HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Prof. Dr.-Ing. Thorsten Weiss

V1.3

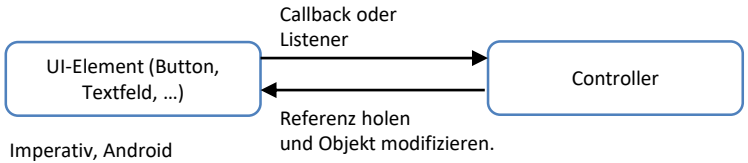
Flutter

- Flutter ist ein **Cross-Plattformsystem** zur Entwicklung von **Apps / Anwendungen für Mobile** und **Desktop**anwendungen.
- Flutter basiert auf der Programmiersprache **Dart**.
- Flutter nutzt OOP-Entwicklung (ähnlich zu Android) und erlaubt so die Entwicklung entsprechender Softwarearchitekturen.
- Die Erstellung von Oberflächen erfolgt programmatisch wie in der Webentwicklung üblich. Ein UI-Builder wie bei der Entwicklung von nativen iOS oder Android Apps ist nicht vorgesehen. Oftmals empfinden viele Programmierer die Designjustage mit CSS als herausfordernd und etwas nervig. Dies wurde bei Flutter recht klar gelöst, aber aus der ersten Erfahrung auch mit etwas mehr Boilerplatecode.
- **Flutter Oberflächen folgend dem deklarativen Stil.** Die UI ergibt sich aus dem State.

Deklarativ, React, Solid, Flutter

Bildquelle: https://docs.flutter.dev

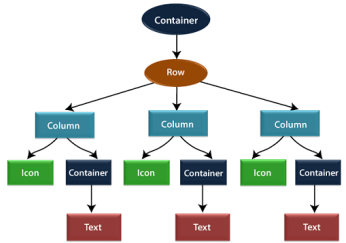
- Android und iOS dem imperativen Stil. Bei **Android** wird zu Interaktion mit dem UI mit Referenzen auf die Instanzen der UI-Elemente gearbeitet. Bei Flutter wird die UI jedes Mal neu erzeugt und die Änderungen angezeigt. Dies entspricht grob dem Vorgehen bei **React, Solid** und weiteren **Webframeworks**.



Imperativ, Android

Widgets

- In Flutter werden alle Views und UI Elemente durch sogenannte Widgets repräsentiert.
- Es werden sehr viele Möglichkeiten angeboten, Widgets zu nutzen.
- **Widgets** sind meist verschachtelt. Die etwas sperrige UI-Definition bei Android/iOS entfällt dadurch, was aber in mehr Zeilen Code in den Controllern resultiert.
- Hierdurch werden Teile von Screens wie hier die **TabBar** aus kleinen Widgets in einem Container zusammengefasst. Hierzu gibt es eine „**Table Logik“** mit **Rows** und **Columnes**.



Bildquelle: https://www.javatpoint.com/flutter-layouts

- Eine Row und Column kann ein child oder mehrere Children haben. Hier zeigt sich, dass Dart solche verschachtelten Strukture perfekt beherrscht.

Widgets

- Alle **UI Elemente** in Flutter bestehen aus Widgets bzw. sind Widgets.
- Das **Composing** von Oberflächen gestaltet sich durch Zusammenfügen und Entwickeln von Widgets.
- Diese Vorgehensweise kann ein wenig an die Webentwicklung mit React/Vue/Solid erinnern, wo ebenfalls Logik und UI-Struktur in einer Datei zusammengefasst sind.
- Flutter unterscheidet **zwei Arten von Widgets:**

Stateless Widgets

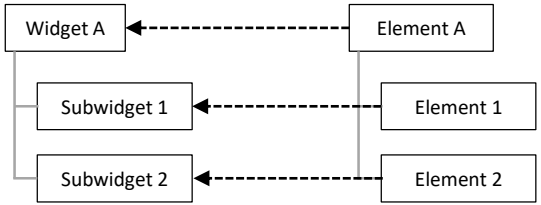
Rein darstellende Widgets, die ihren Inhalt nicht ändern. Beispiel: Textfelder, Cards ohne Interaktion

Stateful Widgets

Widgets, die Zustände haben, die sich ändern können (vgl. States bei React), **z.B. Editfields**

- Widgets sind also **Teile eines UIs**, die wiederum Widgets enthalten können.
- Ein **Element** instanziiert ein Widget, mounted dieses auf dem Screen und zeigt es an.
- Der **Element-Tree** repräsentiert, was derzeit auf dem Screen angezeigt wird. Dieser enthält verschachtelte Widgets.
- Jede Widget Klasse hat eine dazugehörige Elementklasse und eine Methode zur Instanziierung.
- Stateless Wigets erzeugen ein Stateless Element.
- Die Stateless Widgets Klassen haben eine createElement Methode, die aufgerufen wird, wenn das Widget auf dem Screen gemounted wird. Die Referenz des instaziierten Widgets wird dann zum Element-Tree hinzugefügt.
- Kind-Elemente werden nach der Hierarchie eingefügt **und deren build-Methode aufgerufen**.

Widgets (Blueprints) aus denen Elemente hervorgehen



Elements werden jetzt auf dem Screen angezeigt.

Stateless Widgets

Beispielkomponente eines Stateless Widgets

- Das Widget ist statisch und die Elemente ändern sich nicht.
- Die 2 Textfields sind statisch.
- Die **App** wird in der main() – Funktion mit **runApp(…) gestartet.**
- Es gibt verschiedene Design Konzepte: **Scaffold** steht für **Material**, **Cupertino** für **iOS**. Weitere Designs für Desktop und Mobile stehen zur Verfügung.
- Scaffold ist ein Widget, das den ganzen Screen ausfüllt und grundlegende Elemente wie AppBar

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

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
            appBar: AppBar(
              title: const Text('Stateless Widget'),
            ),
            body: const Center(
                child: Column(
                    mainAxisAlignment: MainAxisAlignment.center,
                    children: [
                  Text('Mobile Anwendungen'),
                  Text('Cross Plattform mit Flutter')
                ]))));
  }
}
```

Widget build function

Scaffold stellt ein „Gerüst“ her, das den ganzen Screen einnimmt.

zentriert

Widget enthält weitere Children

Codesample unter 1_StatelessWidget.zip

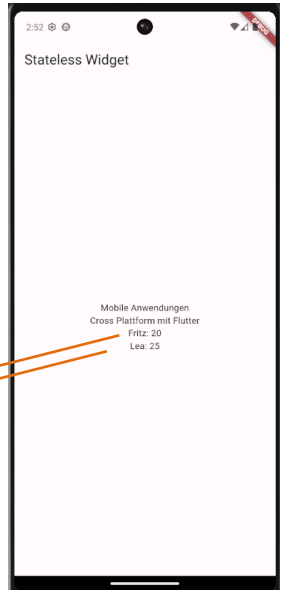
StatelessWidgets

- **Stateless Widgets** beinhalten interne sich nicht Eigenschaften wie beispielsweise Texte oder Buttons, die zwar eine Methode aufrufen, aber die UI nicht ändern.
- Um trotz der Unveränderlichkeit flexible Komponenten zu erhalten, können dem Constructor Parameter übergeben werden, die zur „Konfiguration“ des Stateless Widgets dienen und es so flexibel machen.

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

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
            appBar: AppBar(
              title: const Text('Stateless Widget'),
            ),
            body: const Center(
                child: Column(
                    mainAxisAlignment: MainAxisAlignment.center,
                    children: [
                  Text('Mobile Anwendungen'),
                  Text('Cross Plattform mit Flutter'),
                  PersonAge(name: 'Fritz', age: 20),
                  PersonAge(name: 'Lea', age: 25)
                ])));
  }
}

class PersonAge extends StatelessWidget {
  final String name;
  final int age;
  const PersonAge({required this.name, required this.age});

  @override
  Widget build(BuildContext context) {
    return Text('$name: $age');
  }
}
```

Auf diese Weise können UI komfortabel komponiert werden.

