

Entwicklung einer Cross-Plattform-App zur Datenerfassung würzig belegter Fladenbrote

Bearbeiter:	Tony Spegel Stiftsgasse 32 07407 Rudolstadt
Betreuer:	Prof. Herr Stepping
Matrikel-Nr.:	639872
Fachsemester:	8
Studiengang:	Wirtschaftsingenieurwesen / E-Commerce
Modul:	Mobile App Entwicklung II
Eingereicht am:	11.07.2019

Inhaltsverzeichnis

1	Motivation	1
2	Ziele	1
3	Übersicht App-Technologien	2
4	Flutter's Performance	2
5	Dart	3
6	Deklaratives UI	4
7	Herausforderungen	6
7.1	Layout & Design	6
7.2	Daten zwischen Widgets teilen	9
7.3	Fehlende Date-Library	12
8	Fazit	12

Abkürzungsverzeichnis

AOT	Ahead-of-time
FAB	Floating Action Button
SDK	Software Development Kit
SPA	Single-Page Application
SPOT	Single Point of Truth
TKP	Tiefkühlpizza
UI	User Interface
VM	Virtual Machine
WORA	Write once, run anywhere

Abbildungsverzeichnis

1	Flutter-Logo	2
2	Dart-Logo	2
3	Vergleich deklaratives & imperatives UI	4
4	Beispiel: imperativer Stil	5
5	Beispiel: deklarativer Stil	5
6	App: Hauptansicht	6
7	App: Formular	7
8	App: Dismissible	8
9	App: Formular mit Shared-Data	9
10	Dart: ChangeNotifier	10
11	Dart: MultiProvider	11
12	Dart: Letzter Tag des Monats	12

Tabellenverzeichnis

1	Übersicht Arten von Apps	2
---	------------------------------------	---

1 Motivation

Diese Ausarbeitung dokumentiert die Entwicklung einer Cross-Plattform-App um die Daten von meist würzig belegten Fladenbrotten erfassen zu können. Damit gemeint sind vor allem Pizzen sowie deren Varianten. Die ursprüngliche Idee entstand durch einen Beitrag des Subforums *r/dataisbeautiful* der Social-News-Aggregator-Plattform Reddit. Dieses Subforum legt besonderen Wert darauf Datensätze möglichst sinnvoll, ansprechend und zugänglich aufzubereiten. Nicht selten sind diese Datensätze eher skurril und handeln, wie in diesem Fall, auch von Lebensmitteln. Da ich unter anderem häufig und gern Pizzen esse, lag der Entschluss nah, eben diese zu erfassen. Motiviert durch den Einstieg im Wahlmodul *Mobile App Entwicklung I* weiter native Apps zu programmieren sowie aus privatem Interesse, stand die Entscheidung schnell, dieses Mal eine Cross-Plattform-Technologie zu nutzen. Grob zusammengefasst ergeben sich dabei im nächsten Abschnitt folgende Ziele.

2 Ziele

- Cross-Plattform-Technologie nutzen
- Ansprechende App im Material-Design
- Umsetzung als Single-Page Application (SPA)
- Pizzen und deren Daten erfassen/darstellen
- Cloud NoSQL-Datenbank *Firestore* nutzen

3 Übersicht App-Technologien

Um Apps zu entwickeln gibt es viele Möglichkeiten. Diese lassen sich grob in folgende Arten einteilen

Art	Charakteristik
Hybrid	Web-Apps werden im nativen Kontext in einer WebView eingebunden
Native	Adressieren konkrete Zielplattformen und deren Programmiersprachen. Android (Java, Kotlin, Dart), iOS (Objective-C, Swift)
Web Apps	Über einen Server bereitgestellte plattformunabhängige Anwendungen

Tab. 1: Übersicht Arten von Apps

Ich bin großer Fan von Write once, run anywhere (WORA) und entwickle üblicherweise vor allem Web-Apps. Um etwas neues zu lernen und daran zu wachsen, entschied ich mich, dieses Mal dazu eine Cross-Plattform-Technologie zu nutzen. Die Entscheidung fiel dabei auf das von Google entwickelte Open Source User Interface (UI)-Kit *Flutter*.



