio ein neues Projekt angelegt werden. Dabei kann zwischen verschiedenen Presets ausgewählt werden. Man muss einen Name und ein Javapackage, sowie eine minimale SDK festlegen.

Android Entwicklung laut dem Android Team

Das Team rund um Android gibt neuen Android Entwicklern eine klare Richtung vor. So wird mit Android Jetpack (auch AndroidX) ein Framework bzw. eine Sammlung von Tools zur Verfügung gestellt, um skalierbare und testbare Apps zu erstellen. Jetpack selbst integriert dabei einige Grundsätze, wie `Seperation of Concerns` und `UI driven by Models`. Dazu kommt vor allem das Model-View-ViewModel (MVVM) Design Pattern zum Einsatz.

MVVM selbst wird mittels mehrere Tools innerhalb des Frameworks umgesetzt. Es wird das Erstellen und Managen von UI Elementen in eigene XML Dateien ausgelagert. Innerhalb diser XMLs kann mittels "Data Binding" Werte aus den ViewModels angezeigt bzw. manipuliert werden. Das Navigieren zwischen UI Elementen wird von einee Navigations Komponente übernommen, welche wieder ein eigenständiges Tool innerhalb des Frameworks ist.

Schließlich wird folgender Aufbau versucht zu erreichen:

Activities oder Fragmente binden verschiedene XML Dateien und stellen die Schnittstelle zwischen Code und UI dar. Sie ermöglichen das managen vom Lebenszyklus der Komponenten und somit direkt das managen von allokierten Ressourcen.

Über Ihnen stehen ViewModels, welche den State einzelner Elemente oder des ganzen Bildschirms halten. Zum Beispiel gibt es einen Bildschirm mit mehreren Texteingabe Elementen, deren Werte werden im ViewModel in Variablen gehalten und wenn nötigt transformiert. Das ViewModel ist prinzipiell nichts weiter als ein standard Model nur für UI Elemente.

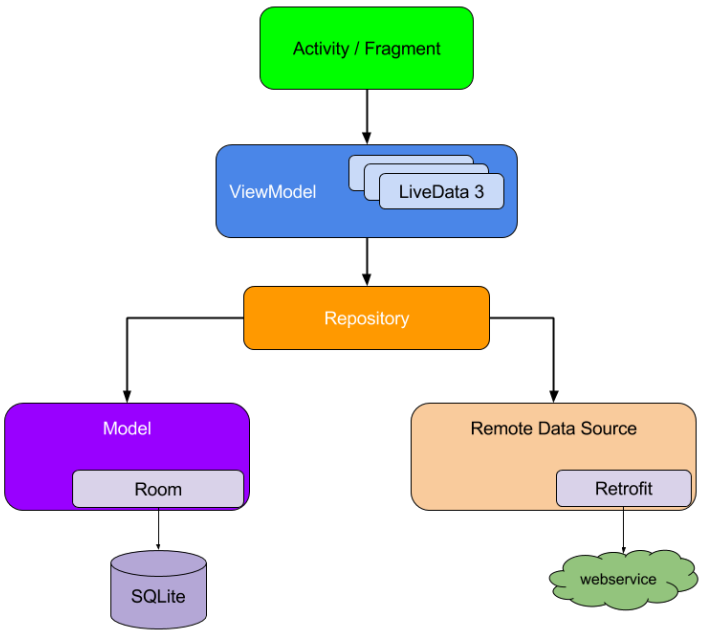

Figure 2: Android Jetpack App Structure

Unterhalb des ViewModls kommt das Repository Pattern zum Einsatz. Ein Repository ist einfach eine Sammlung von DAO Objekten bzw. ein es stellt selber ein DAO Objekt dar nur das nicht nur eine Datenquelle dahinter steht. So könnte eine Funktion in einem Repository heißen `saveUser()`. Diese Versucht den Nutzer in der Cloud, also einer `remote data source`, zu speichern. Sollte dies nicht gelingen wird entsprechender Nutzer erstmal lokal gespeichert. Ein Controller oder ViewModel muss dabei keinerlei Informationen über mögliche DB Fehler besitzen.

Zu letzt wird durch Jetpack eine Beispiel Sunflower App zur Verfügung gestellt, wo die verschiedenen AndroidX Tools aufgezeigt werden.