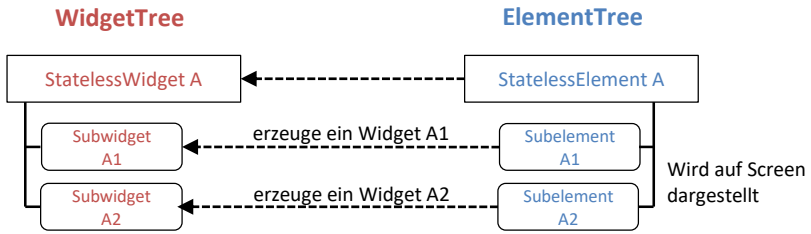
Die final Variable ist nach dem Setzen im Constructor nicht mehr veränderbar



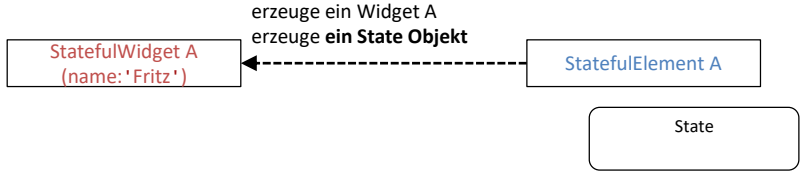
Codesample unter 2_StatelessWidget_Var.dart

StatefulWidgets

- **Stateless Widgets** beinhalten interne sich ändernde Zustände wie beispielsweise Texte in einem Textfeld oder die änderbare Farbe/Text eines Buttons.
- Um die **Zustände** (States) zu verwalten und zu speichern werden i**n Dart State-Klassen** genutzt. Dieses Konzept ist in etwa vergleichbar mit states in React oder Signals in Solid.
- Bei **Stateless Widgets** fordert Element A den Bau des Widget A an, fügt es dem ElementTree hinzu, wodurch das Element dargestellt wird.



- Bei Stateful Widgets muss ein Mechanismus enthalten sein, der den Screen aktualisieren kann.



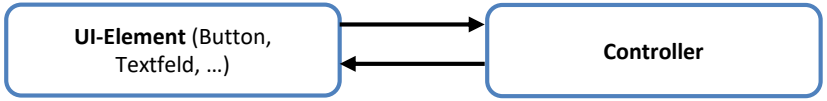
StatefulWidgets

- **Stateful Widgets beinhalten UI-Elemente**, deren Zustände sich ändern können.
- Sie können in **Stateless Widgets** eingebunden werden.
- Flutter optimiert so das Rendern, da nur Komponenten mit **Stateful Widgets** während der Anzeige aktualisiert werden.

- Bei GUI Systemen kann beim Erlernen immer folgende zwei Punkte geklärt werden.
  - Wie kommen **Nutzerinteraktionen zum Controller** und können verarbeitet werden?
  - Wie kann der **Controller UI-Elemente manipulieren** und Inhalte anzeigen bzw. ändern.

- Bei Flutter ist dies in **Stateful Widgets** möglich:

Callback-Methode, die dem UI-Element angehängt wird



Durch **Veränderung einer State Variable** wird erkannt, dass die Komponente neu gerendert werden muss.

- Um States bereitzustellen, wird in einer **Stateful** Klasse eine Methode **createState** genutzt, die eine Klasse, die von State abgeleitet ist, bereitstellt.
- Diese Struktur ist typisch für Flutter und wird kann auch mehrere Zustände erhalten.

- Die hohe Verschachtelung bietet dabei ein mächtiges Tool bei der Komposition von Screens. Insgesamt ist der Aufbau recht klar.

- *Anmerkung TW: Verglichen mit Webfrontends erscheint die Einführung der zusätzlichen Klasse etwas sperrig. Man gewöhnt sich aber schnell daran. Wichtig ist auch, dass Flutter eine enorme Anzahl an Widgets fertig anbietet, die in UI-Webframeworks aufwendig gebaut und in Android und iOS ebenfalls mit viel Zeitaufwand integriert werden müssen. „Irgendwas ist schließlich immer ☺“*

```dart
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Stateful Widget Example'),
        ),
        body: Center(
          child: ToggleTextWidget(),
        ),
      ),
    );
  }
}

class ToggleTextWidget extends StatefulWidget {
  @override
  _ToggleTextWidgetState createState() => _ToggleTextWidgetState();
}

class _ToggleTextWidgetState extends State<ToggleTextWidget> {
  String _displayText = 'Mobile';

  void _toggleText() {
    setState(() {
      _displayText = _displayText == 'Mobile' ? 'Anwendungen' : 'Mobile';
    });
  }

  @override
  Widget build(BuildContext context) {
    return Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        Text(
          _displayText,
          style: TextStyle(fontSize: 40, fontWeight: FontWeight.bold),
        ),
        SizedBox(height: 40),
        ElevatedButton(
          onPressed: _toggleText,
          child: Text('Toggle Text'),
          style: ElevatedButton.styleFrom(
            padding: EdgeInsets.symmetric(horizontal: 15, vertical: 10),
            textStyle: TextStyle(fontSize: 30),
          ),
        ),
      ],
    );
  }
}
```

Scaffold stellt ein „Gerüst“ her, das den ganzen Screen einnimmt.

Private Variablen werden in Dart durch ein Underscore _ definiert

Methode wird beim Klicken ausgelöst

Column ordnet UI-Elemente untereinander an

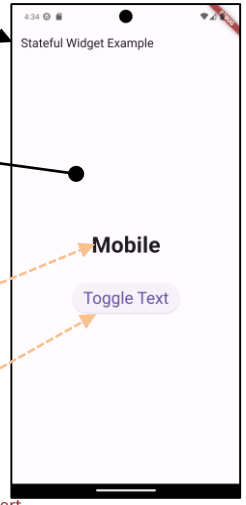
zentriert die Ansicht.

Children enthält die UI-Elemente, Die untereinander angeordnet werden. Hier ein Textfeld, dann ein Abstand, dann der Button

Dynamischer Inhalt hier Variable _dynamicText. Wenn diese sich in der State Klasse ändert, wird neu gerendert

Text enthält den Text des Buttons

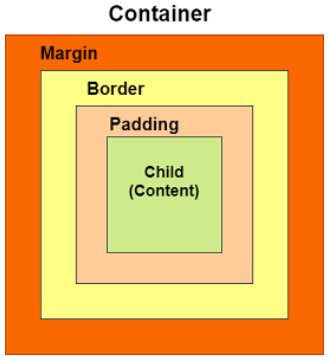
Mit style können die Eigenschaften des Buttons festgelegt werden.



Containers, Rows, Columns

Container

- Jeder Container kann Widgets enthalten und einen Margin, Border, Padding haben. Dieses Konzept ist quasi wie bei CSS umgesetzt.



https://www.javatpoint.com/flutter-container

- Jeder Container kann verschiedene Eigenschaften annehmen wie Farbe, Aligning und child/children aufnehmen.