Abb. 1: Flutter-Logo



Abb. 2: Dart-Logo

4 Flutter's Performance

Eine der Besonderheiten von *Flutter* ist es, dass dieses UI-Kit weder eine WebView noch die vom Betriebssystem mitgelieferten Widgets benutzt. Widgets sind in der Welt von *Flutter* alles von Bedienelemente bis hin zu Layout-Helfern. Statt diese mitgelieferten Widgets zu nutzen, setzt *Flutter* auf eine eigene Rendering-Engine welche häufig mit einer 2D-Spiele-Engine verglichen wird. Eines der Entwicklungsziele von *Flutter* war es nämlich, besonders performante Apps entwickeln zu können welche mit einer hohen Hertz-Zahl (60-120 Hz) laufen. Mit diesem Ansatz ist es möglich, die gesamte UI über den Grafikchip des Systems zu berechnen und die CPU zu entlassen. Dadurch wird es außerdem möglich, Änderungen die man im Code vornimmt, nahezu ohne Verzögerung in der App zu sehen, ohne das ein Neuladen oder ähnliches notwendig ist.

5 Dart

Flutter-Apps werden in der Programmiersprache *Dart* entwickelt. *Dart* ist eine objektorientierte, klassenbasierte Sprache mit Garbage-Collector und wird vor allem durch *Google* entwickelt. Sie wurde vor allem durch *C#*, *Erlang*, *JavaScript*, *Smalltalk* sowie *Strongtalk* beeinflusst. Im *Dart*-Software Development Kit (SDK) enthaltenen sind: die *Dart*-Virtual Machine (VM), dem Compiler *dart2js* welcher es erlaubt *Dart* in *JavaScript* zu kompilieren sowie der Paketmanager *Pub*. *Dart*-Code kann mittels Ahead-of-time (AOT) kompiliert werden. Dies hat zum Vorteil das Programmcode vor der Ausführung in native Maschinensprache übersetzt wird und somit wesentlich schneller ausgeführt werden kann. Apps geschrieben mit *Flutter* werden automatisch per AOT kompiliert.

6 Deklaratives UI

Im Gegensatz zu imperativen Frameworks wie dem *Android SDK* oder dem *iOS UIKit* handelt es sich bei *Flutter* um ein so genanntes deklaratives Framework. Dies bedeutet, dass das UI von Flutter immer aktuellen *"State"* reflektiert. Wird beispielsweise eine Option in den Einstellungen einer App geändert, so ändert sich der *"State"* der App welches das Neuzeichnen der App auslöst (eine Checkbox wird gefüllt, ein Switch aktiviert). Imperativ würde bedeuten, dass es Methoden wie *widget.setText* gibt um Werte direkt zu ändern. Hier wird der *"State"* geändert und das UI wird komplett neu gezeichnet.