Java App Entwicklung und deren Probleme

Noch vor dem eigentlichen Start der Entwicklung kamen Probleme auf. So sind viele Tutorials der Android Macher als auch die Sunflower App nur mit Kotlin erstellt wurden. Kotlin ist die zweite Programmiersprache, welche neben Java vollständig auf Android unterstützt wird. Zwar gibt es einige Überschneidungen innerhalb der Idiome von Kotlin und Java, allerdings war die Syntax doch stark andersartig.

Obwohl Kotlin Idiome wie `functional programming` anbietet wurde sich für Java als Basis entschieden, da die Zeit nicht ausreichenwürde eine neue Sprache zu lernen.

Allerdings stellte sich eine geringe Java Dokumentationsdichte als Problem heraus und führte nur von Fehler zu Fehler. Denn auch innerhalb der Android Community wird fast ausschließlich nur noch Kotlin verwendet, sodass Fehler langsam und mühsählig gelöst werden mussten.

Da dies zu lange gedauert hat wurde sich entschiedenen auf Flutter umzusteigen.

Flutter vs Java

Generelle Unterschiede

Flutter ist Googels neustes Framework, welches es ermöglicht mit einer Code Basis (Dart Programmierungssprache) für Desktop, Web und Mobil Apps zu entwickeln. Dafür wird Dart Code in nativen Android Code generiert und schließlich als APK verpackt. Flutter selbst verbindet UI und Code durch sogenannte Widget. Diese Widget bieten viele verschiedene Funktionen und können in einem sogenannten Widget Baum verschachtelt und ergeben somit die UI der App.

Ein simples Beispiel wäre folgender Code:

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

```
void main() {
  runApp(MaterialApp(
    title: 'Flutter Tutorial',
    home: TutorialHome(),
  ));
}

class TutorialHome extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    // Scaffold is a layout for the major Material Components.
    return Scaffold(
      appBar: AppBar(
        leading: IconButton(
          icon: Icon(Icons.menu),
          tooltip: 'Navigation menu',
          onPressed: null,
        ),
        title: Text('Example title'),
        actions: <Widget>[
          IconButton(
            icon: Icon(Icons.search),
            tooltip: 'Search',
            onPressed: null,
          ),
        ],
      ),
      // body is the majority of the screen.
      body: Center(
        child: Text('Hello, world!'),
      ),
      floatingActionButton: FloatingActionButton(
        tooltip: 'Add', // used by assistive technologies
        child: Icon(Icons.add),
        onPressed: null,
      ),
    );
  }
}
```

Wenn genauer geschaut wird erkennt man ein `onPressed` Attribut beim `FloatingActionButton`. Dieses Attribute übernimmt eine simple Funktion und führt den Code darin aus. Das verführt natürlich zum managen des States innerhalb der UI allerdings bietet das Flutter Team auch eine Jetpack ähnliche Lösung für Statemanagement an. Die Entwickler sollen Providers nutzen. Provider ist

ein Wrapper um die sogennanten Inherited Widget. Ein Provider kann somit in jedem Widget unterhalb des eigentliches Providers aufgerufen und genutzt werden.

Sodass schließlich eine solche Struktur realisiert werden kann:

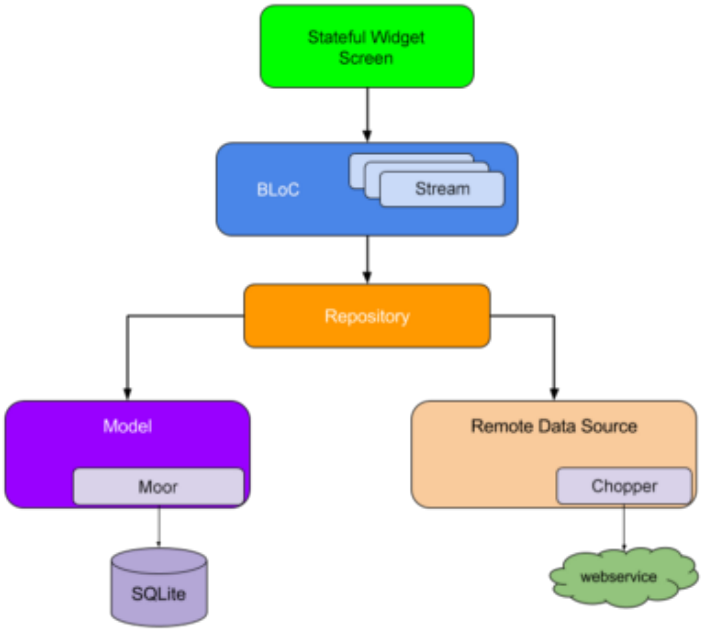

Figure 3: Flutter App Struktur

`BLoC` ist eine weitere State Management Möglichkeit um sogenannte Streams und Sinks.