Eigenschaften

```
Container({Key key,
           AlignmentGeometry alignment,
           EdgeInsetsGeometry padding,
           Color color,
           double width,
           double height,
           Decoration decoration,
           Decoration foregroundDecoration,
           BoxConstraints constraints,
           Widget child,
           Clip clipBehavior: Clip.none
});
```

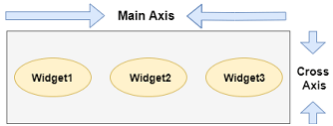
Beispiel

```
Container(
    color: Colors.green,
    child: Text("TextWidget",
               style: TextStyle(fontSize: 25)),
    width: 200.0,
    height: 100.0,
    padding: EdgeInsets.all(35),
    margin: EdgeInsets.all(20),
    color: Colors.green
)
```

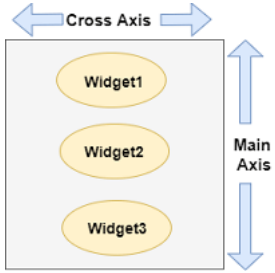
Rows & Colums

- Jeder Container kann Widgets enthalten und einen Margin, Border, Padding haben. Dieses Konzept ist ähnlich wie bei CSS umgesetzt.

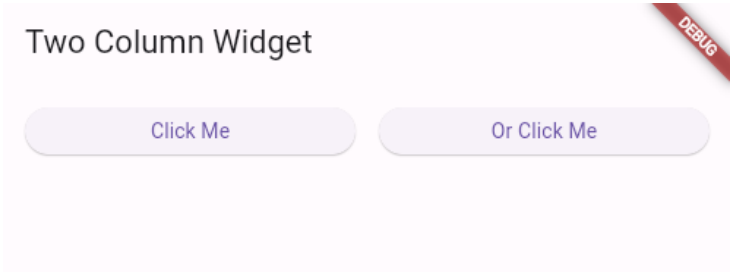
- Widgets können in Rows und Colums und verschachtelt angeordnet werden.



https://www.javatpoint.com/flutter-container



- Hier ein Bespiel für ein Widget mit 2 Columns. Siehe RowsColumns



EditFields

- Die **Eingabe von** Text, sowie **Fokushandling** (also welches Textfeld die Eingabe der Tastatur erhält) ist in Flutter erfreulich praktikabel gelöst.
- Im folgenden Beispiel wird ein Text eingegeben. Wenn dieser das **Wort hallo enthält**, färbt sich der Text darunter grün.

```
...

class TextWidget extends StatefulWidget {
  @override
  _TextWidgetState createState() => _TextWidgetState();
}

class _TextWidgetState extends State<TextWidget> {
  String _displayText = '';

  @override
  Widget build(BuildContext context) {
    Color textColor = _displayText.toLowerCase().contains('hallo')
        ? Colors.green
        : Colors.red;
    return Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: <Widget>[
        Padding(                                    Ob
            padding: const EdgeInsets.all(50),
            child: TextField(
              onChanged: (value) {
                setState(() {
                  _displayText = value;
                });
              },
              decoration: InputDecoration(
                labelText: 'Text eingeben',
              ),
            )),
        SizedBox(height: 50),
        Text(
          _displayText,
          style: TextStyle(fontSize: 20, color: textColor),
        ),
      ],
    );
  }
}
```

Beim neu Rendern wird textColor ebenfalls erneuert, da die build Methode aufgerufen wird

Padding um das Textfeld mit 50 px

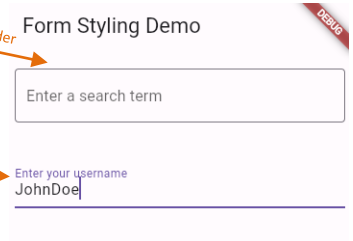
Setzen des states _displaytext

Die Farbe wird beim neu Rendern gesetzt, da _displayText geändert wurde.

- Textfelder können unterschiedlich gestaltet werden, um optimale User Experience und User Interfaces zu kreieren.
- **Gestaltungen wie Rahmen können mit Decorators gestaltet werden:**
  Beispiel:

```
child: TextField(
     decoration: InputDecoration(
       border: OutlineInputBorder(),
       hintText: 'Bitte etwas eingeben...',
     ),
),
```

OutlineInputBorder



- Flutter kennt neben einem TextField auf TextFormField

```
child: TextFormField(
   decoration: const InputDecoration(
     border: UnderlineInputBorder(),
     labelText: 'Enter your username',
   ),
 ),
```

TextFormFields können gut zu Formularen zusammengefügt werden.

- Wenn ein TextField in Flutter **den „Focus“** hat, so gehen die Tastatureingaben an dieses Textfeld. Durch Umschalten des Focus auf ein anderes Textfeld kann die Eingabe in diesem Textfeld erfolgen. Dies kann programmatisch oder automatisch beim Erscheinen erfolgen.

  Beispiel, siehe: https://docs.flutter.dev/cookbook/forms/focus

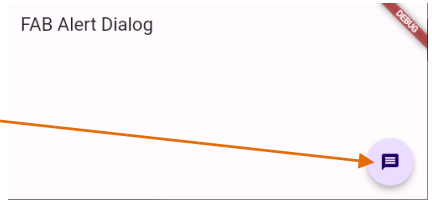
- Die Validierung von Eingaben kann mit validators erfolgen:

  https://docs.flutter.dev/cookbook/forms/validation

Floating Action Buttons, Snackbars

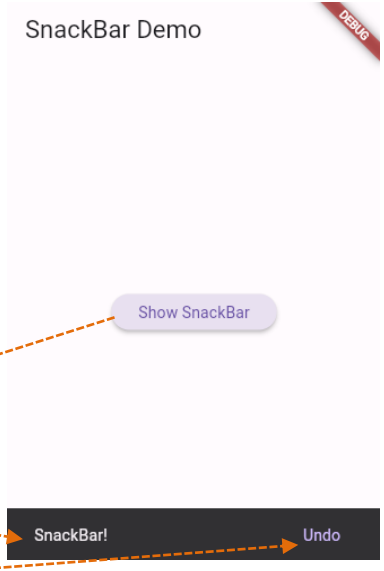
- Floating Action Buttons sind in Android Apps verbreitet, bei iOS jedoch nicht. Daher eignen sie sich für Android und Webapps.
- Für Schnellzugriffe der wichtigsten Funktion sind diese gut geeignet, da sie in einer für den Daumen gut erreichbaren Zone liegen.
- Es stehen einige gängige Icons bereit.

```
....
class MyWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Stack(
      children: [
        Positioned(
          bottom: 16.0,
          right: 16.0,
          child: FloatingActionButton(
            onPressed: () {
              _showAlertDialog(context);
            },
            child: Icon(Icons.message),
            shape: CircleBorder(),
          ),
        ),
      ],
    );
  }

----
```



- Snackbars dienen dazu, kurze Infos an den Nutzer zu geben.
- In die Snackbar können auch Buttons eingefügt werden.

