
Informatikprojekt

# Vorstellung des UI-Frameworks „Flutter“ anhand einer Money-Tracker Anwendung

Hochschule Ravensburg-Weingarten
Fakultät Elektrotechnik und Informatik
Studiengang Angewandte Informatik
Doggenriedstraße
88250 Weingarten

**Simon Förster**
29990

**Luca Ruf**
29916

# Inhaltsverzeichnis

# 1 Projektbeschreibung

## 1.1 Motivation

Bei der nativen Entwicklung von mobilen Anwendungen für Android und iOS ist man meist gezwungen auf unterschiedliche UI-Frameworks und Programmiersprachen zurückzugreifen. Auf der Suche nach einem Framework, welches Cross-Platform Entwicklung unterstützt, wurde man auf das von Google entwickelte Framework Flutter aufmerksam.
Seit dem kürzlichen Release von Flutter 2.0 ist es neben der Entwicklung von mobilen Cross-Platform Anwendungen möglich Web- und Desktopanwendungen mit der selben Codebasis zu erstellen.

## 1.2 Zielsetzung

Ziel dieses Projekts ist die Vorstellung des UI-Frameworks 'Flutter' als Ganzes. Es soll ein Einblick in Entwicklung geben, mit der ohne größere Anpassungen am Code, Anwendungen auf verschiedenen Plattformen lauffähig gemacht werden können. Es soll der Entwicklungsprozess mit Flutter anschaulich gemacht werden, um das Potential des jungen Frameworks zu testen.
Bei der Recherche nach Frameworks, die Cross-Platform Entwicklung unterstützen, wurden verschiedene Technologien entdeckt. Diese Technologien unterscheiden sich in gewissen Bereichen, die im Laufe dieser Arbeit verglichen und bewertet werden sollen.
Im Verlauf des Projekts wird zur Veranschaulichung der Möglichkeiten eine Anwendung erstellt, welche sowohl als mobile Anwendung als auch als Webanwendung genutzt werden kann.
Die Anwendung soll eine Art 'Expense-Tracker' für zwei oder mehr Personen darstellen, die helfen soll, gemeinsame Ausgaben zu verwalten. Ähnliche Apps auf dem Markt sind Splitwise oder Tricount. Auf den Funktionsumfang der Anwendung wird in Kapitel 4 genauer eingegangen.
Es soll gezeigt werden, wie die Konvertierung des Codes zu einer Webanwendung funktioniert und welche Möglichkeiten und Probleme sich dadurch ergeben.

# 2 Grundlagen

## 2.1 Flutter

Flutter ist ein von Google entwickeltes Open-Source UI-Entwicklungs-Kit und wurde erstmals 2017 veröffentlicht. Flutter ist damit noch sehr jung und bietet die Möglichkeit Cross-Platform mobile Apps auf einer einzigen Codebasis zu entwickeln. Die genutzte Programmiersprache hierbei ist Dart. Zu den unterstützten Plattformen zählen Android, iOS und seit dem Release von Flutter 2.0 im April 2021 auch Windows, Linux, macOS, Google Fuchsia und das Web. [1]

## 2.2 Dart

Dart ist eine klassenbasierte, also objektorientierte, typfeste Programmiersprache aus der C-Familie. Dart ist ebenfalls von Google entwickelt und optimiert für UI-Entwicklung. Die Programmiersprache verspricht native Performance auf verschiedensten Plattformen. Dazu kommt ein JIT-Compiler zum Einsatz, der auch Hot-Reload ermöglicht. Für 'FlutterWeb' muss der Dartcode in JavaScript übersetzt werden, da im Web momentan keine Kompilate unterstützt werden. [2]

## 2.3 Cross-Platform Entwicklung

'Cross-Platform' ist ein allgemeiner Begriff der beschreibt, dass eine Technologie über mehrere Plattformen hinweg lauffähig ist. Im Kontext dieser Arbeit sprechen wir von Cross-Platform Entwicklung im Bezug auf mobiles Cross-Platform (iOS, Android) und Cross-Platform im Zusammenhang mit Flutter 2.0 (iOS, Android, Web). Seit dem Release von Flutter 2.0 ist es auch möglich DesktopApps für verschiedene Systeme zu generieren, dies wird aber in dieser Arbeit nicht behandelt.
Möchte man eine native App für iOS, Android und das Web entwickeln, muss man auf drei unterschiedliche Technologien zurückgreifen. Swift für iOS, Kotlin oder Java für Android und JavaScript (HTML/CSS) für Webapplikationen.
Da dies zeitaufwändig und kostspielig ist versuchen Cross-Platform-Frameworks die Möglichkeit zu bieten aus einer Codebasis, Apps für mehrere Plattformen mit vergleichbarer Performance zu generieren. Hierbei ist die Art der Plattform ein entscheidender Faktor. Bildschirmgröße, verfügbare Hardware (Kamera, Geolocator, ...) und die Art des Inputs (Touch, Maus und Tastatur) sind einige Unterschiede von Plattformen, die bei der Entwicklung bedacht werden müssen. [3]
Trotz dieser Hürden verspricht Cross-Plattform Entwicklung Einsparungen von Zeit und Geld und eine größere Reichweite des entwickelten Produkts über verschiedenen Plattformen.
Weitere Beispiele für Cross-Platform-Frameworks sind Ionic, React Native, Unity und Xamarin.

### 2.3.1 Cross-Platform vs Native Development

Neben der Cross-Platform Entwicklung gibt es natürlich auch die native Entwicklung. Hier wird die Anwendung speziell für die Zielplattform realisiert und angepasst. Bei der Entscheidung für Unternehmen oder Entwickler welche Methode sie für ihre Entwicklung und Ansprüche nutzen, sollte man beide Varianten gegenüber stellen. Beide Methoden haben ihre Vor- und Nachteile und man kann damit keinen pauschalen Sieger festlegen. Es gibt einige Punkte, die bei einer App Entwicklung in Betracht gezogen werden können.

| Faktor | Native Entwicklung | Cross-Platform Entwicklung |
|---|---|---|
| Kosten | Hohe Entwicklungskosten | Niedrige Entwicklungskosten |
| Codebasis | Code funktioniert nur auf einer Plattform | Eine Codebasis für mehrere Plattformen |
| Performance | Höchste native Performance | Hohe Performance fast wie nativ möglich |
| Gerätefunktionen | Alle Gerätefunktionen möglich | Teilweise native Möglichkeiten |
| Nutzer Erreichbarkeit | Anwendung auf Nutzer einer Plattform begrenzt | Nutzer unterschiedlicher Plattformen |
| Entwicklungsdauer | Hohe Entwicklungsdauer | Schnelle Entwicklung |

Abbildung 1: Cross-Platform vs. native Entwicklung [4]

Aus Abbildung 1 lässt sich entnehmen, dass die Hauptvorteile einer Cross-Platform Entwicklung sich aus der Flexibilität, Entwicklungsaufwand und den damit verbundenen Kosten herauskristallisieren. Durch die Implementierung einer Codebasis ergibt sich eine Art Kettenreaktion, welche der Cross-Platform Entwicklung zugute spricht. Auf der anderen Seite spricht die Qualität für die native Entwicklung. Ist das Ziel eine perfekt angepasste qualitative Anwendung, sollte diese Methode gewählt werden. Dennoch sollten davor eigene Anforderungen evaluiert werden, da jedes Projekt individuell ist.

# 3 Flutter als UI-Framework

## 3.1 Widgets

Die Kernkomponente von Flutter sind sogenannte Widgetes. UI-, Styling-, Layout- und Animations-Widgets sind Klassen, die von 'StatelessWidget' oder 'StatefulWidget' erben. Durch die Kombination und Komposition dieser Widgets wird die gesamte Benutzeroberfläche gebaut. Dadurch entsteht eine Baumstruktur von Widgets, die ihre Wurzel im 'Root-Widget' hat. Die Flutter SDK stellt sehr viele Widgets fertig zur Verfügung. Widgets sind vergleichbar mit (reusable) Components aus anderen Frameworks.

```
class ExampleWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Container(
      child: Center(
        child: Text("This is an Example Widget!"),
      ),
    );
  }
}
```

Listing 1: Beispiel Widget [5]

In Listing 1 ist ein einfaches Beispiel-Widget zu sehen. Es erbt von 'StatelessWidget', da es keine sich ändernde Variablen hält, die für die Darstellung relevant sind (State). Die 'build()' Methode, die von der Basisklasse überschrieben wird, liefert ein Widget zurück. Vor dem 'return' Statement, kann Logik ausgeführt werden, um Daten zu laden oder zu berechnen. In diesem Beispiel liefert die build-Methode ein geschachtelten Container zurück. In diesem Container befindet sich ein Center-Widget, dass seine child-Komponente mittig im verfügbaren Raum auslegt. Die letzte Ebene des Widgets ist ein Text-Widget, dem ein fester String übergeben wurde.