Technisches Beispiel

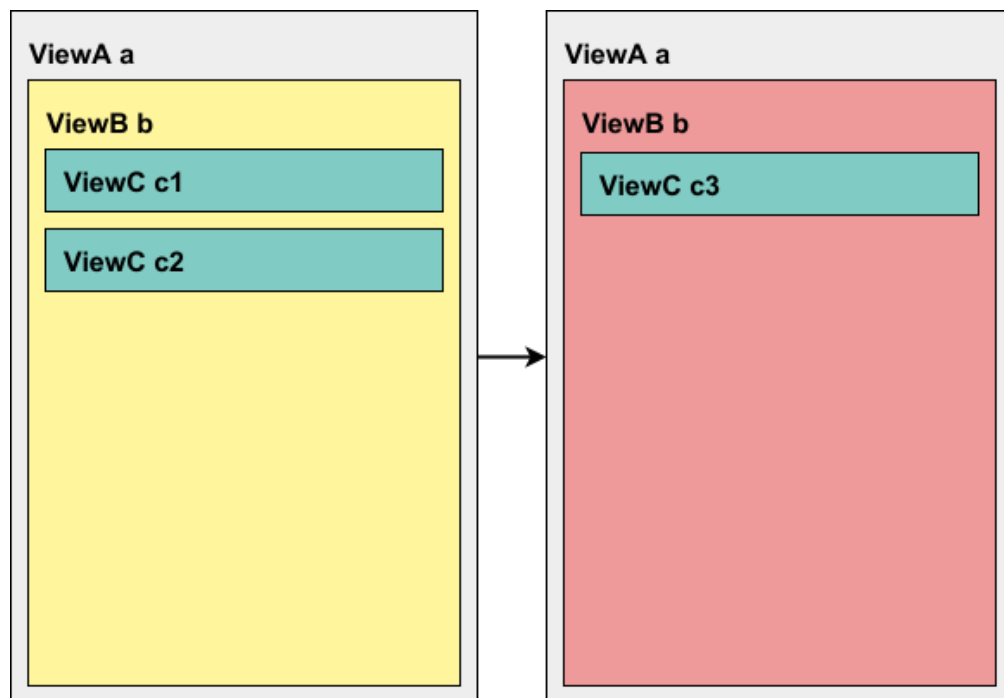


Abb. 3: Vergleich deklaratives & imperatives UI

Im imperativen Stil würde man eine Instanz *b* des so genannten *Owners* der View *ViewB* nutzen und mit Hilfe eines Selektors wie beispielsweise *findViewById* Änderungen auf dieser anwenden (und somit diese implizit invalidieren). Das kann ungefähr so aussehen:




```
// Imperative style
b.setColor(red)
b.clearChildren()

ViewC c3 = new
ViewC()
b.add(c3)
```

Abb. 4: Beispiel: imperativer Stil

Außerdem müsste dieses Setup im Konstruktor von *ViewB* dupliziert werden, da die UI (der Single Point of Truth (SPOT)) möglicherweise die Instanz *b* überlebt. Im deklarativen Frameworks sind View-Konfigurationen (wie *Flutter's* Widgets) unveränderlich (immutable) und an sich nur simple Blaupausen. Um ein UI zu ändern, löst ein Widget einen Rebuild auf sich selbst aus (in *StatefulWidgets* häufig per *setState()*) und erzeugt damit einen neuen Widget-Subtree.



```
// Declarative
// Style
return ViewB(
  color: red,
  child: ViewC(),
)
```

Abb. 5: Beispiel: deklarativer Stil

7 Herausforderungen

Die folgenden Abschnitte zeigen Auszüge, die sich bei der Entwicklung als besonders anspruchsvoll herausgestellt haben. Dabei werden zum einen technologische als auch Aspekte des Layouts und Designs betrachtet.

7.1 Layout & Design

Die App besteht grundsätzlich aus drei Bereichen: einer Liste, einem Informations-Panel und einem Formular. Diese Bereiche sollen nahtlos ineinander übergehen um ein möglichst professionellen Gesamteindruck zu vermitteln. Das finale Layout entspricht in der Hauptansicht der untenstehenden Abbildung.