Flutter vs native Java Vor- und Nachteile

	Flutter	native Android
APK Größe	größer als bei Java	mit Java ist die kleinstmögliche App Größe erreichbar
Statemanagement	Provider, Bloc, Inherited Widget	AndroidX mit MVVM
Sprache	Dart (oft als Javascript 2.0 bezeichnet)	Java oder Kotlin

	Flutter	native Android
Entwicklungsteams	Extrem viele Beispiele durch die Entwickler	Bei Jetpack leider fasst ausschließlich nur Kotlin Beispiele
Support für Beispiele		
Support durch Community	Extrem groß, viele Medium Artikel, viele Tutorials	fasst nur Kotlin generell weniger beliebt als Flutter
Geschwindigkeit	Leicht langsamer als native Apps, durch schlechte Optimierung Tendenz extrem langsam zu werden	Schnellstmögliche App
Entwicklungsumgebung	Visual Studio Code (Flutter Extension), Android Studio (Flutter + Dart Extension)	Android Studio, auch VSCode (eher weniger genutzt)

Flutter praktische Erfahrung während der Umsetzung

Einrichten der Entwicklungsumgebung

Auch wenn Visual Studio Code (VSCode) zum entwicklen genutzt werden soll muss Android Studio installiert werden, insofern die Apps innerhalb eines Emulators ausgeführt werden soll. Es ist auch möglich Apps auf dem eigenen Handy auszuführen und zu testen. Sollte ein Emulator verwendet werden muss unter SDK eine Android Version installiert werden und unter AVD (Android Virtual Device) ein neuer Emulator angelegt werden.

Innerhalb von VSCode muss dann nur noch die Flutter Extension installiert werden.

Installieren von Flutter

Flutter selbst ist einfach zu installieren und auch gut für jede Plattform (Windows, Mac, Linux, ChromeOS) beschrieben.

https://flutter.dev/docs/get-started/install

Projekt erstellen

Wurde Flutter richtig erstellt kann mit dem `flutter` CLI Tool ein Projekt angelegt werden. Es sollte vorher mit `flutter doctor` geschaut werden ob alle Extensions und Installationen richtig erkannt wurden. Die Rückgabe sollte ungefähr so aussehen, wo bei Xcode bei Linux, Chrome OS und Windows wegfällt.

```
$ flutter doctor

Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, 1.20.2, on Mac OS X 10.15.7 19H2, locale de-DE)

[✓] Android toolchain - develop for Android devices (Android SDK version 29.0.3)
[✓] Xcode - develop for iOS and macOS (Xcode 12.1)
[✓] Android Studio (version 4.0)
[✓] VS Code (version 1.50.1)
[!] Connected device
    ! No devices available

! Doctor found issues in 1 category.
```

Nun kann schließlich mit `flutter create --project-name FlutterProject /Pfad/Zum/Projekt/Ordner` ein neues Beispiel Projekt erstellt werden. Entweder über VSCode oder mittels `flutter run` kann die App ausgeführt werden.

Erfahrung während der Umsetzung

Das Aufsetzen von Flutter beinhaltet zwar wenige Schritte mehr als bei Android, da Android Studio alle nötigen Dateien selbst installiert, allerdings war es trotzdem sehr einfach.

Flutter selbst ermöglichte uns extrem schnell und einfach die Hälfte der Bildschirme aus dem vorher definierten Pretotype zu erstellen und mit Funktionalität zu versehen. Dies war dem Baukastensystem der Widget und dem Hot Reloade geschuldet, welcher die App fast sofort aktualisierte anstatt komplett neu zu bauen, wie bei Android.

Im folgenden Kapitel gehen wir auf den Flutter Code ein und ziehen dann im letzen Kapitel ein Fazit zu Flutter und Android Native.