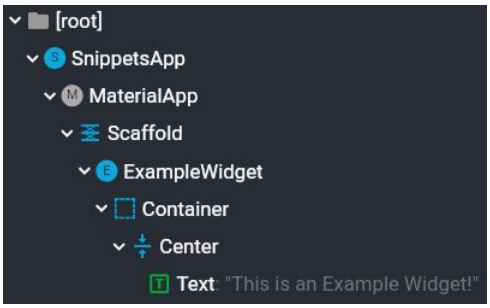



Abbildung 2: WidgetTree von ExampleWidget [5]

In Abbildung 2 ist der zugehörige WidgetTree des ExampleWidgets zu sehen. In diesem Beispiel lebt das ExampleWidget unterhalb eines Scaffolds, der die Basiskomponente eines Screens darstellt. Darüber findet man eine MaterialApp, in der globale Themes nach dem MaterialDesign eingestellt werden können und die SnippetsApp, die nach dem Projekt benannt ist und von der 'main()' Methode instantiiert wird.
Über Konstruktoren können Widgetklassen Werte übergeben werden, die deren Verhalten beeinflussen. In Flutter wird sehr viel mit optionalen und benannten Parametern in den Konstruktoren gearbeitet, somit ist der Sourcecode übersichtlich und leicht zu verstehen. Diese benannten Parameter sind in Listing 1 bei dem Container- und Center-Widget als 'child' zu sehen. Im Prinzip ist ein WidgetTree also nichts anderes, als verschachtelte Konstruktoren von Widgets. Durch dieses Grundprinzip ist es sehr einfach eigne modulare Widgets zu bauen, die dann projektübergreifend genutzt werden können.

```
[...]
// Aufruf des Konstruktors
return ConstructorExampleWidget(
    amount: 5,
    fruitName: "oranges",
);
[...]

class ConstructorExampleWidget extends StatelessWidget {
  final int amount;
  final String fruitName;

  //Konstruktor mit Fallback-Werten
  const ConstructorExampleWidget({this.amount = 0, this.fruitName = ""});

  @override
  Widget build(BuildContext context) {
    return Container(
      child: Center(
        child: Text("I have $amount $fruitName."),
      ),
    );
  }
}
```

Listing 2: Konstruktor Demonstration [5]

In Listing 2 ist eine Widgetklasse (ConstructorExampleWidget) und eine aufrufender Stelle zu sehen. Das Widget hat zwei Membervariablen, die über den Konstruktor übergeben werden können. Wird kein Wert für eine der Variablen übergeben, so erhalten diese einen festen Wert (amount = 0; fruitName = "";). Dadurch sind die Parameter optional, aber falls diese nicht übergeben werden trotzdem nicht 'null'. Falls notwendig, gibt es auch 'nullable Types', die mit einem '?' hinter dem Typnamen angegeben werden (int?, String?, ...).
An der aufrufenden Stelle wird anschaulich, wie die Parameter über die Namen der Mem-

bervariablen übergeben werden. Die Reihenfolge spielt hierbei keine Rolle. Das Resultat ist für den Entwickler sehr gut leserlich und verständlich, ohne dass er den Aufbau des 'ConstructorExampleWidgets' genau kennt.
Ein Konstruktor im 'klassischen' Sinn, mit positionellen Parametern, kann ebenfalls genutzt werden. Das Text-Widget hat beispielsweise ein positionelles Argument, den String, den das Text-Widget darstellen soll. Auch bei dieser Herangehensweise können weitere optionale, benannte Parameter benutzt werden.

Die Flutter SDK stellt viele Widgets zur Verfügung, die für die meisten Applikationen nur korrekt zusammengefügt werden müssen. Es besteht trotzdem die Möglichkeit aus einzelnen 'low-level' Widgets, wie SizedBox, Transform oder GestureDetector sich eigene Widgets nach belieben zu bauen und zu personalisieren. Die Mächtigkeit der Widgets in der Flutter SDK reichen von Containern, SizedBoxes, Columns, Rows bis hin zu AppBars, Drawers, ButtomNavigationBars, FloatingActionButtons, ListViews, GridViews,...

Ein Überblick über die Widgets stellt Google auf 'https://flutter.dev/docs/development/ui/widgets' zur Verfügung. Die Eigenheiten und Unterschiede von 'Stateful-' und 'Stateless-' Widgets werden im Kapitel 3.5 State-Management näher beleuchtet.

## 3.2 Layout

Wie alles in Flutter wird das Layout von UI-Komponenten von speziellen Widgets oder entsprechende Eigenschaften auf den UI-Komponenten übernommen. In Listing 1 wurde bereits das Center-Widget vorgestellt, dessen einzige Aufgabe es ist seine child-Komponente mittig im verfügbaren Raum zu plazieren. Alternativ kann durch die Verwendung des Align-Widgets ein Child-Widget, je nach Wert des 'alignment' Properties ebenso im verfügbaren Raum zentriert, links-unten, rechts-oben, usw. plaziert werden.

```
return Align(
  alignment: Alignment.center,
  child: Text("This Text is centered!"),
);
```

Listing 3: Alternative zu Center [5]

Row- und Column-Widgets sind elementar wichtig für das Layout von Komponenten. Im Gegensatz zu den meisten Widgets kann ihnen eine Liste von Widgets als 'children' übergeben werden. Diese children werden dann anhand von übergebenen Layouteigenschaften (MainAxisAlignment, CrossAxisalignment, VerticalDirection, ...) plaziert.

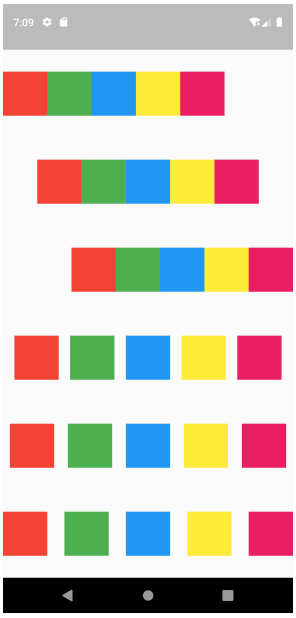


Abbildung 3: Verschiedene Werte des 'MainAxisAlignment' Parameters [5]

In Abbildung 3 sind sechs verschiedene Row-Widgets zu sehen, die alle die gleichen children-Komponenten haben. Der Unterschied dieser Row-Widgets ist einzig der Wert der 'MainAxisAlignment', der bestimmt mit welchem Abstand die children-Widgets positioniert werden. Bei den ersten drei Row-Widgets von oben ist der Wert der MainAxisAlignment 'start', 'center' und 'end'. Hierbei werden die children-Widgets ohne Abstand linksbündig, zentriert oder rechtsbündig ausgelegt. Das vierte Row-Widget trägt die Eigenschaft 'spaceEvenly'. Hierbei liegt der gleiche Abstand zwischen dem Rand und den äußersten Widgets, als auch zwischen den einzelnen Widgets. Die Row Nummer 5 wurde mit der Einstellung 'spaceAround' ausgelegt. 'spaceAround' funktioniert gleich wie 'spaceEvenly' nur, dass der Abstand von den äußersten Widgets zum Rand halb so groß ist, wie der zwischen den inneren Widgets. Die letzte Row ist mit dem Wert 'spaceBetween' ausgestattet. Dies bewirkt, dass der Raum zwischen den Widgets maximiert wird und folglich kein Rand zwischen den äußersten Widgets und der Bildschirmrand besteht. [6]

Auch für Layoutansprüche wie **Padding** stellt die Flutter SDK eigene Widgets zur Verfügung, die spezialisiert für diese Aufgabe sind. Zum einen gibt es das Padding-Widget, dem über die padding Eigenschaft ein EdgeInsets-Widget übergeben werden kann und dementsprechend seiner child-Komponente Padding verleiht. Das EdgeInsets-Widgets verfügt über mehrere benannte Konstruktoren, wodurch je nach Nutzen das gewünschte Verhalten hervorgerufen wird.
Alternativ besitzt auch das Container-Widget eine Eigenschaft, über die direkt ein EdgeInsets-Widget als Padding übergeben werden kann. Dies zeigt, dass viele Widgets der Flutter

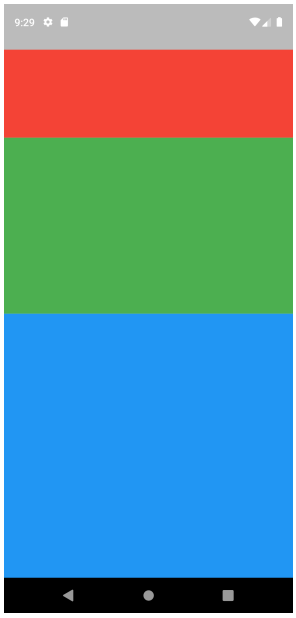
SDK austauschbar sind. Nichtsdestotrotz sind diese spezialisierten Widgets sehr handlich und intuitiv im Gebrauch.

```
//Beispiel Padding-Widget
return Padding(
  padding: EdgeInsets.only(left: 8, top: 8),  //benannter Konstruktor 'only'
  child: Text("This Text has Padding"),
);

//Beispiel Padding im Container-Widget
return Container(
  padding: EdgeInsets.all(8),  //benannter Konstruktor 'all'
  child: Text("This Text has Padding"),
);
```

Listing 4: Beispiele für den Einsatz von Padding [5]

Um zur Laufzeit des Programms die tatsächliche **Größe** der einzelnen UI-Komponenten bestimmen zu können, ermittelt der Flutter-Compiler den verfügbaren Raum und platziert als erstes Widgets mit fester Größe. Unter den Widgets mit variabler Größe wird der restliche Raum aufgeteilt. Hierbei verhalten sich Widgets unterschiedlich. Beim Listing 1 wurde das Container-Widget vorgestellt. Ein leerer Container nimmt sich den maximalen Raum, der ihm zur Verfügung steht. Bei Row- und Column-Widgets kann angegeben werden, ob diese sich über den ganzen verfügbaren Raum erstrecken, oder nur die Länge/Breite des größten Widgets in ihren children annehmen sollen.
Um dieses Verhalten beeinflussen zu können stehen dem Entwickler mehrere Widgets zur Verfügung. Mit dem Expanded-Widget kann ein Widget verpackt werden, welches dadurch immer den maximal verfügbaren Raum beansprucht. Alternativ können Widgets mit einem Flexible-Widget ausgestattet werden. Das Flexible-Widget kann ein FlexFit von 'tight' oder 'loose' übergeben werden oder ein 'flex' Wert. Der Flex-Wert, dem ein Integer übergeben werden kann bestimmt die Größe der Widgets auf der Hauptachse. Hierbei werden die Flex-Werte aller Widgets ermittelt (Standard ist flex = 1) und den verfügbaren Raum durch die Summe der einzelnen Flex-Werte geteilt. Jedes Widget erhält nun so viele Bruchstücke, wie in seinem Flex-Wert angegeben wurden.