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

void main() => runApp(const SnackBarDemo());

class SnackBarDemo extends StatelessWidget {
  const SnackBarDemo({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'SnackBar Demo',
      home: Scaffold(
        appBar: AppBar(
          title: const Text('SnackBar Demo'),
        ),
        body: const SnackBarPage(),
      ),
    );
  }
}

class SnackBarPage extends StatelessWidget {
  const SnackBarPage({super.key});

  @override
  Widget build(BuildContext context) {
    return Center(
      child: ElevatedButton(
        onPressed: () {
          final snackBar = SnackBar(
            content: const Text('Snackbar'),
            action: SnackBarAction(
              label: 'Undo',
              onPressed: () {
                // Some code to undo the change.
              },
            ),
          );

          // Find the ScaffoldMessenger in the widget tree
          // and use it to show a SnackBar.
          ScaffoldMessenger.of(context).showSnackBar(snackBar);
        },
        child: const Text('Show SnackBar'),
      ),
    );
  }
}
```



Images und Mediendaten

- Analog zu Android und iOS können auch bei Flutter Assets, Icons, Bilder in verschiedenen Auflösungen abgelegt werden.
- Um Assets nutzen zu können, müssen diese in einem Unterordner des Projektverzeichnisses abgelegt werden. Hierbei wird meist ein Ordner /assets angelegt.
- Empfehlenswert sind Unterordner /images /data /fonts
- Die **Pfade**, wo sich Assets befinden, müssen in der Datei **pubspec.yaml eingetragen werden.** Das Flutter Framework baut daraus eine **AssetBundle**, auf das in der App **zugegriffen werden** kann.
- Bei Bildern und Icons Dabei können auch **verschiedene Auflösungen in verschiedenen Ordnern mit demselben Dateinamen** abgespeichert werden. Dazu werden Unterordner angelegt, die für verschiedene Auflösungen stehen:

```
.../my_icon.png (mdpi baseline)
.../1.5x/my_icon.png (hdpi)
.../2.0x/my_icon.png (xhdpi)
.../3.0x/my_icon.png (xxhdpi)
.../4.0x/my_icon.png (xxxhdpi)
```

- Für Icons werden von Vektorgrafikprogrammen wie Adobe Plugins angeboten, die die verschiedenen Auflösungen exportieren.
- Die **Verzeichnisse der Assets** müssen in der Datei **pubspec.yaml** eingetragen werden. Wenn keine Dateinamen hinter dem Pfad angegeben werden, dann werden alle Bilder importiert:

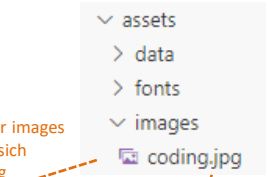
```
assets:
  - assets/images/
  - asstes/data/
```

- Auf die Assets kann mit Image Asset zugegriffen werden
- In iOS werden Assets über mainBundle verwaltet, was Flutter übernimmt.

- Die App-Icons müssen in den von Flutter erstellten Projekten im Ordner iOS und Android aktualisiert werden. Siehe https://docs.flutter.dev/ui/assets/assets-and-images
- Für Android ist dies der Ordner /res/mipmap….
- Für iOS der Ordner AppIcon.appiconset

- Der Launchscreen kann ebenfalls angepasst werden. Siehe Link.

```dart
void main() {
  runApp(MaterialApp(
    home: CenteredImage(),
  ));
}

class CenteredImage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Centered Image'),
      ),
      body: Center(
          child: Column(children: <Widget>[
        Image.asset(
          'assets/images/coding.jpg',
          fit: BoxFit.cover,
        ),
        const Text("An image of a coding person.",
            style: TextStyle(fontSize: 20.0)),
        SizedBox(height: 20),
        Image.network('https://picsum.photos/250?image=9'),
        const Text("An image loaded from internet.",
            style: TextStyle(fontSize: 20.0))
      ])),
    );
  }
}
```



Im Ordner images befindet sich coding.jpg

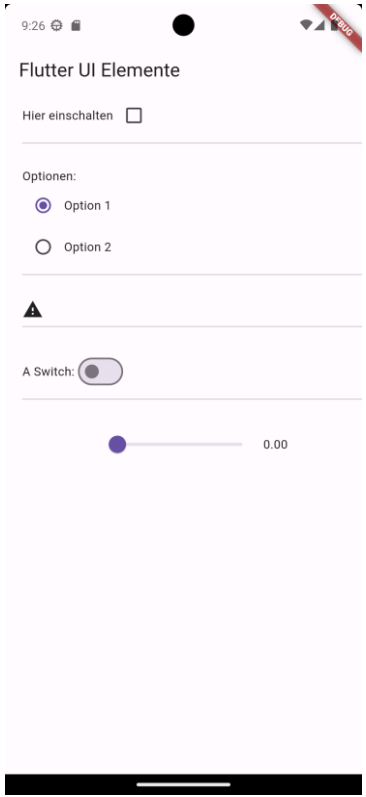
Breite ausfüllen

Bilder/Videos aus Links

- Bilder aus einem link können über Image.network(…) geladen werden.
- Fade ins während dem Laden können ebenfalls realsiert werden. Siehe hierzu: https://docs.flutter.dev/cookbook/images/fading-in-images
- Zum Abspielen von Videos müssen Permissions gesetzt werden: https://docs.flutter.dev/cookbook/plugins/play-video

UI Elemente: Checkbox, Radio Buttons, Icons, Slider, Switch, Cards, Rating, Alerts

- Flutter bietet die wichtigsten UI-Elemente, die in Mobile Apps häufig genutzt werden an.
- Im Beispiel 5_Widget_UI_Elemente.dart ist der Quellcode für die folgende UI gegeben.
- Öffnen sie die Datei und kommentieren sie die Parts.

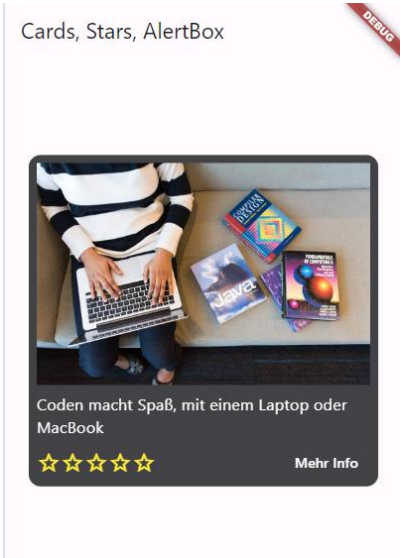


Workshop: Schreiben sie als Kommentar in ihr File:

- Wie werden die Snackbars geöffnet?
- Wie wurde die Linksbündigkeit der erzielt?
- Wie wird die Default Einstellung, dass Option 1 beim Start selektiert ist erreicht?
- Wie wird der Klick auf das Icon realisiert?
- Wie werden die Divider eingesetzt?
- Wie wird der Slider und die Ausgabe des Werts umgesetzt?

- Checkbox
- Radio
- Buttons
- clickable Icons
- Switch
- Slider

- Weitere etwas komplexere UI-Elemente sind Cards, Sterne zum Rating, AlertBox.
- Im Beispiel 6_Widget_Card.dart ist der Quellcode für die folgende UI gegeben.



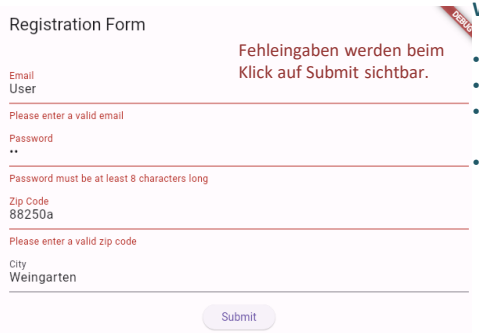
Workshop: Schreiben sie als Kommentar in ihr File:

- Wie werden die Cards definiert?
- Wie funktioniert das Rating mit Stars?
- Beachten Sie die Definition der AlertBox, die mit der Methode showAlertBox(…) geöffnet wird.
- Wie wird die Hintergrundfarbe eingestellt?

- Card
- Ratings
- AlertBox

UI Elemente: Forms, Validation, Date / Time Picker

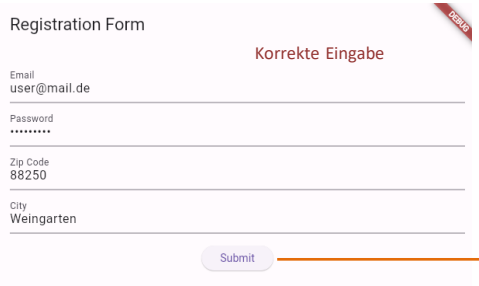
- Forms sollten prinzipiell bei Apps sparsam eingesetzt werden, da Nutzer Eingaben meist nicht mögen.
- Im Beispiel **7_Widget_Registration_Form.dart** ist ein Beispiel für ein Form mit Validation gegeben.
- Das Feld validator ist hierfür sehr nützlich. Durch RegEx können Eingaben in gewohnter Weise geprüft werden.
- Hier ist ersichtlich, dass der Aufwand nicht sehr hoch ist.

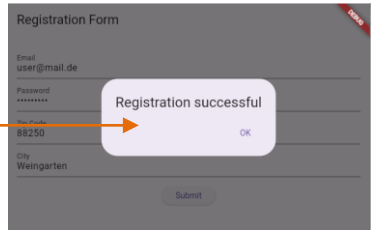


Workshop: Schreiben sie als Kommentar in ihr File:

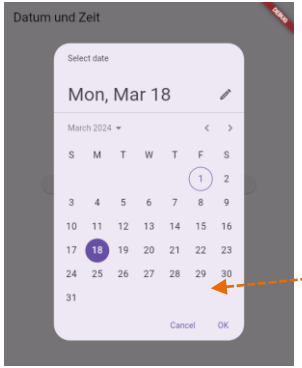
- Wie wird die E-Mail Validation umgesetzt?
- Wie werden die Punkte bei Password eingestellt?
- Wie wird der Submit-Button gehandelt?

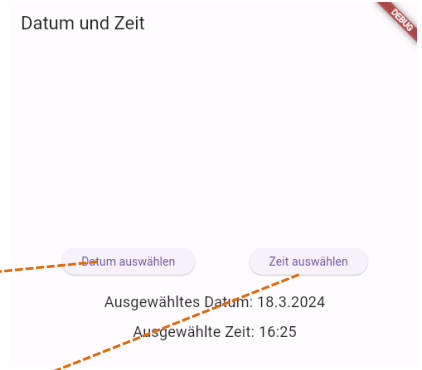
- Wie könnte eine Validierung erfolgen, dass der Ort mehr als 2 Buchstaben haben muss?

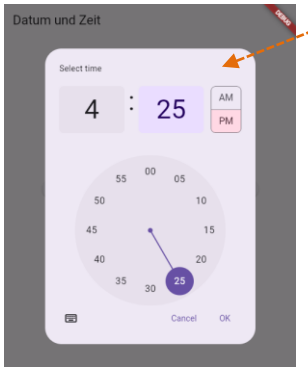




- Im Beispiel **8_Widget_DateTime.dart** sind Time- und Datepicker, wie sie in Android üblich sind enthalten
- *Diese können je nach Device anders aussehen!*







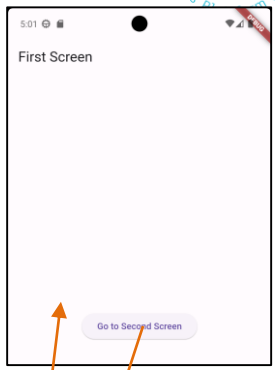
Navigation

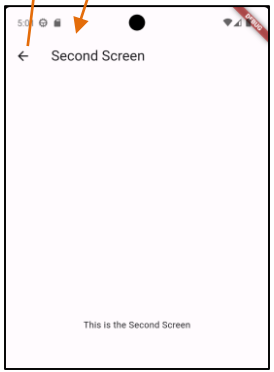
- Für einfache Stacks kann ein Navigator genutzt werden.
- Bei Android ist dies mit expliziten Intents, bei iOS mit Segues vergleichbar.
- Beim Schließen eines Views wird der darunterliegende View wieder sichtbar.

- Mit der **builder** Funktion von MaterialPageRoute wird der **second** Screen erstellt und angezeigt. Context ist die Quelle, also der Screen, von dem aus der nächste Screen erstellt wird.

- Mit **der Navigator.pop() Funktion** wird der Screen **geschlossen** (entspricht bei Android this.finish())