Skatify Code mit Flutter

Struktureller Aufbau der App

Innerhalb des `lib` Ordners befinden sich alle Dateien bezüglich der UI und der Funktion der App. Folgende Ordnerstruktur wurde verwendet:

```
.
├── configs
│   ├── routes.dart
│   └── service_locator.dart
├── main.dart
├── models
```

```
│   ├── game.dart
│   └── player.dart
├── services
│   └── game_service.dart
├── viewmodel
│   ├── gameconfigprovider.dart
│   └── roundresultprovider.dart
└── views
    ├── fallbackview.dart
    ├── gameconfigview.dart
    ├── homeview.dart
    └── resultroundview.dart
```

Der `views` Ordner beinhaltet die einzelnen Bildschirme, welche im Pretotype definiert wurden. `Viewmodel` ist der Ordner welche die Flutter Provider hält, welche quasi die ViewModels von Flutter sind. Dort wird der State gemanagt und verschiedene Services integriert und ausgeführt. Die Services sind im `services` Ordner zu finden. Services beinhalten Logik, wie der `GameService` welcher Models bezüglich des Spiels und der Spieler hält und zum starten, stoppen von Spielen zuständig ist. Entsprechende Models sind im `models` Ordner zu finden.

Unterhalb des `configs` Ordner existieren globale Konfigurationsobjekte. Die Datei `routes.dart` definiert Pfade innerhalb der App, also verschiedene Bildschirme und ermöglicht es anderen Bildschirmen in den nächsten zu wechseln. Der Service Locator hingegen ist ein Tool, welches das erstellen von Factories und Singeltons vereinfacht. Es werden nun nicht mehr direkt Services instanziert, sondern mittels dem Service Locator geholt. Dieser abstrahiert z.B. das Erstellen von Singletons weg, sodass z.B. im `GameService` keinerlei Code erstellt werden muss, um zu garantieren, dass nicht zwei Instanzen erstellt werden.

`main.dart` stellt den Eingangspunkt der App dar.

```
import .....

void main() {
 // setup and initialize locator bevore app starts to prevent calls before init
  setupLocator();
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MultiProvider(
      providers: [
        ChangeNotifierProvider(
```

```
        create: (_) => GameConfigProvider(getIt.get<GameService>())),
        ChangeNotifierProvider(
        create: (_) => RoundResultProvider(getIt.get<GameService>())),
      ],
      child: MaterialApp(
        title: 'Skatify',
        debugShowCheckedModeBanner: true,
        theme: ThemeData(
          primarySwatch: Colors.green,
          visualDensity: VisualDensity.adaptivePlatformDensity,
        ),
        onGenerateRoute: onGenerateRoute,
        initialRoute: '/home',
      ),
    );
  }
}
```

Hier werden mittels eines `MultiProvider` die verschiedenen Provider bereitgestellt und ihnen wird der `GameService` übergeben, welcher vom Service Locator als Singleton erstellt wurde.

Danach wird eine `MaterialApp` angelegt, diese gibt allen Widgets unterhalb ein Material Aussehen. Des Weiteren wird mit der Material App eine Funktion angegeben `onGenerateRoute`, welche die Wege zwischen den Bildschirmen handhabt.

Provider am Beispiel der GameConfigView

Innerhalb der `onGenerateRoute` wird das `GameConfigView` Widget aufgerufen und erzeugt.

```
Route<dynamic> onGenerateRoute(RouteSettings settings) {
  switch (settings.name) {
    case Routes.HomeRoute:
      return MaterialPageRoute(builder: (context) => HomeView());
    case Routes.GameRoute:
      return MaterialPageRoute(
          builder: (context) => GameConfigView(
                playerCount: settings.arguments,
              ));
    case Routes.ResultRoundRoute:
    return MaterialPageRoute(builder: (context) => ResultRoundView());
    default:
     return MaterialPageRoute(builder: (context) => FallbackView());
  }
}
```