In Abbildung 4 befinden sich drei Container, die mit einem Flexible-Widget verpackt wurden, in einem Column-Widget. Durch die Flex-Werte werden den Containern beim Auslegen der Widgets unterschiedliche Größen zugeteilt. Der verfügbare Raum der Hauptachse ist die ganze Höhe des Screens. Die Summe der Flex-Wert der Widgets im Column beträgt 6 (1 + 2 + 3). Nach der oben erwähnten Rechnung erhält nun jeder Container die Anzahl der Bruchstücke, die in seinem Flex-Wert angeben wurde. Dementsprechend erhält der rote Container 1/6 der Bildschirmhöhe, der grüne 1/3 (2/6) und der blaue Container 1/2 (3/6). Durch dieses System werden Größen **responsive** verändert, das heißt bei Veränderung der Bildschirmhöhe, ändert sich auch die Größe der Widgets.



```
return Column(
    children: [
        Flexible(
          flex: 1,
          child: Container(color: Colors.red),
        ),
        Flexible(
          flex: 2,
          child: Container(color: Colors.green),
        ),
        Flexible(
          flex: 3,
          child: Container(color: Colors.blue),
        ),
    ],
);
```

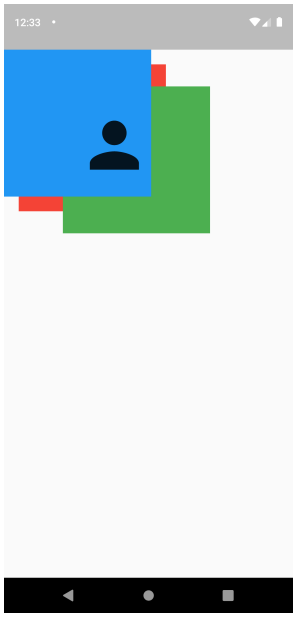
Abbildung 4: Beispiel des Flexible-Widgets [5]

Bei andere Komponenten wie dem Text-Widget muss eine feste Größe über die Font-Size gegeben sein, um diese im verfügbaren Raum zu plazieren. Möchte man Widgets mit variabler Größe zwingen eine feste Größe anzunehemen, kann bei manchen Widgets der width/height Parameter auf eine feste Größe setzt werden. Alternativ gibt es die 'Sized-Box' ein spezialisiertes Widget dafür, seiner child-Komponente eine feste Größe zu geben.

Sollten die Layoutansprüche es erfordern, dass sich Widgets überlappen, oder mehrere Widgets übereinander liegen, so lässt sich dies nicht über Row- oder Column-Widgets realisieren. Speziell für diesen Anspruch stellt die Flutter SDK das Stack-Widget zur Verfügung. Das Stack-Widget ermöglicht es Widgets zu 'stapeln'.

In Abbildung 5 ist die Funktionsweise des Stacks anhand eines einfachen Beispiels zu sehen. In Kombination mit dem Positioned-Widget lassen sich mit diesem Widget jede Art von komplexer UI bauen. Besonders nützlich ist ein Stack bei der Realisierung von Bannern, um Widgets im Hintergrund darzustellen oder für komplexe Animationen, die in Kapitel 3.4 behandelt werden.

Vor allem im Cross-Platform Development sind die Themen **Responsiveness** und **Adaptiveness** sehr wichtig.
Eine App, welche als **Responsive** gilt, beachtet bei der Layoutgestaltung die Größe des Bildschirms. Das heißt, bei einer Veränderung der Bildschirmgröße, egal ob manuell oder gerätebedingt, verändert sich das Layout. Dies kann beispielsweise eine andere Anordnung




```
return Stack(
    children: [
        Positioned(
            left: 20,
            top: 20,
            child: MyColoredBox(color: Colors.red),
        ),
        Positioned(
            left: 80,
            top: 50,
            child: MyColoredBox(color: Colors.green),
        ),
        MyColoredBox(color: Colors.blue),
        Positioned(
            left: 100,
            top: 80,
            child: Icon(Icons.person, size: 100),
        ),
    ],
);
```

Abbildung 5: Beispiel des Stack-Widgets [5]

von Tabelleninhalten oder einfach die Veränderung der Breite eines Widgets sein.[7]
In Flutter ist Responsive-Design mit den zuvor gezeigten Mechanismen sehr intuitiv zu implementieren. Widgets nehmen bei der Veränderung der Bildschirmgröße automatisch neue Größen an. Zusätzlich ist es möglich die Bildschirmgröße des aktuellen Geräts zur Laufzeit abzufragen und auf dieser Basis die Benutzeroberfläche verändert darzustellen. Mit dieser Methode kann zum Beispiel bei Mobilgeräten herausgefunden werden, ob es sich um ein Smartphone oder Tablet handelt. Je nachdem können die Ansprüche an die Benutzeroberfläche angepasst werden, um ein benutzerfreundliches UI zur Verfügung zu stellen.

```
Widget build(BuildContext context) {
    Size size = MediaQuery.of(context).size;

    return Container(
      child: size.width < 850
          ? Text("This is a smartphone")
          : Text("This is a tablet"),
    );
  }
```

Listing 5: Responsive Design [5]

Der Aufruf von MediaQuery.of(context) in Listing 5 gibt Zugang zu lokalen Kontextvariablen, darunter auch die Größe des Displays, auf dem die App ausgeführt wird. Aufgrund dieser Größe kann nun das entsprechende UI zurückgeliefert werden. An diesem Beispiel

ist auch die Verwendung von konditioneller Logik im WidgetTree dargestellt. Liefert die Abfrage 'size.width < 850' den Wert 'true' zurück, so wird das obere Text-Widget zurück gegeben und in allen anderen Fällen das untere Text-Widget.

Die **Adaptivität** einer App beschreibt wiederum die Fähigkeit, sich an unterschiedliche Geräte anzupassen. So ist es bei einer WebApp wichtig, die Maus und Tastatur zu berücksichtigen. Bei einer mobilen App erfolgt die Eingabe über den Touchscreen. Auch die Auswahl an Widgets spielt hier eine Rolle, eine AppBar in einer mobilen Anwendung ist durchaus sinnvoll, bei einer WebApp allerdings nicht zwingend notwendig. Ebenso, ob Text markierbar (Web- und Desktop App) sein sollte oder nicht (mobile App). [7]
Analog zu Listing 5 ist es möglich im Fluttercode zur Laufzeit zu prüfen, auf welcher Art von Gerät die App ausgeführt wird.

```
if (Platform.isAndroid || Platform.isIOS) {
    return Text("This is a mobil device");
}
if (Platform.isWindows || Platform.isLinux || Platform.isMacOS) {
    return Text("This is a desktop device");
}
if (kIsWeb) {
    return Text("This is viewed in the web");
}
```

Listing 6: Adaptive Design [5]

In Listing 6 ist zu sehen, dass nicht nur nach der Plattformart (Mobil, Web, Desktop) sondern auf Basis des Betriebsystems verschiedene Benutzeroberflächen angeboten werden können. Mit dieser Möglichkeit all diese Plattformen bedienen zu können kommt der Nachteil, dass im Code der Benutzeroberfläche sehr viel if-else-Statements eingebaut werden müssen, um die App adaptiv für alle Oberflächen zu gestalten.

## 3.3 Styling

Die Gestaltung einer Flutter-App folgt den Richtlinien des Material-Designs. Das Material-Design wurde von Google entwickelt und liefert Prinzipien, Komponenten, Design- und Styling-Vorgaben zur einheitlichen Erscheinung von Web und mobilen Applikationen. [8]
Wie in Abbildung 2 bereits gezeigt wurde, ist das erste Widget unterhalb der Wurzel des Projekts eine 'MaterialApp'. Dieses Widget bietet die Möglichkeit Werte zu übergeben, die das Aussehen und Verhalten der gesamten Applikation verändern. Der wohl bedeutendste Parameter der MaterialApp ist das 'theme'. Dieser Eigenschaft kann ein ThemeData-Widget übergeben werden, in dem alle Styling-relevanten Eigenschaften gesetzt werden können. Zu diesen Eigenschaften gehören unter anderem 'primarySwatch' (Farbpalette für Buttons, DatePicker, TimePicker, ...), 'brightness' (Light-/Dark Theme), 'primaryColor' (Hauptfarbe der App) und 'textTheme' (Schriftart und Größe der Textkomponenten).

Die einzelnen UI-Komponenten, die sich tiefer im WidgetTree befinden nehmen dann automatisch die nach dem MaterialDesign definierten Farben aus dem Theme der MaterialApp an. Sollten individuelle Wünsche an das Styling einzelner UI-Komponenten vorliegen, kann der ThemeData auch spezielle BottomNavigationBarThemeData, FloatingActionButtonThemeData, ... übergeben werden.
Das stylen von Widgets direkt im WidgetTree ist möglich, wie zum Beispiel Abbildung 4 mit dem color-Parameter eines Containers zeigt. Als Best-Practice zählt allerdings dies zu vermeiden und stattdessen den zentralen Ort ThemeData zu nutzen.
Die Flutter SDK stellt alle Komponenten die im MaterialDesign beschrieben werden zur Verfügung. Dies ermöglicht es, dass die vorgegebenen Widgets automatisch die Styles der ThemeData annehmen und somit ohne großes Wissen von Design schöne Oberflächen entstehen.


```
return MaterialApp(
    theme: ThemeData(
        brightness: Brightness.light,
        primarySwatch: Colors.blue,
    ),
    [...]
);
```

Abbildung 6: Demonstration des Material-Design Theme [5]

In Abbildung 6 ist zu sehen, wie nur mit den Einstellungen 'brightness' und 'primarySwatch' die gezeigten UI-Komponenten automatisch die richtigen Farben annehmen, ohne diese explizit zuweisen zu müssen.
Über diese ThemeData hinweg stellt die Flutter SDK viele weitere Widgets zur Verfügung, mit denen die Form, Farbe, Rand und andere Dekorationen verändert werden können. Natürlich können diese Widgets auch schon direkt im ThemeData verwendet werden, um die UI-Komponenten über die Farbe hinweg zu personalisieren.