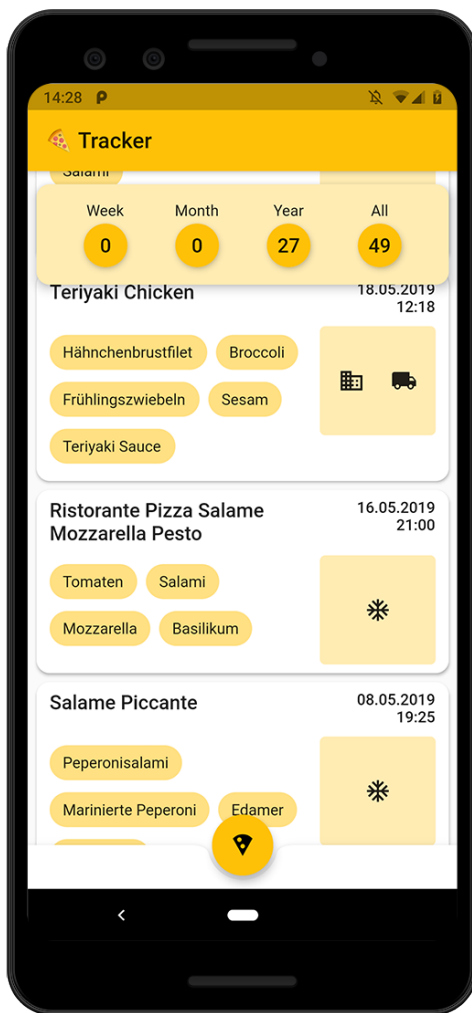


Abb. 6: App: Hauptansicht

Die Anordnung von Elementen erfolgt bei Flutter wie im Web: Die Reihenfolge im Quellcode gibt zugleich deren Reihenfolge auf der Y-Achse an. Elemente können sich dabei (auf der Z-Achse) nicht überlagern. Um Elemente übereinander darzustellen gibt es im Web die Property *z-index*, in Flutter nutzt man das *Widget Stack*. Im *Stack-Widget* ist das erste Element das auf der Z-Achse niedrigste Element. Grundsätzlich lassen sich viele Layout-Konzepte aus dem Web in Flutter übertragen. So gibt es Widgets wie *Row*, *Column*, *Grid*, *Flex* und *Expanded* welche in ihrer Funktionalität gleich ihren Web-Pendants entsprechen. Interessant ist, dass *Row* allerdings nicht umbricht, sobald zu viele Elemente in einer Zeile vorhanden sind und man für diesen Fall das *Widget Wrap* benutzen muss. Grundsätzlich bietet Flutter eine große Auswahl an Widgets, die über reine Layout-Funktionalitäten hinaus gehen und optische Standardwerte mit sich bringen. So gibt es das *Card-Widget* welches über dessen Properties *elevation* und *shape* einfache Mittel bereitstellt, um die Material-Design-typischen Schatten und abgerundeten Ecken zu erzielen.

Um nicht nur am oberen, sondern auch am unteren Ende der App (am sogenannten

Floating Action Button (FAB)) ein *Überfließen* zu erzielen, muss die Fläche der Liste auf die komplette Höhe des Bildschirms erweitert werden.

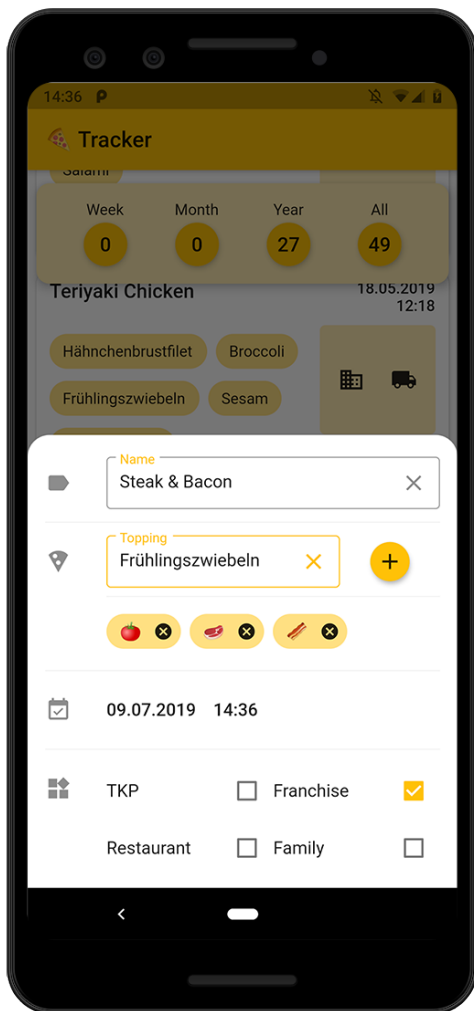


Abb. 7: App: Formular

Das Formular an sich besteht mehreren, grundsätzlich unabhängigen, Widgets:

- Name
- Toppings
- Datum & Uhrzeit
- Art der Pizza:
 - Tiefkühlpizza (TKP)
 - Franchise
 - Restaurant
 - Family
 - Delivery
 - Selfmade
- Ort an dem die Pizza gegessen wurde

Hierbei stellten sich vor allem die Widgets für die Eingabe der Toppings und der Art der Pizza als besonders anspruchsvoll heraus. Bei den Toppings kam zum einen zwei Mal das *Row*-, ein Mal das *Wrap* sowie für die Toppings selbst das *Chip*- (mit *onDeleted*-Methode), bei der Art der Pizza vor allem das *Grid*- sowie das *Checkbox*- (in Kombination) mit dem *Text*-Widget zum Einsatz.

Um im Formular abgerundete Ecken darstellen zu können, benötigt man das extern erhältliche Widget *showRoundedModalBottomSheet*. Allerdings kann auch dieses nicht über eine Höhe von 50% des Bildschirms hinaus gehen und zwingt einen somit zum scrollen.

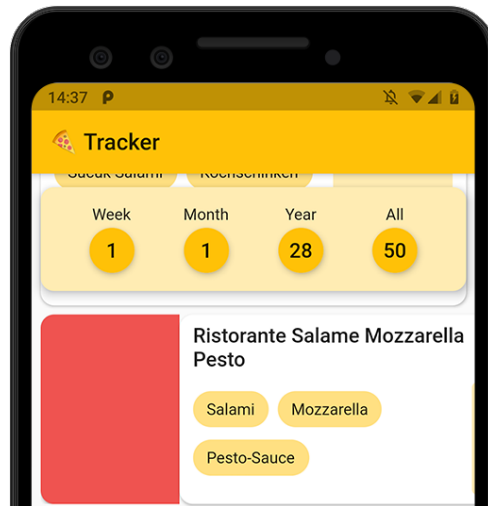


Abb. 8: App: Dismissible

Um Pizzen schnell wieder löschen (und später potentiell bearbeiten) zu können, bietet sich das *Dismissible*-Widget an. Dieses kann man beliebig um andere Widgets packen und diesem durch Wischen in bestimmte Richtungen verschiedene Aktionen zuweisen.