Die Home View selbst sieht so aus:

```
class GameConfigView extends StatelessWidget {
  final int playerCount;
  final double _roundMin = 1;
  final double _roundMax = 20;

  GameConfigView({this.playerCount, Key key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    final _prov = Provider.of<GameConfigProvider>(context);
    _prov.playerCount = playerCount;

    return Scaffold(
      backgroundColor: Colors.grey[200],
      resizeToAvoidBottomInset: false,
      appBar: AppBar(
        title: Text("Neues Spiel"),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          _prov.startGame();
          Navigator.pushNamed(context, Routes.ResultRoundRoute);
        },
        child: Icon(Icons.arrow_forward_ios),
      ),
      body: Padding(
        padding: EdgeInsets.all(16),
        child: Column(
          mainAxisSize: MainAxisSize.max,
          children: [
            Flexible(
              child: ListView.builder(
                itemCount: this.playerCount,
                itemBuilder: (ctx, i) {
                  return _PlayerSelectWidget(i);
                },
              ),
            ),
            Flexible(
              child: SpinBox(
                value: _prov.roundCount,
                onChanged: (newValue) {
                  _prov.roundCount = newValue;
                },
```

```
                decoration: InputDecoration(
                  labelText: "Rundenanzahl:",
                  border: OutlineInputBorder(
                    borderRadius: BorderRadius.circular(5),
                  ),
                ),
                min: _roundMin,
                max: _roundMax,
              ),
            ),
          ],
        ),
      ),
    );
  }
}

class _PlayerSelectWidget extends StatelessWidget {
  final int number;

  _PlayerSelectWidget(this.number);

  @override
  Widget build(BuildContext context) {
    final _prov = Provider.of<GameConfigProvider>(context);

    return Container(
      padding: EdgeInsets.only(bottom: 16),
      child: Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: [
          TextField(
            onChanged: (s) {
              _prov.setName(s, number);
            },
            decoration: InputDecoration(
              contentPadding: EdgeInsets.only(left: 8),
              labelText: "Spieler ${number + 1}",
              border: OutlineInputBorder(
                borderRadius: BorderRadius.circular(5),
              ),
            ),
          ),
        ],
      ),
    );
```

```
  }
}
```

Die View ist ein `StatelessWidget` und hält somit keinerlei State, da dieser in den Provider ausgelagert wurde. `StatefullWidgets` werden oft in Verbindung mit Animationen verwendet, da diese nicht vom Provider gemanagt werden.

Das Root Widget ist ein sogenanntes Skaffold. Dieses stellt verschiedene typische Elemente, wie eine Top Navigationsbar, einen Floating Action Button oder auch eine Bottom Navigationsbar zur Verfügung.

Mit der Zeile `final _prov = Provider.of<GameConfigProvider>(context);` wird den Widgetbaum nach oben gegangen bis zum `MultiProvider` wo schließlich der `GameConfigProvider` initialisiert wird.

Dieser beinhaltet Variablen, zu den einzelnen Spielern und der Rundenanzahl. Die entsprechenden Felder können mit Settern und Gettern gesetzt werden. Der Provider, welcher durch einen `ChangeNotifier` dargestellt wird, updated die UI jedes Mal wenn die Funktion `notifyListeners()` aufgerufen wird. Woraufhin alle Widget, welche solche Werte nutzen neugebaut werden.

```
class GameConfigProvider extends ChangeNotifier {
  double _roundCount = 1;
  List<String> _names = [];
  GameService gs;
  int _playerCount;

  GameConfigProvider(this.gs);

  set playerCount(int c) {
    _playerCount = c;
  }

  set roundCount(double newValue) {
    _roundCount = newValue;
    notifyListeners();
  }

  double get roundCount => _roundCount;

  void setName(String name, int i) {
    if (_names.length == 0) {
      for (int i = 0; i < _playerCount; i++) {
        _names.add("");
      }
    }

    _names[i] = name;
```

```
  }

  // will use the singleton gameservice to create a game
  // provide all values to gameservice
  void startGame() {
    gs.startGame(_roundCount.toInt(), _names);
  }
}
```

Der Game Config Bildschirm sieht dann so aus:

layout: default title: “08 - Fazit Flutter Umsetzung” nav_order: 8 permalink: /fazit —

Fazit

Flutter ermöglichte einen schnelleren Einstieg als Android mit Jetpack. Durch das einfache Widget System war es möglich einen größeren Teil des MVP 1 umzusetzen.

Native Android Entwicklung bietet allerdings auch gute Vorteile und steht nicht unter dauerhaftem Wandel wie Flutter, bedingt durch das große Alter. Allerdings erschweren fehlende Beispiele und eine geringe Unterstützung der Community das Entwickeln. Sodass nur geraten werden kann mit Kotlin zu entwickeln und kein Java mehr zu nutzen.

Ressourcen

Android

- Android Dokumentation
- Jetpack Dokumentation
- Jetpack Beispiel App in Kotlin
- Jetpack Beispiel App in Java (von Community erstellt)
- Android Basics
- Android Developers YouTube Kanal

Flutter

- Flutter Installation
- Flutter Docs
- Flutter Tutorial zu Routen
- Provider Package
- Get It Service Locator
- Fluttericon Package integriert alle großen Icon Packages
- Flutter Boring Show von den Flutter Entwicklern
- Flutter YouTube Kanal