## 3.4 Animationen

Animationen sind in Benutzeroberflächen wichtig, um die Aktionen des Nutzers zu visualisieren. Verschwindet ein Element nach dem Löschen durch den Nutzer abrupt aus einer Liste, so ist der Nutzer eventuell verwirrt, weil er seine Aktion nicht nachvollziehen kann. Bewegt sich das Element in einer Animation von der Bildfläche und ist dabei noch ein Mülleimer-Icon zu sehen, so kann der Nutzer nachvollziehen, was das Resultat seiner Aktion war.
Bei der Entwicklung einer App mit Flutter erhält man einige Animation, ohne diese explizit zu implementieren. Das ein- und ausblenden von neuen Screens, DropdownMenus, Drawern, Date-/TimePickern wird mit einer passenden Animation begleitet, genauso wie das umblättern in einer PageView.
Für explizite Animationen stellt die Flutter SDK wiederum spezialisierte Widgets zur Verfügung. Zum einen besteht die Möglichkeit anstatt einem regulären Widget ein animiertes Widget zu benutzen. AnimatedContainer, AnimatedPositioned oder AnimatedAlign bewirken das Selbe wie ihre nicht animierten Gegenstücke, nur dass diese bei Änderungen des Layouts zwischen den Änderungen animieren. Animiert werden kann je nach Art des Widgets die Position, die Größe, die Farbe, die Form, die Rotation und vieles mehr.
Für individuelle Ansprüche an Animationen und der gleichzeitigen Animation mehrer Widgets kann ein AnimationController-Widget verwendet werden. Dieser Controller kann den zu animierenden Widgets übergeben werden und von ihm aus können alle für die Animation relevanten Parameter wie Dauer, Geschwindigkeit, Beginn und Ende gesteuert werden.

## 3.5 State-Management

In allen bisherigen Beispielen wurden **StatelessWidgets** gezeigt. StatelessWidgets sind Widgets deren Darstellungen nicht von Variablen des Widgets abhängig sind. Dadurch, dass ihre Darstellung nicht variabel ist, müssen diese Widgets zur Laufzeit nicht neu erzeugt werden. StatelessWidgets sollen bevorzugt genutzt werden, solange sich das Verhalten eines Widgets als 'stateless' implementieren lässt. Ist dies nicht möglich, müssen StatefulWidgets verwendet werden.

**StatefulWidgets** erzeugen beim Aufruf ihres Konstruktors eine Instanz ihrer eigenen State-Klasse. Diese State-Klasse hält Membervariablen, von denen die Darstellung des Widgets abhängt, außerdem befindet sich wie auch im StatelessWidget eine build-Methode. Die Membervariablen der Stateklasse können zur Laufzeit geändert werden was bewirkt, dass die alte State-Instanz verworfen wird und eine neue State-Instanz mit geänderten Membervariablen erzeugt wird. Dies wird erreicht, indem man die setState()-Methode in der State-Klasse aufruft und darin die Membervariablen neu setzt. Durch dieses Vorgehen muss nicht das gesamte Widget neu erzeugt werden, sondern ausschließlich ein neuer

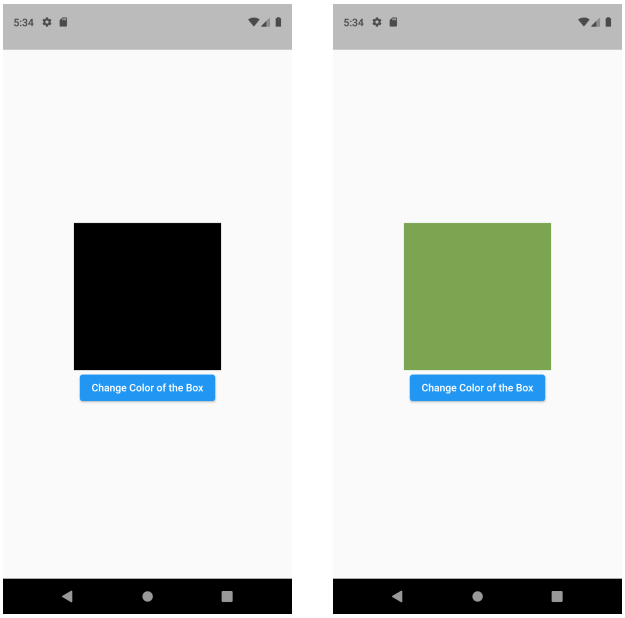
State. Variablen, die ursprünglich beim Erzeugen des Widgets gesetzt wurden bleiben bestehen, da diese im Widget gehalten werden und nicht in der State-Klasse.

```
class StatefulWidgetExample extends StatefulWidget {
  @override
  _StatefulWidgetExampleState createState() => _StatefulWidgetExampleState();
}

class _StatefulWidgetExampleState extends State<StatefulWidgetExample> {
  Color boxColor = Colors.black;  //Original State

  @override
  Widget build(BuildContext context) {
    return Column(
        children: [
            MyColoredBox(color: boxColor),
            ElevatedButton(
                child: Text("Change Color of the Box"),
                onPressed: () {
                    setState(() => boxColor = randomColor()); //State Change
                },
            )
    ]));
  }
}
```

Listing 7: Beispiel StatefulWidget [5]

In Listing 7 ist ein Beispiel eines StatefulWidgets zu sehen. Der konkrete 'State' in diesem Widget ist die Variable 'boxColor'. Wird das Widget instantiiert, so erzeugt es sich eine Instanz seiner State-Klasse mit dem der Membervariable 'boxColor = black'. Die Variable boxColor wird dem child-Widget 'MyColoredBox' als Parameter übergeben. Ändert sich der Wert der Variable boxColor, so soll ein neuer State erzeugt werden mit dem neuen Wert für boxColor. Der neue Wert soll auch dem MyColoredBox-Widget übergeben werden.
Beim Klick auf den Button 'Change Color of the Box' passiert genau dies. Über die Methode setState() wird der Wert der Variable boxColor auf eine zufällige Farbe gesetzt, was das Widget zwingt einen neuen State mit der neuen Farbe zu erzeugen. In Abbildung 3.5 ist in (a) das StatefulWidget vor dem Klicken des Buttons zu sehen und in (b) das selbe Widget mit geändertem State.

StatefulWidgets sind vor allem dann notwendig, wenn sich das UI aufgrund von Nutzereingaben oder Aktionen ändert. Beim Ausfüllen von Formularen, Anzeigen von sich dynamisch ändernden Daten oder beim Ausführen von Animationen sind StatefulWidgets unumgänglich. Der Umgang mit State-Variablen im gesamten WidgetTree ist eine der anspruchsvolleren Aufgaben bei der Entwicklung einer Flutter-App.




(a) Originaler State [5] (b) Geänderter State [5]

Abbildung 7: Beispiel StatefulWidget

## 3.6 Plattformen

Im Kapitel 3.2 wurde bereits gezeigt, wie auf Grundlage der Bildschirmgröße und Platformchecking Fluttercode entstehen kann, der sich responsiv und adaptiv verhält. Plattform und Bildschirmgröße ändern den Anspruch an die Benutzeroberfläche in verschiedenen Aspekten. In dieser Arbeit wurde neben der mobile App auch das Erstellen einer WebApp untersucht, Desktop Applikationen wurden vernachlässigt.

Auf Smartphones ist die Bildschirmgröße sehr beschränkt, meistens ist auf einem Screen nur eine Art von Information/Aktion zu finden und Navigationselemente müssen einfach zu erreichen sein. BottomNavigationBars eignen sich daher auf Smartphones besonders gut. Bereits auf Tablets wirken diese bereits unnötig breit und andere Navigationsmöglichkeiten wie ein Drawer sind besser geeignet. Auf einem Tablet können aufgrund der Bildschirmgröße meist mehrere Screens einer Smartphone-App angezeigt werden. WebApps wiederum müssen mit den Funktionen und Eigenheiten von Browsern umgehen können. Zusätzlich müssen WebApps auch responsiv gestalten werden, da diese sowohl mit einem Mobilgerät als auch von einem Desktop-Computer benutzt werden können.
All diese Aspekte beeinflussen die Anforderungen an ein UI. Da bei Cross-Platform Development all diese Anforderungen in einer Codebasis untergebracht werden müssen, bedarf es sehr viel if-else Logik im UI-Aufbau. Alternativ können auch eigene adaptive Widgets geschrieben werden, die das Verhalten eines Flutter SDK Widgets bieten und intern Platfrom-checking betreiben.

```
class AdaptiveTextWidget extends StatelessWidget {
  final String text;
  const AdaptiveTextWidget(this.text);

  @override
  Widget build(BuildContext context) {
    if (kIsWeb) {
      return SelectableText(text);  //In the Web, Text should be selectable
    }
    return Text(text);
  }
}
```

Listing 8: Adaptives Text-Widget [5]

In Listing 8 ist das oben beschriebene Vorgehen mit einem TextWidget realisiert. Bei der Entwicklung des UIs kann nun überall anstatt dem 'normalen' Text-Widget das AdaptiveTextWidget genutzt werden, welches automatisch bei einer WebApp ein SelectableTextWidget zurück gibt. Dieses Vorgehen hat allerdings nicht nur Vorteile. Will der Entwickler zusätzlich eine TextAlign Eigenschaft übergeben, so muss das AdaptiveTextWidget wieder angepasst werden, um ein TextAlignWidget entgegen zu nehmen.