7.2 Daten zwischen Widgets teilen

Aufgrund der Art und Weise wie *Flutter* entworfen wurde, können Daten zwischen Widgets nur von oben nach unten beziehungsweise vom Eltern- zum Kind-Widget übergeben werden. In der unten stehenden Abbildung ist das Eingabeformular der App zu sehen. Dieses besteht wie bereits erwähnt aus verschiedenen Widgets. Jede "Zeile" ist ein Widget und enthält wiederum Widgets wie Formularfelder, *Checkboxen* oder *Chips*. Auf der Ebene des Formulars selbst, aber in der Abbildung nicht zu sehen, ist der Button zum senden der Daten. Würde das Formular ausschließlich aus Formularfeldern bestehen, wäre dies auch kein größeres Problem. In diesem Fall könnte man auf sämtliche Formularfelder direkt zugreifen. Jedoch besteht das Formular auch aus solchen Widgets die Listen repräsentieren und keine klassischen Formularfelder sind. Diese beiden Bereiche sind in der Abbildung cyanfarben umrandet. Zum einen kann eine Pizza mehrere Toppings und zum anderen gleichzeitig verschiedene Typen annehmen.

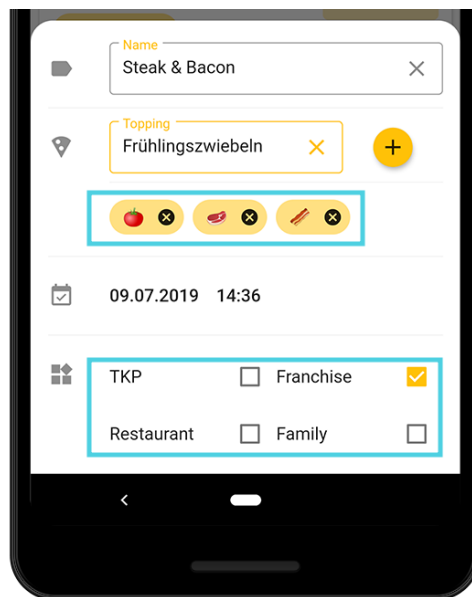


Abb. 9: App: Formular mit Shared-Data

Da diese Listen eine Ebene tiefer als der Button zum Absenden der Daten liegen, kann es keinen direkten Zugriff geben. In *Flutter* gibt es grundsätzlich zwei Arten wie Daten übergeben werden können:

1. Daten über den Konstruktor des Widgets hinzufügen
2. *Flutter's BuildContext*

Jedem Widget steht die Klasse *BuildContext* zur Verfügung. Diese erlaubt es einem Widget, mit Hilfe der folgenden Methoden Daten von jedem Vorfahren anzufordern.

- *inheritFromWidgetOfExactType(Type)*
- *ancestorStateOfType(TypeMatcher)*
- *ancestorWidgetOfExactType(Type)*

Besonders abfrageintensiven Operationen empfiehlt sich *inheritFromWidgetOfExactType(Type)*. Diese Methoden gehören zur Basis-Klasse *InheritedWidget*. Solche *Inherited*-Widgets lösen einen Rebuild im Konsumenten aus wenn deren State geändert wurde. Um das Handling mit diesen Methoden zu vereinfachen, nutze ich das Paket *provider*. Welches vor allem *Syntax Sugar* für *InheritedWidget* bietet. Um Daten einfach zwischen Widgets teilen zu können, empfiehlt es sich, sogenannte Daten-Klassen zur Haltung sowie Manipulation dieser zu vereinfachen. Dies kann nach folgendem Muster gelöst werden:



```
class Topping with ChangeNotifier
{
  List<String> _toppings = [];
  List<String> get toppings
    => _toppings;

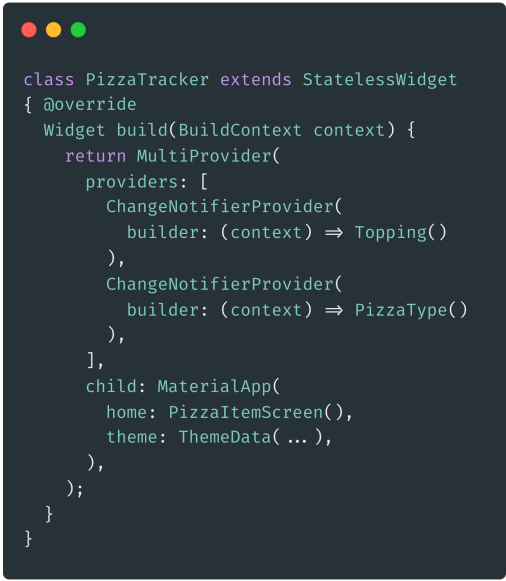
  addToppingToList(
    String topping
  ) {
    _toppings.add(topping);
    notifyListeners();
  }

  removeToppingFromList(
    String topping
  ) {
    _toppings.remove(topping);
    notifyListeners();
  }
}
```