```dart
void main() {
  runApp(MaterialApp(
    home: FirstScreen(),
  ));
}

class FirstScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('First Screen'),
      ),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.push(
              context,
              MaterialPageRoute(builder: (context) => SecondScreen()),
            );
          },
          child: const Text('Go to Second Screen'),
        ),
      ),
    );
  }
}

class SecondScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Second Screen'),
        leading: IconButton(
          icon: Icon(Icons.arrow_back),
          onPressed: () {
            Navigator.pop(context);
          },
        ),
      ),
      body: const Center(
        child: Text('This is the Second Screen'),
      ),
    );
  }
}
```





Router

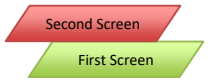
- Flutter unterstützt ein **mächtiges Routing** über „Links“.
- Dies ist vergleichbar mit einem **Router** wie ihn z.B. React einsetzt.
- **Router und Navigators** können **gemeinsam verwendet** werden. Über einen Link kann ein View angezeigt werden, der mittels Navigator überlagert wird.
- Das Beispiel aus https://docs.flutter.dev/cookbook/navigation/named-routes zeigt die Vorgehensweise.

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

void main() {
  runApp(
    MaterialApp(
      title: 'Named Routes Demo',
      // Start the app with the "/" named route. In this case, the app starts
      // on the FirstScreen widget.
      initialRoute: '/',
      routes: {
        '/': (context) => const FirstScreen(),
        '/second': (context) => const SecondScreen(),
      },
    ),
  );
}
```

Route für FirstView

Route für SecondView

Wenn die Route /second angewählt wird,
dann wird der Screen gebaut und



Route: /second

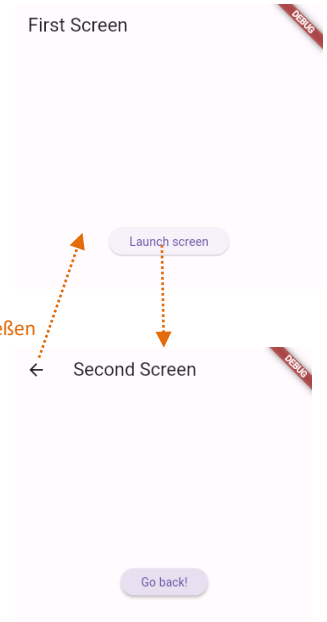
Route: /

```dart
class FirstScreen extends StatelessWidget {
  const FirstScreen({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('First Screen'),
      ),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.pushNamed(context, '/second');
          },
          child: const Text('Launch screen'),
        ),
      ),
    );
  }
}

class SecondScreen extends StatelessWidget {
  const SecondScreen({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Second Screen'),
      ),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            Navigator.pop(context);
          },
          child: const Text('Go back!'),
        ),
      ),
    );
  }
}
```

Beim Klick auf den Button ruft der Navigator die Route /second auf.
Bitte den Unterschied zur Nutzung ohne Router nur mit Navigator beachten. Vgl. letzte Seite :

```dart
onPressed: () {
  Navigator.push(
    context,
    MaterialPageRoute(builder: (context) => SecondScreen()),
  );
},
```

pushNamed „weist den Router an“.