Beim builden einer Flutter WebApp ensteht eine Single-Page-Application. Der Dart-Code wird in JavaScript übersetzt. Die WebApp kann auf zwei verschiedene Arten gebuildet werden. Zum einen kann sie als HTML über die DOM Canvas API gebuildet werden, um den Flutter-Widgets das identische Layout und Aussehen zu geben wie auf der mobilen App. Die Alternative hierbei ist über CanvasKit, welches mit WebAssembly und WebGL die UI-Komponenten lauffähig für Webbrowser macht.
Es wird von Flutter empfohlen keine Static-Content-Pages in Flutter zu entwickeln, da in der Flutter-SPA Informationen fehlen, die für Suchmaschinen relevant sind. Außerdem muss geprüft werden, ob alle Plugins, die in der App verwendet werden kompatibel mit dem Web sind. Die resulierende FlutterWeb-App bietet die Möglichkeit als Progressive-Web-App installiert zu werden und bieten darin ihren vollen Funktionsumfang. [9]

## 3.7 Null-Safety

Seit dem Release von Flutter 2.0 wird 'sound null-safety' unterstützt. Unter 'sound null-safety' versteht man, dass alle Typen standartmäßig nicht den Wert 'null' annehmen können. Sogenannte 'nullable-Types' können explizit mit einem '?' hinter dem Typ deklariert werden (int?, String?, MyWidget?, ...). Außerdem erzeugt der Compiler Fehlermeldugen, wenn auf eine 'nullable'-Variable zugegriffen wird. Mit diesem Vorgehen soll der Entwickler sich damit auseinandersetzen, welche Variable den Wert 'null' annehmen dürfen und welche nicht. Außerdem ist es wegen den Fehlermeldungen des Compilers not-

wendig sicheres 'null-checking' zu implementieren. Die Programmiersprache Dart liefert zur Vermeidung von null-Werten einige nützliche syntaktische Konstrukte, wie 'Fallback-Werte' oder das Prüfen auf null beim dereferenzieren (myObject?.myProperty). [10]

```
class NullSafetyExample extends StatelessWidget {
  final String? fruitName;  //can be null

  const NullSafetyExample({this.fruitName}); //optional Parameter, without Fallback

  @override
  Widget build(BuildContext context) {
    return Container(
      child: Center(
        child: fruitName != null
            ? Text("I like " + fruitName! + ".")
            : Text("Something went wrong."),
      ),
    );
  }
}
```

Listing 9: Beispiel von Null-Safety [5]

In Listing 9 ist ein Widget zu sehen, welches einen nullable String als optionalen Konstruktorparameter entgegen nimmt. Wird der Konstruktor aufgerufen ohne, dass für 'fruitName' ein Wert übergeben wird, so kommt es, ohne null-checking zu einem Laufzeitfehler, weil auf eine null-Variable zugegriffen wird. Im Aufbau des WidgetTrees wird deshalb dieses null-checking durchgeführt und abhängig davon eines der beiden Text-Widgets zurückgegeben. Das '!' hinter fruitName bedeutet, dass die Variable zuvor auf null geprüft wurde, garantiert nicht den Wert null hält und somit sicher referenziert werden kann.
Durch Flutters sound null-safety entsteht qualitativ hochwertigerer Code, bei dem, falls die Mechanismen des null-Handlings richtig verwendet wurden, es zu keinen Null-Referenz-Exceptions kommen sollte.

# 4 Einblick in die Anwendung 'Coin'

## 4.1 Überblick

Wie in Kapitel 1 bereits erwähnt, soll die Anwendung eine Art 'Expense-Tracker' darstellen. Zu den Funktionen gehört das Erfassen, Kategorisieren und Verwalten von Ausgaben von zwei oder mehr Personen. Die Zielgruppen hierbei sind beispielsweise Paare, Freunde, Reisende oder Mitbewohner.
Nutzer sollen in der Anwendung eigene Kategorien erstellen können, um diesen dann dementsprechende Ausgaben zuzuordnen. Somit bietet die Anwendung unterschiedliche Möglichkeiten der Personalisierung. Den Nutzern werden diese Daten dann in Form von Diagrammen dargestellt, um Einblicke in den Umfang und der Verteilung ihrer Ausgaben zu bieten.
Somit ist das Ziel der Anwendung stetige kleine Transaktionen zu reduzieren und den Nutzern einen einfachen Überblick gemeinsamer Finanzen zu bieten.

## 4.2 Workflow

Start der Anwendung ist der Login- beziehungsweise Registrierungsprozess. Ein neuer Nutzer der App kann sich hier einen neuen Account mittels E-Mail und Passwort erstellen und wird nach einer erfolgreichen Registration auf die Startseite weitergeleitet. Besitzt der Nutzer bereits einen Account, kann er sich hier einfach Einloggen und wird ebenfalls weitergeleitet.
Ab diesem Punkt besteht die Anwendung zum derzeitigen Entwicklungsstand aus drei Kernabschnitten. Dem Homescreen, den Statistiken und der Erstellung/Bearbeitung von Expenses. Eine Expense in Coin stellt eine Visualisierung einer Ausgabe dar. In 4.2 sieht man Beispiele dieser drei Abschnitte.
Beginnend mit dem Homescreen, welcher dem Nutzer Überblick über alle Ausgaben von sich und den anderen Teilnehmern bietet. Oben rechts befindet sich ein PopUp Menü. Darüber kann man Einstellungen über Kategorien und dem Nutzerprofil vornehmen, sowie einen Logout durchführen.
Jeder Nutzer kann einen Individuellen Avatar wählen sowie seinen Benutzernamen ändern. Dafür wurde eine Liste an Verfügbaren Avatars bereitgestellt, welche zur Verfügung stehen. In Abbildung 8 kann man eine Vorschau sehen, wie diese Seite aussieht. Da diese Anwendung noch in den Startlöchern steht, gibt es viele Funktionen die noch nicht implementiert worden sind. Für die Zukunft wäre es hier beispielsweise noch möglich, eine Verwaltung von Gruppen zu ergänzen. Gruppen sind in der Anwendung ein gewünschtes Feature, welches in Zukunft umgesetzt werden soll. Damit soll es möglich sein mit unterschiedlichen Nutzern, unterschiedliche Ausgaben zu verwalten.

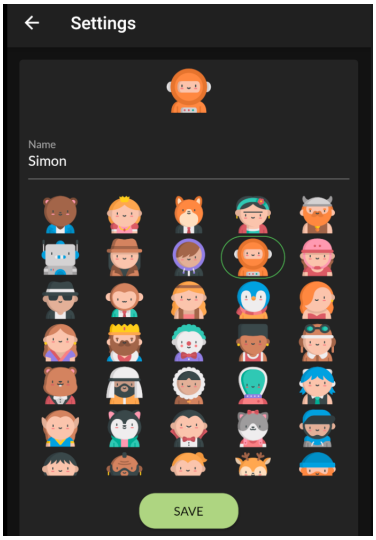
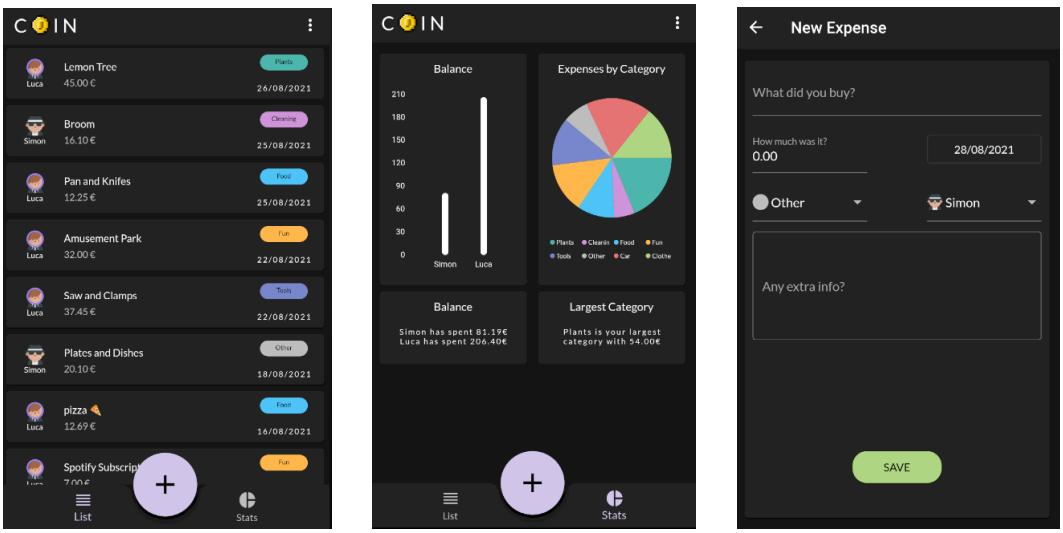



Abbildung 8: Einstellungen [5]

In der Mitte des Homescreens befindet sich die Liste mit Ausgaben, diese kann man sich im Detail anschauen sowie bearbeiten und löschen.
Die Seite der Statistiken ermöglicht einen Überblick über die Finanzlage einzelner Nutzer sowie eine Gewichtung der einzelnen Kategorien nach Betrag. Zukünftige Erweiterungen könnten hierbei noch implementiert werden. Beispielsweise ist es bis jetzt noch nicht möglich, gewisse Ausgaben für andere Personen in Zahlung zu setzen, damit könnte ein System für Rückstände einzelner Personen realisiert werden.
Als letzter Hauptbestandteil der Anwendung wird das Erstellen der Ausgaben betrachtet. Man hat die Möglichkeit für diverse Informationen. Zu diesem Zeitpunkt hat man hier die Möglichkeit Basis Informationen der Ausgabe wie Name, Betrag, Datum und eine kurze Beschreibung anzugeben. Diese Ausgabe lässt sich dann in eine Kategorie einteilen, die einem durch die Statistiken einen Überblick über Ausgaben Gebiete geben kann. Anschließend kann ein Nutzer gewählt werden, welcher für die Ausgabe zuständig war.
Hier soll in Zukunft noch, wie bereits erwähnt, ein System hinzugefügt werden, welches erlaubt diese Ausgabe einem speziellen Nutzer in Zahlung zu stellen.
Aufgrund der Wahl des Backends und einer NoSQL Datenbank, ist es möglich hier Änderungen des Datenmodells im Verlaufe der Anwendung stetig anzupassen, ohne größere Systemanpassungen vorzunehmen. Dies wird in Kapitel 4.4 und 4.4.2 genauer behandelt.