Abb. 10: Dart: ChangeNotifier

Über das sogenannte *Mixin "ChangeNotifier"* lassen sich Eigenschaften und Methoden dieser Klasse auf die eigene Daten-Klasse übertragen. Die in den Methoden *addToppingToList* & *removeToppingToLis*t aufgerufene Methode *notifyListeners* informiert das Formular immer dann darüber, dass Änderungen an den Listen stattgefunden haben. Somit ändert sich der State der Widgets an sich sowie des Formulars, welches wiederum einen Rebuild auslöst. Um diese Daten-Klassen dann im Formular nutzen zu können, müssen

die Provider eine Ebene höher, in diesem Fall in der *main.dart* wie folgt eingebunden werden:



```
class PizzaTracker extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MultiProvider(
      providers: [
        ChangeNotifierProvider(
          builder: (context) => Topping()
        ),
        ChangeNotifierProvider(
          builder: (context) => PizzaType()
        ),
      ],
      child: MaterialApp(
        home: PizzaItemScreen(),
        theme: ThemeData( ... ),
      ),
    );
  }
}
```

Abb. 11: Dart: MultiProvider

7.3 Fehlende Date-Library

Um Filter für die verschiedenen Zeiträume setzen zu können, ist es notwendig, ausgehend vom jetzigen Zeitpunkt, den jeweiligen Start- und Endpunkt eines Zeitraums zu ermitteln. Wichtig bei der Berechnung von Zeiträumen ist unter anderem auch die korrekte Einbeziehung der Umstellung von Sommer- und Winterzeit oder ob eine Woche an einem Sonntag oder Montag beginnt. Im Web gibt es für solche und ähnliche Berechnungen Bibliotheken wie beispielsweise *Moment.js* oder *date-fns*. Solch eine Bibliothek gibt es allerdings noch nicht im Dart-/Flutterkosmos. Dementsprechend schrieb ich verschiedene Utility-Funktionen wie beispielsweise folgende um den letzten Tag eines Monats zu erhalten:

A screenshot of a code editor with a dark background and light-colored text. The code is a Dart function named `lastDayOfMonth` that takes a `DateTime` object named `today` as input. It returns a `DateTime` object representing the last day of the month. The function uses a conditional expression to handle the transition to the next year. The code is as follows:

```
DateTime lastDayOfMonth(DateTime today)
{ return (today.month < 12) ?
    DateTime(
        today.year,
        today.month + 1,
        0,
        23, 59, 59
    ) :
    DateTime(
        today.year + 1,
        1,
        0,
        23, 59, 59
    );
}
```

Abb. 12: Dart: Letzter Tag des Monats

8 Fazit

Die Entwicklung einer App mit *Flutter* brachte eine steile Lernkurve mit sich und bot eine neue Sicht auf bereits bekannte Konzepte. Als besonders herausfordernd stellte sich die Einarbeitung und Umsetzung von Layout und Design heraus. Auch das verhältnismäßig junge Alter des Frameworks brachte Herausforderungen mit sich; nicht immer gibt es aus anderen Frameworks bekannte Bibliotheken oder entsprechende Tutorials oder Blogbeiträge um sich Wissen neben der üblichen Dokumentation anzueignen. In diesem Projekt nicht betrachtet wurde die Cross-Plattform-Funktionalität mit der auch *iOS*-Apps entwickelt werden können. Alles in allem war dieses Projekt, auch wenn mit einem zwinkernenden Auge zu sehen, sehr lehrreich und spannend umzusetzen.