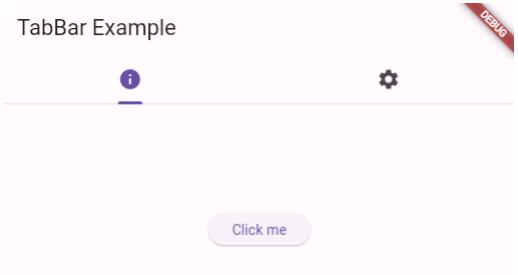


Screen vom Stack nehmen, schließen

Schließt den View wieder

TabBar Apps

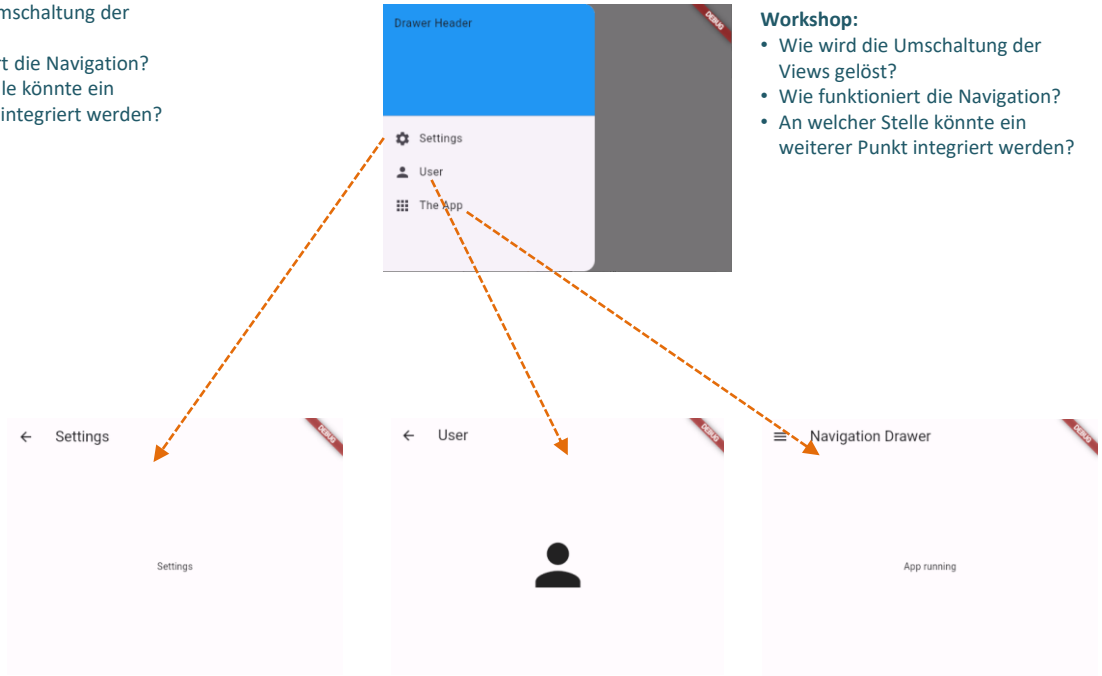
- Tab Bar Apps sind in iOS schon sehr lange bekannt. In Android war diese Darstellung seltener zu finden.
- Top Bar App in **04_TabBArApp_Top.dart**



Workshop:
- Wie wird die Umschaltung der Views gelöst?
- Wie funktioniert die Navigation?
- An welcher Stelle könnte ein weiterer Punkt integriert werden?

Navigation Drawer

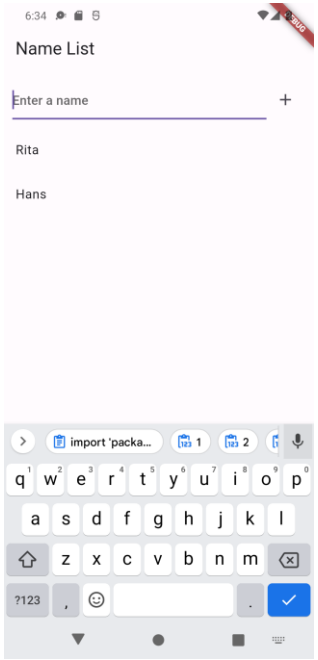
- Navigation Drawer / Burger Menus bieten eine Navigation an, die häufig in Android zu finden ist. Im Grunde ist diese Art der Navigation eher für die Bedienung im Web nützlich. Für Mobile ist das Burger Menu weit schlecht erreichbar für den Daumen.
- Navigation Drawer App in **05_NavigationDrawer.dart**



Workshop:
- Wie wird die Umschaltung der Views gelöst?
- Wie funktioniert die Navigation?
- An welcher Stelle könnte ein weiterer Punkt integriert werden?

Listen

- Listen sind in Flutter um Längen einfacher zu realisieren als in Android oder iOS.
- 01_List.dart zeigt ein Beispiel. Im Textfeld kann ein Name eingegeben werden und mit dem +-Icon hinzugefügt werden. Durch Swipen kann der Eintrag gelöscht werden.



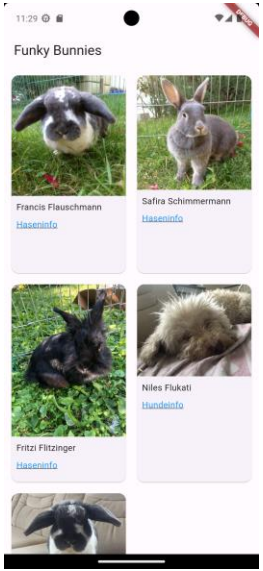
Workshop:

- Wie werden die Listeneinträge gespeichert?
- Wie wird das Einfügen eines Namen realisiert?
- Wie wird die Swipe Funktion ermöglicht?
- Wie werden Einträge gelöscht.

1_List.dart

Card Views

- Card Views werden gerne genutzt, wenn Cards in Kacheln angezeigt werden sollen.
- 01_List.dart zeigt ein Beispiel. Im Textfeld kann ein Name eingegeben werden und mit dem +-Icon hinzugefügt werden. Durch Swipen kann der Eintrag gelöscht werden.



Workshop:

- Wie werden die einzelnen Datensätze gespeichert?
- Wie wird das Grid realisiert?
- Wie werden dir Cards erzeugt?
- Wie wird der Text clickbar?
- Wie wird die Snackbar geöffnet?

- Erweitern sie die Grids so, dass eine 5-Sterne Bewertung eingefügt wird.

2_GridFunkyPets.dart

Persistente Speicherung

- In Android und iOS stehen verschiedene Möglichkeiten zur Verfügung, um **Daten persistent speichern** zu können.
- **Android**: Hier werden Daten üblicherweise in **SQLite Datenbanken** persistiert. Eine bekannte ORM ist Room Library. Alternativ können einfache Key-Value-Paare als „**sharedPreferences**“ gespeichert werden. Dateien können im **internen Speicher** und auf der **SD Karte** angelegt werden.
- **iOS**: In iOS wird **Core Data** verwendet, um Daten zu speichern. Core Data ist ein ORM für **SQLite**. *Anmerkung TW: Mein Eindruck und der vieler Entwickler ist, dass dieses Framework äußerst sperrig ist. Die Konzeption halte ich für enorm umständlich und zudem unklar im Aufbau.* Auch bei iOS besteht die Möglichkeit, **Key-Value Paare als Settings** zu speichern. Das Anlegen von **Dateien** in der **Sandbox** der App sollte auch möglich sein.
- Auch im Browser ist ein **Lightweight SQLite** integriert. Zudem können im **LocalStorage Key-Value Paare** gespeichert werden.
- Der gemeinsame Nenner, um Metadaten in allen drei Umgebungen zu speichern ist also **SQLite und für sehr kleine Datenmengen wie Settings etc. die Key-Value-Paare.**
- Flutter bietet für beide Möglichkeiten eine Integration an.