(a) Homescreen [5] (b) Statistics [5] (c) Expense [5]

Abbildung 9: Kernabschnitte

## 4.3 Architekturmuster

Das Architekturmuster beschreibt den Aufbau des Projektes in der Softwareentwicklung. Dabei wird das Projekt in mehrere Bausteine aufgeteilt, um so im späteren Verlauf leichter Änderungen und Erweiterungen vornehmen zu können.
Bei dieser Anwendung wurde sich an einem bekannten Muster **Model View Controller (MVC)** orientiert. MVC teilt das System in drei unterschiedliche Komponenten auf. Aus dem Name lassen sich daher die Komponenten, **Model (Modell)**,**View (Präsentation)** und **Controller (Steuerung)** herleiten und kann in Abbildung 10 Visuell nachvollzogen werden.
Das Model beinhaltet die Daten, welche durch die Präsentation angezeigt werden sollen. Die Präsentation ist für die Darstellung zuständig und wird durch Änderungen im Modell informiert um gegebene Präsentations Änderungen vorzunehmen. Die Steuerung ist für die Kommunikation zuständig. Die Präsentation informiert hierbei die Steuerung über Benutzerinteraktionen. Dadurch können Änderungen im Modell und damit Anpassungen in der Präsentation vorgenommen werden. [11]
In der Anwendung Coin wurde sich an diesem Muster orientiert und ein ähnlicher Aufbau realisiert. Die Software wird in Model, View und Services aufteilt. Services stellt in diesem Fall eine Art Geschäftslogik dar, in der die Logik zur Kommunikation mit Services wie Firebase zuständig ist. Zusätzlich dienen dieses Services ebenfalls als Kommunikation zwischen dem Modell und Präsentation. Das Modell enthält die Objekte in welche beispielsweise die Daten aus der Datenbank umgewandelt werden. In der Präsentation werden die ganzen Widgets und damit der Widget-Baum realisiert.

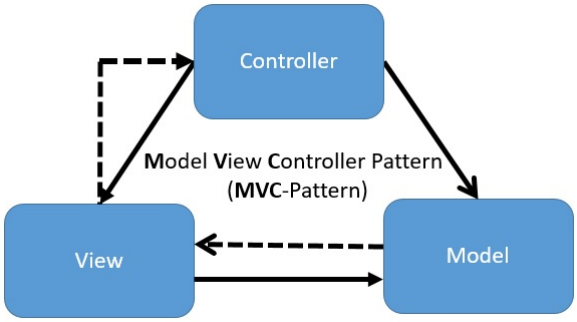



Abbildung 10: Model View Controller [12]

## 4.4 Backend

Als Backend für die Anwendung Coin wurde die von Google Entwickelte Plattform Firebase genutzt. Firebase stellt eine Backend-as-a-Service Lösung für die App Entwicklung zur Verfügung. Mittels eines Software Development Kits (SDK) stellt Firebase viele Services bereit, darunter zählt unter anderem ein Authentifizierungsservice und eine Datenbank. Firebase arbeitet ebenfalls gut mit Flutter zusammen, da dies auch ein von Google Entwickeltes Framework ist. Deshalb haben wir uns dazu entschieden, unser backend auf der Basis von Firebase aufzubauen.

### 4.4.1 Authentifizierung

Einstiegspunkt der Anwendung ist die Login beziehungsweise Registrierungsseite. Die meisten Anwendungen heutzutage brauchen ein System der Nutzerzuordnung. Firebase stellt hier über den Authentifizierungsservice SDKs bereit, welche das Implementieren der Logik vereinfacht. Der Service bietet unterschiedliche Arten der Authentifizierung an. Ein Beispiel hierfür wäre die Möglichkeit über E-Mail und Passwort, aber auch Soziale Medien wie Facebook oder Twitter sind als Authentifizierung möglich.[13] Für diese Anwendung wurde eine einfache E-Mail und Passwort Zuordnung genutzt.
In den unten stehenden Listings 10 und 11 wird mittels Firebase Authentifizierung die Registrierung und die Anmeldung mit bestehendem Account realisiert. In Listing 10 erkennt man in der Vorletzten Zeile, dass der User, welcher von Firebase zurückgegeben wird, auf einen CoinUser() umgewandelt wird, da im späteren Verlauf der Anwendung mit diesem Objekt gearbeitet wird.

```
Future registerWithEmailAndPassword(String name, String email, String
    password) async {
  UserCredential result = await _firebaseAuth.
      createUserWithEmailAndPassword(email: email, password: password);
  User? user = result.user;
  //Create a user record in firestore
  await FirestoreService().userSetup(CoinUser(uid: user!.uid, name: name));
  return CoinUser.fromFirebaseUser(user);
```

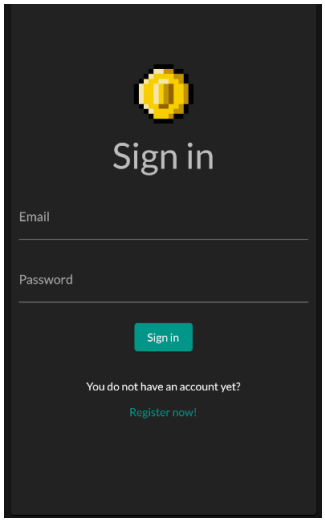
```
}
```

Listing 10: Register [5]

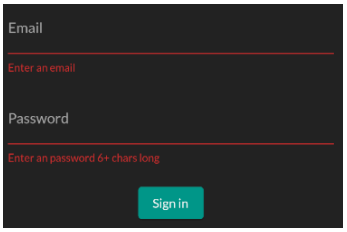
```
Future signIn(String email, String password) async {
  UserCredential res = await _firebaseAuth.
  signInWithEmailAndPassword(email: email, password: password);
  User? user = res.user;
  return CoinUser.fromFirebaseUser(user);
}
```

Listing 11: Sign In [5]

Mittels Form Validierung kann hier im Voraus auf Eingabefehler geprüft werden. Weitere Prüfungen, wie Passwort, oder ob ein Nutzer zu gegebener E-Mail Adresse bereits existiert, werden dann im Hintergrund von Firebase übernommen. In Abbildung 4.4.1 sieht man den Login Bildschirm der Anwendung. Daneben befindet sich ein Beispiel einer Validierung. Versucht ein Nutzer sich ohne E-Mail und Passwort anzumelden, werden die Felder Rot Markiert und ein Hilfstext wird bereitgestellt.




(a) Login [5]




(b) Login Validation [5]

Abbildung 11: Login Bildschirm

Die SDK bietet ebenfalls einen Listener an, der über Änderungen des Benutzers benachrichtigt. Diese Events werden über einen Stream zurückgegeben, auf den mit beispielsweise einem Streamprovider zugegriffen werden kann. Dardurch kann man unter anderem auf Logouts reagieren und dementsprechend die Darstellung ändern, sobald durch den Provider ein re-build ausgelöst wird und auf die Login Seite zurückkehren.

### 4.4.2 Datenbank

Für die Anwendung 'Coin' wurde die Datenbank von Firebase namens Firestore genutzt. Diese ist eine dokumentenorientierte NoSQL-Datenbank. Das bedeutet im Gegensatz zu einer SQL-Datenbank, in der die Daten in Tabellen mit Zeilen und Spalten geordnet werden, dass die Daten hier als Dokumente gespeichert werden. Diese Dokumente werden in Sammlungen organisiert. Um eine Referenz zu einer SQL-Datenbank zu ziehen, könnte man ein einzelnes Dokument als eine Zeile in einer Tabelle ansehen.
Innerhalb eines Dokuments werden die Daten in unterschiedliche Felder gespeichert.
Firebase bietet grundsätzlich zwei Verschiedene Datenbanken an. Firestore und Realtime Database. Beides sind NoSQL Datenbanken, jedoch besitzt Firestore ein etwas einfacheres Schema in Form von Dokumenten und bietet etwas bessere Query Performance sowie Skalierbarkeit. Deshalb wurde für diese Anwendung Firestore ausgewählt. [14] In Abbildung 12 kann man ein Dokument aus der Datenbank sehen.