- Um SQLite nutzen zu können, sollte zuvor das Konzept der Futures eingeführt werden. Futures sind vergleichbar mit Promises in Javascript. Somit wird asynchrone Programmierung wie in Javascript ermöglicht.
- SQLite speichert die Daten in einer Datendatei, die in der Sandbox der App gespeichert wird.

Pfad, wo sich die SQLite Datei befindet. Es können auch mehrere SQLite Datenbanken leicht erzeugt und genutzt werden.

Mit Hilfe von Futures wird in diesem Beispiel eine Datenbankoperation ausgeführt. Eine Tabelle wird erzeugt mit 4 Feldern:
- id als Integer
- name als TEXT
- Zeitstempel für Erzeugung und Update

```
@override
  void initState() {
    super.initState();
    _initDatabase();
  }

  Future<void> _initDatabase() async {
    final documentsDirectory = await getApplicationDocumentsDirectory();
    final path = join(documentsDirectory.path, 'names_database.db');
    _database = await openDatabase(path, version: 1, onCreate: (db, version) {
      return db.execute(
        'CREATE TABLE Person(id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT,
                            createdAt TEXT, updatedAt TEXT)',
      );
    });
    _loadNames();
  }
```

SQLite

- Die SQLite Integration in Flutter ist deutlich einfacher als bei Android und iOS.
- Notwendig sind folgende Pakete in **pubspec.yaml.**

```
dependencies:
  flutter:
    sdk: flutter
  sqflite: ^2.3.3
  path: ^1.9.0
  path_provider: ^2.1.3
```

Siehe: https://pub.dev/packages/sqflite

- Um mit einer SQLite Datenbank zu interagieren, werden hauptsächlich 4 Operationen nötig sein: **CRUD: Create, Read, Update, Delete**.

- **Daten lesen (Read):**

```
late Database _database;

...

Future<void> _loadNames() async {
    final List<Map<String, dynamic>> names = await _database.query('Person');
    setState(() {
      _names = names;
    });
  }
```

Future _loadName Methode lädt alle Persons

- **Daten hinzufügen (Create), Änderungen mit der Methoden .update**

```
Future<void> _addName(String name) async {
    final String now = DateTime.now().toIso8601String();
    await _database.insert(
      'Person',
      {'name': name, 'createdAt': now, 'updatedAt': now},
    );
    _loadNames();
    _nameController.clear();
  }
```

Daten einfügen mit insert

Danach Daten neu laden (da diese nun beim Erzeugen eine ID bekommen haben). Die Tabelle in der Beispiel-App wird aktualisiert.

- **Daten löschen (Delete):**

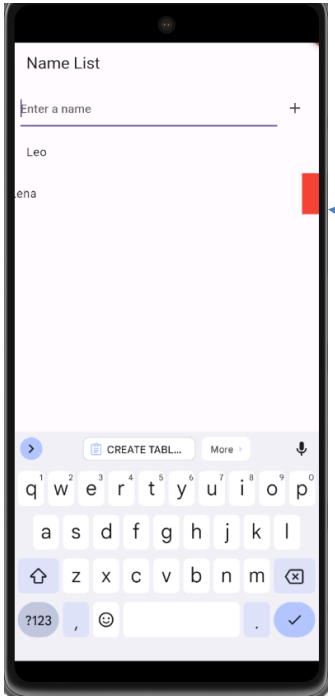
```
Future<void> _deleteName(int id) async {
    await _database.delete(
      'Person',
      where: 'id = ?',
      whereArgs: [id],
    );
    _loadNames();
  }
```

Nach dem Löschen, Daten neu laden.

1_SQLite_List.dart

SQLite/Dismissable

- Beispiel-App (mit SQL Operationen der letzten Seite)
- Architektonisch empfiehlt es sich, die SQL-Operationen in ein separates Dart-File auszulagern. So quasi können Services implementiert werden, die von mehreren Flutter Widgets genutzt werden können.



Nach dem Löschen, Daten neu laden.

Löschen durch Swipe mit dem Widget Dismissable

Workshop:

- Erweitern sie die App um Vorname, Nachname und City.
- Stellen Sie das Ergebnis in der Liste mit Komma getrennt dar.

- Konzeption: Mit welchen UIs könnten die Einträge geändert werden.

```
@override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Name List'),
      ),
      body: Column(
        children: <Widget>[
          Padding(
            padding: const EdgeInsets.all(8.0),
            child: Row(
              children: [
                Expanded(
                  child: TextField(
                    controller: _nameController,
                    decoration: const InputDecoration(
                      hintText: 'Enter a name',
                    ),
                  ),
                ),
                IconButton(
                  icon: const Icon(Icons.add),
                  onPressed: () => _addName(_nameController.text),
                ),
              ],
            ),
          ),
          Expanded(
            child: ListView.builder(
              itemCount: _names.length,
              itemBuilder: (context, index) {
                final item = _names[index];
                return Dismissible(
                  key: Key(item['id'].toString()),
                  direction: DismissDirection.endToStart,
                  onDismissed: (direction) {
                    _deleteName(item['id']);
                  },
                  background: Container(color: Colors.red),
                  child: ListTile(
                    title: Text(item['name']),
                  ),
                );
              },
            ),
          ),
        ],
      ),
    );
  }
}
```

TextField

Name hinzufügen

Hinzufügen des Eintrags

Richtung Dismissable

Löschen des Eintrags

Farbe Dismissable

1_SQLite_List.dart

SharedPreferences

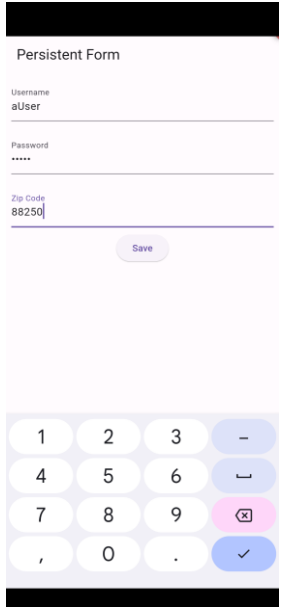
- Um Key-Value Paare in Android Apps zu nutzen, ist die Installation des folgenden Paketes notwendig.
- Im Terminal eingeben:

```
flutter pub add shared_preferences
```

- Die Bibliothek bietet den komfortablen Zugriff auf die SharedPreferences.

```dart
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Form with SharedPreferences',
      home: FormPage(),
    );
  }
}

class FormPage extends StatefulWidget {
  @override
  _FormPageState createState() => _FormPageState();
}

class _FormPageState extends State<FormPage> {
  final TextEditingController _usernameController = TextEditingController();
  final TextEditingController _passwordController = TextEditingController();
  final TextEditingController _zipController = TextEditingController();
  late SharedPreferences prefs;

  @override
  void initState() {
    super.initState();
    SharedPreferences.setMockInitialValues({});  Notwendig, siehe Quellcode
    loadPreferences();
  }

  Future<void> loadPreferences() async {
    prefs = await SharedPreferences.getInstance();  SharedPrefernces laden
    setState(() {
      _usernameController.text = prefs.getString('username') ?? '';
      _passwordController.text = prefs.getString('password') ?? '';
      _zipController.text = prefs.getString('zip') ?? '';
    });
  }

  bool get isSaveButtonEnabled {
    return _usernameController.text.length >= 2 &&
        _passwordController.text.length >= 2 &&
        _zipController.text.length >= 2;
  }
```

```dart
  Future<void> savePreferences() async {
    await prefs.setString('username', _usernameController.text);
    await prefs.setString('password', _passwordController.text);
    await prefs.setString('zip', _zipController.text);
  }                                                    SharedPrefernces speichern

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Persistent Form'),
      ),
      body: Column(
        children: <Widget>[
          Padding(
            padding: const EdgeInsets.all(8.0),
            child: TextField(
              controller: _usernameController,
              decoration: const InputDecoration(labelText: 'Username'),
              onChanged: (text) => setState(() {}),
            ),
          ),
          Padding(
            padding: const EdgeInsets.all(8.0),
            child: TextField(
              controller: _passwordController,
              obscureText: true,
              decoration: const InputDecoration(labelText: 'Password'),
              onChanged: (text) => setState(() {}),
            ),
          ),
          Padding(
            padding: const EdgeInsets.all(8.0),
            child: TextField(
              controller: _zipController,
              keyboardType: TextInputType.number,
              decoration: const InputDecoration(labelText: 'Zip Code'),
              onChanged: (text) => setState(() {}),
            ),
          ),
          ElevatedButton(
            onPressed: isSaveButtonEnabled ? savePreferences : null,
            child: const Text('Save'),
          ),
        ],
      ),
    );
  }
}
```

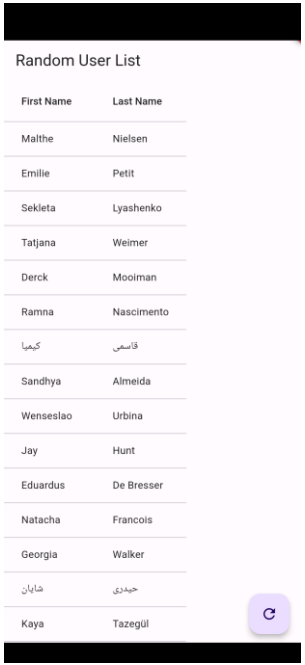


Anbindung von Webservices

- Zur Abfrage von REST-Calls kann das Dart Paket http genutzt werden.
- Im Terminal eingeben:

```
flutter pub add http
```

- Mehr Infos unter: https://pub.dev/packages/http
- Zur Darstellung der Tabelle wurde das Widget Datatables genutzt.
- Als API wird https://randomuser.me/api/?results=20 genutzt. Bitte ausprobieren.




```dart
class User {
  final String firstName;
  final String lastName;

  User({required this.firstName, required this.lastName});

  factory User.fromJson(Map<String, dynamic> json) {
    return User(
      firstName: json['first'],
      lastName: json['last'],
    );
  }
}
```

https://dart.dev/language/constructors

In Dart, a **factory constructor** is a special type of constructor that doesn't always create a new instance of its class. Instead, it can return an existing instance or a subtype, providing a more flexible way to instantiate classes compared to traditional constructors.
Key Features of Factory Constructors:
1. Instance Reuse: Factory constructors can return objects from a cache, or they can implement a singleton pattern where they return the same instance every time, which is useful for managing resources efficiently.
2. Condition-Based Instance Creation: They can return instances of different types based on input parameters or other conditions. This is handy when a function needs to decide which subclass to instantiate or when returning instances of an interface based on specific criteria.
3. Initialization Logic: They allow for more complex initialization logic before the object is returned to the caller.

- Für REST-Calls bietet es sich an, asynchron mit Futures zu arbeiten.
- Im Terminal eingeben:

```
flutter pub add http
```

- Mehr Infos unter: https://pub.dev/packages/http
- Zur Darstellung der Tabelle wurde das Widget Datatables genutzt.
- Als API wird https://randomuser.me/api/?results=20 genutzt. Bitte ausprobieren.

```dart
List<User> _users = [];

...

Future<void> _loadUsers() async {
    setState(() {
      _isLoading = true;
    });
    final response = await http.get(Uri.parse('https://randomuser.me/api/?results=20'));
    if (response.statusCode == 200) {
      final data = jsonDecode(response.body) as Map<String, dynamic>;
      final List<dynamic> users = data['results'];


      setState(() {
        _users = users.map((user) => User.fromJson(user['name'])).toList();
        _isLoading = false;
      });
    } else {
      // Handle the error; for simplicity, we're just setting _users to empty
      setState(() {
        _users = [];
        _isLoading = false;
      });
    }
}
```

JSON Decode wird zum Parsen genutzt.
Dies ist nicht so komfortabel wie in JS/TS

Get und URL

JSON Objekt mit dynamic List
https://www.geeksforgeeks.org/dynamic-initialization-of-list-in-dart/
Dies ermöglicht eine recht flexible Nutzung.

JSON Objekt mit dynamic List
https://www.geeksforgeeks.org/dynamic-initialization-of-list-in-dart/
Dies ermöglicht eine recht flexible Nutzung.

Workshop:
- Erweitern sie das Widget so, dass die Stadt in der Tabelle dargestellt wird.
- Stylen Sie die Tabelle so, dass der Last Name fett gedruckt wird.

Nutzung von Maps (OpenStreetMap)

- Zur Nutzung der Karte müssen zwei Pakete integriert werden:
- Im Terminal eingeben:

```
flutter pub add flutter_map
flutter pub add latlong2
```

- Flutter Map ist eine Bibliothek, um OpenStreetMap einzubinden.
- Hierbei können neben der Anzeige an einem bestimmten Längen- und Breitengrad einstellen.
- Flutter _Map kann Marker, Circle, Polygon und viele weitere Markierungen in der Karte vornehmen. Der Blickwinkel lässt sich komfortabel nutzen.
- Eine Dokumentation ist unter https://docs.fleaflet.dev/ zu finden.
- Hinweis: Viele Tutorials bauen auf ältere Versionen von Flutter_Map auf, deren API sich bei den neueren Versionen stark geändert hat. Daher eine aktuelle Doku nutzen: https://pub.dev/packages/flutter_map und https://docs.fleaflet.dev/

Satelliten / Gelände Views sind auch möglich

Das Widget zeigt eine Full Screen Map an, die einen Marker im T-Gebäude hat.
Der Circle zeigt ganz grob die Ausmaße des Campus.

```dart
class MapWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return FlutterMap(
      options: const MapOptions(
        initialCenter: LatLng(47.80887005657478, 9.644583696582753),
        initialZoom: 14,
      ),
      children: [
        TileLayer(
          urlTemplate: 'https://tile.openstreetmap.org/{z}/{x}/{y}.png',
          userAgentPackageName: 'com.example.app',
        ),
        const CircleLayer(
          circles: [
            CircleMarker(
              point: LatLng(47.813113914533126, 9.65390919445553),
              radius: 250,
              useRadiusInMeter: true,
              color: Color(0xAA777777),
            )
          ],
        ),
        const MarkerLayer(
          markers: [
            Marker(
                point: LatLng(47.81298947955553, 9.651690117073615),
                width: 80,
                height: 80,
                child: Icon(Icons.location_on, size: 50.0, color: Colors.red)),
          ],
        ),
      ],
    );
  }
}
```

Initiale Position beim Starten der Widget.

Workshop:

- Zeichnen Sie ein Polygon um das Hauptgebäude und verändern Sie den initialen Zoom.
- Wie kann der Marker clickbar gemacht werden?

4_OpenStreetMap.dart