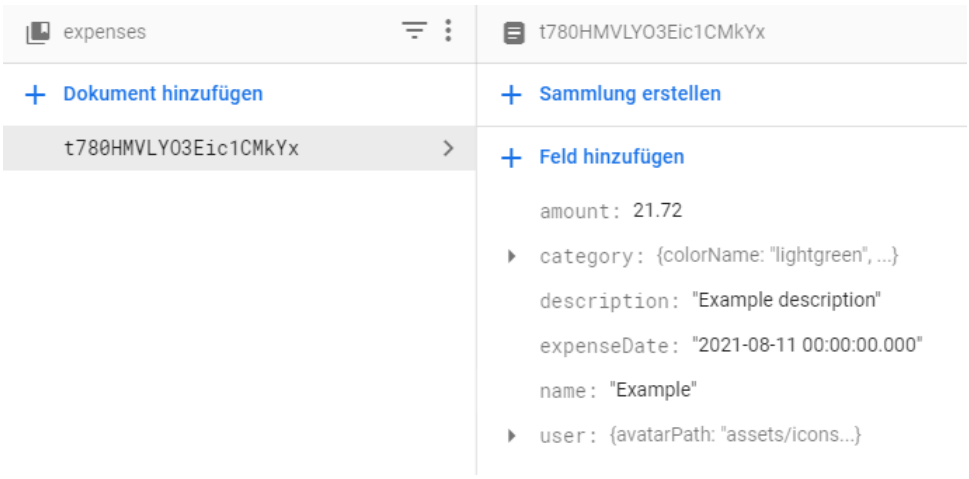



Abbildung 12: Dokument aus Firestore [5]

Auf der linken Seite der Abbildung 12 sieht man die ID eines Dokumentes. Ein Dokument muss immer eindeutig identifizierbar sein, dies kann wie in diesem Fall über eine zufällig generierte ID geschehen, oder über Individuelle Kriterien.
Auf der rechten Seite sieht man die Felder, denen dann am Ende die Werte zugeordnet werden. Es ist auch Möglich innerhalb von einem Dokument Untersammlungen zu erstellen, um so eine gewisse Hierarchie zu erlangen.

NoSQL Datenbanken bieten eine schemalose Herangehensweise, das bedeutet man hat theoretisch die Möglichkeit für jedes Dokument unterschiedliche Felder und Datentypen zu nutzen. Eine generelle Herangehensweise ist aber jedoch, sich etwas Gedanken über das

Datenmodell zu machen, um so unter anderem effektivere Queries erstellen zu können. Die Vorteile der Schemalosigkeit spiegelen sich eher in der Möglichkeit wieder, zu späteren Zeitpunkten gewisse Änderungen beziehungsweise Ergänzungen an Feldern und Typen vorzunehmen, ohne Probleme mit der Datenbank zu bekommen. [14]
Im Vergleich zu einer SQL Datenbank, in der mittels SQL Daten aus verschiedenen Tabellen in einer einzigen Abfrage zusammengeführt werden können, ist dies bei NoSQL nicht so einfach möglich. Deshalb kann es hier vorkommen, dass Daten doppelt gespeichert werden müssen, um so Daten aus unterschiedlichen Sammlungen ohne mehrfache Datenbank abfragen zu tätigen. Dies nennt man auch **Data denormalization**. Denormalization beschreibt diese redundanten Daten. Herkömmliche relationale Datenbanken normalisieren die Daten, da sie über Joins die unterschiedlichen Tabellen zusammenführen können. Da NoSQL Datenbanken diese Art von Query nicht ermöglicht, werden hier gewisse Daten redundant abgespeichert. Zu den Vorteilen davon gehören beispielsweise schnellere read Operationen und einfachere Queries. Nachteile einer denormalisierten Strategie sind write Zugriffe. Da Daten redundant vorkommen, müssen die neuen Daten an jeder Stelle aktualisiert werden. Dies kann man in Firestore über einen sogenannten WriteBatch realisieren, welcher eine Anzahl an write Operationen durchführt. [15]
In Abbildung 8 tritt solch eine Situation auf. Nutzer werden in einer eigenen Root Collection verwaltet und gleichzeitig in einem Expense Objekt selbst als Map gespeichert. Hier muss also bei einem Update des Nutzers eine Batch Operation ausgeführt werden, welche zusätzlich noch alle Objekte dieses Nutzers in den unterschiedlichen Expenses aktualisiert.

Um sich dies etwas genauer vorstellen zu können, wird als Beispiel der Aufbau des Datenmodells von Coin näher gebracht.
In Abbildung 13 ist die Struktur zu erkennen. Groups und Users stellen zwei Root-Kollektionen dar. Users beinhalten eine einfache Ebene mit Dokumenten. Groups hingegegen erstreckt sich über eine gewisse Hierarchie. Ein Dokument unter Groups enthält ein Feld mit dem Name der Gruppe sowie zwei Untersammlungen categories und expenses. In Abbildung 12 war der Aufbau einer expense zu sehen. Man sieht hier das Feld "category"welches eine Map darstellt. Hier kann man die Redundanz erkennen. Category wird hier extra als Map abgespeichert, obwohl es dazu auch eine Sammlung gibt. Neben dem oben beschrieben Nachteil mit der Datenredundanz, sind im Gegenzug Datenbankabfragen gegen diese Dokumente umso leichter.

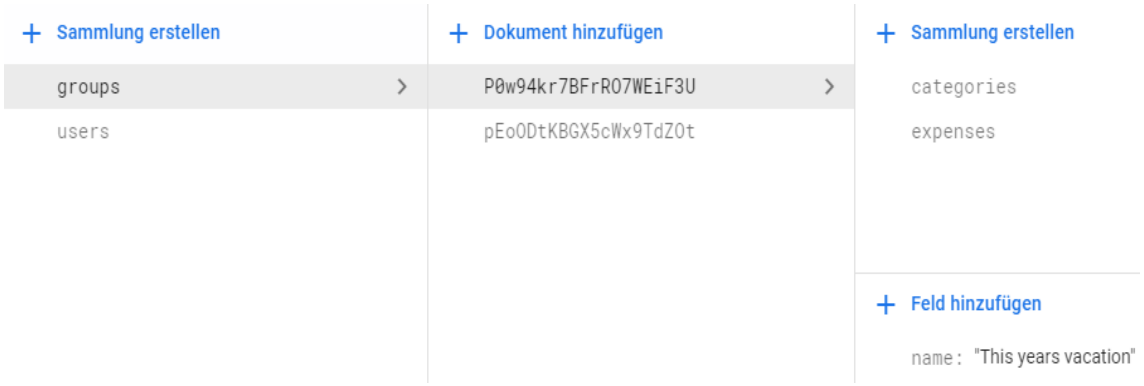



Abbildung 13: Datenbankmodell [5]

### 4.4.3 Datenbankzugriff

Um den Datenbankzugriff zu vereinfachen sollte man sich wie oben beschrieben Gedanken über das Datenmodell machen. Untersammlungen ermöglichen es eine Hierarchie zu erschaffen. Es gibt drei Methoden wie so eine Struktur aussehen könnte:

**Verschachtelte Daten in Dokumenten**
Wie oben beschrieben, ist es möglich komplexe Objekte wie Arrays oder Maps in einem Feld zu speichern. Diese Variante funktioniert gut, solange diese Objekte eine einfache Struktur haben und keine hohe Notwendigkeit zur Skalierung besteht. In diesem Fall ist dies eine einfache und effektive Herangehensweise und flacht die Datenstruktur etwas ab. [14]

**Untersammlungen**
Sollte die Struktur Skalierbar sein, sollte man die Objekte welche Wachstumspotential bieten, in Untersammlungen organisieren. Untersammlungen bieten den Vorteil, dass die Elterndokumente mit steigender Skalierung der Objekte nicht mitwachsen. Hier werden diese Objekte in einer eigenen Sammlung innerhalb des Dokuments gespeichert. [14]

**Root-Level Sammlungen**
Ist eine Sammlung nicht auf bestimmte Dokumente begrenzt, bietet es sich hier an diese Sammlung als Root-Sammlung zu nutzen. Hier wird eine Sammlung direkt auf der Root-Ebene erstellt und nicht innerhalb eines Dokumentes. [14]

Generell funktioniert der Zugriff immer nach einem Schema. Um einer Hierarchie zu folgen, wird immer abwechselnd auf eine Sammlung und dann wieder auf ein Dokument zugegriffen. In Listing 12 wird der Zugriff auf die Sammlung Expenses dargestellt.

**Read**
Man sieht hier gut die Iteration über die Hierarchie. Wichtig ist hierbei noch das Ende.

Da eine Datenbank wie Firebase Dokumente in einem Art JSON Format darstellt, gibt ein Read Zugriff eine Map zurück. Die Antwort von Firestore wird dann in ein Expense Objekt umgewandelt. Die fromFirebase() Methode erstellt für jedes Dokument aus dem Stream ein Expense Objekt, welches die Daten aus den Feldern der Map bezieht.

```
Stream<List<Expense>> get expenses {
    FirebaseFirestore db = FirebaseFirestore.instance;
    return db.collection('groups').doc(currentGroup).collection("expenses")
        .orderBy("expenseDate", descending: true)
        .snapshots()
        .map((snapshot) => snapshot.docs
            .map((document) => Expense.fromFirebase(document))
            .toList());
  }
```

Listing 12: Datenbank read [5]

**Write**

Bei einem Write Zugriff ist der Anfang wieder gleich. Es wird die Hierarchie durch iteriert und die set() Methode genutzt, welche bestehende Daten überschreibt. Falls das Dokument noch nicht existiert, wird ein neues mit den Daten anlegt. Wie oben schon beschrieben, speichert Firestore die Dokumente als Map. Deshalb muss man die Daten als Map übergeben, damit Firestore diese als Dokument speichern kann.

```
Future writeExpense(Expense expense) async {
    FirebaseFirestore db = FirebaseFirestore.instance;
    return db.collection('groups').doc(currentGroup).collection("expenses")
        .doc(expense.id)
        .set({
            'name': expense.name,
            'amount': expense.amount,
            'description': expense.description,
            'category': expense.category!.toMap(),
            'user': expense.user!.toMap(),
            'expenseDate': expense.expenseDate.toString(),
    });
```

# 5 Vergleich mit anderen UI-Frameworks

## 5.1 Xamarin

Xamarin ist eine Tochtergesellschaft und damit das Cross-Plattform Framework von Microsoft. Es erweitert die .NET Plattform und ermöglicht die Entwicklung in C#. Ebenso verspricht Xamarin eine native Performance der Anwendung. Mit Xamarin lassen sich alle Dinge, welche sich mit beispielsweise Java, Kotlin oder Swift realisieren lassen, auch mit C# umsetzen und bietet somit 100% native Entwicklung. Aus Xamarin-Code können iOS, Windows und Android Apps generiert werden. Xamarin verspricht eine durchschnittliche plattforübergreifende Wiederverwendbarkeit von 90% des Codes. Die gesamte 'Business-Logik' kann in C# entwickelt werden und plattformübergreifend genutzt werden. Plattformspezifische Elemente müssen gesondert behandelt werden. Daher eignet sich Xamarin vor allem dann, wenn die Anwendung viel BusinessLogik beinhaltet. Xamarin.Forms liefert viele vorgefertigte UI-Komponenten, die in einer XML-Struktur definiert werden. Xamarin bietet sich vor allem dann an, wenn bereits ein Microsoft-/.NET-System besteht, mit der die Xamarin-App kommunizieren soll. [16]

## 5.2 React Native

React Native ist ein von Facebook entwickeltes Framework, welches ebenfalls
Plattformübergreifende Entwicklung ermöglicht. Es basiert auf React-JS, somit nutzt der Entwickler JavaScript, welches anschließend in native Plattform UI gerendert wird. Mit ReactNative lassen sich auch Teile mit Swift, Java oder Kotlin implementieren.
React Native bringt ein Basispaket an Komponenten mit, welche sich direkt in die native UI umwandeln und gibt somit ein Gefühl von einer nativen Anwendung. Das bedeutet also React Native kann direkt auf die nativen APIs zugreifen.
Durch das erstellen von Plattformspezifischen Komponenten kann man die Anwendung ebenfalls aus einer Codebasis für zwei verschiedene Plattformen nutzen. Ähnlich wie Flutters Hot-Reload bietet auch React Native ein "Fast-Refresh", was die Entwicklung deutlich beschleunigt und vereinfacht. Zu den unterstützten Plattformen werden iOS und Android gezählt. Tatsächlich bietet React Native noch keine offizielle Möglichkeit, Webanwendungen zu erstellen. Durch die weitverbreitet Sprache JavaScript haben Entwickler aus der Web Branche einen leichten Einstieg in die Mobile Entwicklung.[17]

# 6 Ausblick in die Zukunft

Mit Flutter hat Google ein eigenes Cross-Platform-Framework ins Rennen geschickt. Im mobilen Cross-Platform-Framework Markt ist Flutter bereits angekommen mit einigen namenhaften Marken, die ihre Apps mit Flutter entwickelt haben (BMW, New York Times, Ebay, Reflecty, ...). [18] Die hauseigene Programmiersprache Dart ist für das Entwickeln von Benutzeroberflächen optimiert, syntaktisch elegant und liefert eine hervorragende Developerexperience. Sie nimmt positive Aspekte von JavaScript und von objectorientierten Programmiersprachen und vereinigt diese auf gelungene Weise. Die Vielzahl der Widgets in der Flutter SDK macht es enorm einfach komplexe UIs in kürzester Zeit zu gestalten.

Vor allem bei der Entwicklung mobiler Apps bietet sich die Entscheidung für ein Cross-Platform-Framework an. In der Regel soll mit einer App sowohl der iOS als auch der Android Markt erreicht werden. Dafür zwei Apps in der jeweils nativen Umgebung zu entwickeln ist zeitaufwendig. Zusätzlich ist der Unterschied von iOS und Android Geräten nicht immens, wodurch nur selten auf Eigenheiten der Systeme eingegangen werden muss. Trotz allem ist Flutter noch ein sehr junges Framework, bei dem die kommenden Jahre zeigen werden, ob es sich vor allem im mobilen Cross-Platform-Development als Standard etablieren wird.

Die weiteren Plattformen wie das Web oder Desktop Applikationen wird mit großer Wahrscheinlichkeit nicht der Fokus des Flutter-Teams sein. Die Welt des Web-Development ist bereits gesättigt mit JavaScript-Frameworks, die mit dem für das Web native HTML, CSS und JS arbeiten. Diese Frameworks werden für die meisten Aufgaben mit großer Sicherheit ein besseres Ergebnis liefern, als eine aus Dartcode in JavaScript übersetzte FlutterWeb Anwendung. Mit Angular hat Google bereits sein eigenes dieser JavaScript Frameworks auf dem Markt, dessen Entwicklung durch Flutter nicht beendet werden wird.
Trotzdem bietet FlutterWeb eine Alternative zu der etablierten Webtechnologie (HTML, CSS, JS). Mit wenig Aufwand, kann ein Team, dessen primärer Fokus auf einer mobilen App liegt, eine Version einer WebApp für Endnutzer bereitstellen.

# 7 Fazit

Zuletzt lässt sich sagen, dass das Projekt erfolgreich abgeschlossen werden konnte. Die Anwendung 'Coin', die im Laufe dieser Arbeit entwickelt wurde stellt alle Funktionen bereit, die für ihren Nutzen erforderlich sind. Weitere Features, die die App voll funktionsfähig machen würden sind zum Beispiel das Anlegen und Verwalten mehrerer Gruppen, in der ein Nutzer Mitglied sein kann. Dafür wäre ein Einladungssystem notwendig über das ein Gruppenersteller Mitglieder in seine Gruppe hinzufügen kann.

Über den Umfang des Projekts hinaus wurde der Service Firebase als 'serverless-Architecture' für Authentifiztierung und Datenbankdienste genutzt. Durch diese Technologie war es möglich den Funktionsumfang überhaupt zu realisieren und auf den Frontend-Development Prozess zu konzentrieren. Die Arbeit mit einer NoSQL-Datenbank ist zugeschnitten auf Daten die zur Präsentation dienen. Der Umgang damit stellt für jeden Entwickler mit SQL-Hintergrund eine Umstellung und Lernerfahrung dar. Die Integration in Firebase lief aufgrund guter Kompatibilität reibungslos ab und auch die von Firebase angebotenen Flutter-Plugins stellten keine Hürde in der Entwicklung dar.

Laut [19] hat Flutter im Jahr 2021 die Spitze zum meistgenutzten mobile App Framework erklommen. Die nahe Zukunft wird zeigen, ob Flutter es schafft sich an dieser Stelle zu halten. Auf alle Fälle birgt Flutter als Ganzes mächtiges Potential. Diese Arbeit soll den grundlegenden Aufbau und die Funktionsweise von Flutter darbieten. Die Projektapplikation 'Coin' soll demonstrieren, zu was das Flutter-Framework fähig ist, auch wenn sie nur einen Bruchteil des Umfangs der Flutter SDK zeigen kann. Der rasche Aufstieg spricht für das junge Framework und ist ein Signal an jeden Medien-Designer und Frontend-Developer in der mobilen App Entwicklung. Weitere Informationen zu allen Aspekten von Flutter sind auf 'https://flutter.dev/' zu finden.

# Literatur


[1] Wikipedia. Flutter (Software). http://de.wikipedia.org/w/index.php?title=Flutter%20(Software)&oldid=210551157. Accessed 21.August.2021.

[2] Dart documentation. https://dart.dev/. Accessed 23.August.2021.

[3] chrissikraus. Was ist eine Cross-Platform App? https://www.dev-insider.de/was-ist-eine-cross-platform-app-a-898699/. Accessed 22.August.2021.

[4] Ryan Smith. Top 10 Best Cross Platform App Development Frameworks in 2021. https://www.codenameone.com/blog/top-10-best-cross-platform-app-development-frameworks-in-2021.html. Accessed 25.August.2021.

[5] Eigene Aufnahme.

[6] Flutter Layout. https://flutter.dev/docs/development/ui/layout. Accessed 24.August.2021.

[7] Flutter documentation. https://flutter.dev/docs. Accessed 21.August.2021.

[8] Material Design. https://material.io/design. Accessed 25.August.2021.

[9] Flutter Web. https://flutter.dev/web. Accessed 26.August.2021.

[10] Dart Codelabs. https://dart.dev/codelabs/dart-cheatsheet. Accessed 27.August.2021.

[11] Wikipedia. Model View Controller. http://de.wikipedia.org/w/index.php?title=Model%20View%20Controller&oldid=215088958, 2021. Accessed 25.August.2021.

[12] Model View Controller Pattern Definition & Erklärung: Datenbank, DWH & BI Lexikon. https://www.datenbanken-verstehen.de/dbv/uploads/model_view_controller_pattern.jpg, 2019. Accessed 25.August.2021.

[13] Firebase Authentication. https://firebase.google.com/docs/auth. Accessed 25.August.2021.

[14] Cloud Firestore. https://firebase.google.com/docs/firestore/. Accessed 25.August.2021.

[15] Wright, Gavin and Vaughan, Jack. What is denormalization and how does it work? https://searchdatamanagement.techtarget.com/definition/denormalization.

[16] Was ist Xamarin? https://docs.microsoft.com/de-de/xamarin/get-started/what-is-xamarin. Accessed 25.August.2021.

[17] React Native - Learn once, write anywhere. https://reactnative.dev. Accessed 25.August.2021.

[18] Flutter Showcase. https://flutter.dev/showcase. Accessed 29.August.2021.

[19] Cross-platform mobile frameworks used by software developers worldwide from 2019 to 2021. https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/. Accessed 29.August.2021.

# Abbildungsverzeichnis

# Listings