Fluttern wir mit Dart zur nächsten App

Teil 4: Dart und Flutter – ein wunderbares Paar

In einem Tutorial zur Programmiersprache Dart gehört auch deren meiste Verwendung bei der Programmierung von Apps mit Flutter dazu. Flutter ist ein UI-Toolkit von Google, das es ermöglicht, plattformübergreifende Anwendungen mit nur einer Codebasis zu entwickeln. Durch die Kombination mit Dart bietet Flutter eine performante Umgebung für die Erstellung reaktionsschneller und optisch ansprechender Apps. In diesem Artikel werden die Grundlagen von Flutter erläutert. Wir zeigen, wie Flutter mit Dart zusammenarbeitet, welche Werkzeuge benötigt werden und wie eine App strukturiert wird. von Dr. Veikko Krypczyk

Flutter ist ein Open-Source-Framework, das 2017 von Google veröffentlicht wurde. Es bietet eine einheitliche Umgebung zur Entwicklung von nativ kompilierten Anwendungen für Android, iOS, Web, Windows, macOS und Linux. Wichtige Merkmale von Flutter sind:

* *Plattformübergreifende Entwicklung*: Ein einziger Code für verschiedene Plattformen.
* *Schnelle Entwicklung mit Hot Reload*: Änderungen am Code werden sofort in der App reflektiert.
* *Moderne, flexible User Interface-Gestaltung*: Dank eines widgetbasierten Systems.
* *Hohe Performance*: Verwendet eine eigene Rendering-Engine für native Geschwindigkeit.

In diesem vierten Teil der Artikelserie über die Programmiersprache Dart beschäftigen wir uns mit dem Cross Plattform-Framework Flutter (siehe Kasten „*Übersicht zur Artikelserie*“). Die Programmierung der Apps erfolgt in Dart.

Übersicht zur Artikelserie

Teil 1: Grundlagen und Syntax

Teil 2: Objektorientierte Programmierung

Teil 3: Asynchrone Programmierung

**Teil 4: Dart und Flutter**

Teil 5: Best Practices und fortgeschrittene Themen.

./.

Die Architektur (*Abbildung 1*) von Flutter ist so konzipiert, dass sie eine hohe Performance und eine konsistente Benutzererfahrung auf verschiedenen Plattformen ermöglicht. Im Zentrum steht die *Dart-Engine*, die den Flutter-Code in maschinennahe Sprache übersetzt. Dies geschieht durch Ahead-of-Time (AOT)-Kompilierung für Produktionsanwendungen, was eine nahezu native Ausführungsgeschwindigkeit ermöglicht. Während der Entwicklung nutzt Flutter hingegen die Just-in-Time (JIT)-Kompilierung, die in Kombination mit Hot Reload schnelle Änderungen am Code erlaubt, ohne die gesamte Anwendung neu starten zu müssen. Ein weiteres zentrales Element ist die *Skia-Rendering-Engine*, die die Benutzeroberfläche direkt auf den Bildschirm zeichnet, anstatt plattformspezifische User Interface-Komponenten zu verwenden. Dies unterscheidet Flutter von Frameworks wie React Native oder .NET MAUI, die sich auf native User Interface-Elemente stützen. Skia sorgt dafür, dass Animationen flüssig dargestellt werden und ermöglicht eine einheitliche Optik und Performance, unabhängig vom zugrunde liegenden Betriebssystem.

Das *Dart-Framework* bildet die oberste Schicht der Architektur und enthält eine umfangreiche Sammlung vorgefertigter Widgets für die Gestaltung moderner Benutzeroberflächen. Diese Widgets sind vollständig anpassbar und orientieren sich entweder an Material Design (Google) oder Cupertino (Apple), sodass eine native Nutzererfahrung auf Android- und iOS-Geräten gewährleistet wird. Zudem ist das Framework so strukturiert, dass Entwickler eigene Widgets einfach erstellen und wiederverwenden können, was die Wartung und Skalierbarkeit von Apps erleichtert.

Abbildung\_1.png

Abbildung 1: Flutter Architektur-Layers [1].

Welche Rolle spielt Dart bzw. welche Vorteile bringt der Einsatz dieser Programmiersprache bei der App-Entwicklung? Dart wird nativ kompiliert, wodurch Flutter-Apps oft schneller sind als andere Cross-Plattform-Technologien. Durch eine asynchrone Programmierung bietet Dart *Futures*, *Streams* und *async/await*, um reaktive Anwendungen zu ermöglichen. Es wird die objektorientierte Entwicklung mit Klassen, Vererbung, Interfaces und Mixins unterstützt, was eine saubere Code-Struktur ermöglicht. Die Effizienz der Entwicklung gilt insgesamt durch das bereitgestellte Tooling als gut und ist für produktionsreife Apps geeignet.

Einrichtung der Entwicklungsumgebung

Um mit Flutter zu entwickeln, muss die Entwicklungsumgebung entsprechend eingerichtet werden. Dies umfasst die Installation des *Flutter SDK*, das bereits das *Dart SDK* enthält, sowie die Einrichtung notwendiger Entwicklungswerkzeuge und Emulatoren. Flutter unterstützt die Betriebssysteme Windows, macOS, Linux und Web-Applikationen, sodass plattformübergreifende Entwicklung problemlos möglich ist. Zusätzlich stellt Flutter verschiedene *CLI-Tools* bereit, die Entwicklern helfen, die Installation zu überprüfen, Abhängigkeiten zu verwalten und Projekte effizient zu steuern.

Installation von Flutter unter Windows und macOS

Die Installation von Flutter unterscheidet sich je nach Betriebssystem, folgt jedoch einem ähnlichen Ablauf. Das *Flutter SDK* kann direkt von der offiziellen Webseite [2] heruntergeladen werden. Wir erläutern die Installation für Windows und macOS. Die Dokumentation beschreibt darüber hinaus das Vorgehen für Linux-Distributionen und Chrome OS.

**Windows:**

1. *Download*: Das Flutter-SDK als ZIP-Datei von der Website herunterladen.
2. *Entpacken*: Das SDK an einem Speicherort wie `*C:\flutter*` ablegen.
3. *Umgebungsvariablen setzen*: Den Pfad `*C:\flutter\bin*` zur Umgebungsvariable *Path* hinzufügen.
4. *Abhängigkeiten installieren*: Android Studio oder Visual Studio Code für die Entwicklung nutzen.
5. *Installation überprüfen*: Den Befehl `*flutter doctor*` im Terminal ausführen und ggf. auf der Basis der ausgewiesenen Meldungen Installationen und Konfigurationen anpassen.

**macOS:**

1. *Installation über Homebrew*:

brew install --cask flutter

1. *Manuelle Installation:* Alternativ kann das SDK als ZIP-Datei heruntergeladen und in *`/Users/<benutzername>/flutter*` entpackt werden.
2. *Umgebungsvariablen setzen*: Den Pfad zur `*flutter/bin*`-Ordner der Datei *`.zshrc*` oder *`.bash\_profile*` hinzufügen.
3. *Xcode installieren*: Das kann über den App Store geschehen.
4. *Abhängigkeiten prüfen*:

flutter doctor

Nach der Installation stehen verschiedene Werkzeuge zur Verfügung, die für die Entwicklung mit Flutter und Dart benötigt werden. Das *Flutter SDK* enthält alles, was für die Entwicklung von Flutter-Apps benötigt wird, darunter die Flutter-Bibliotheken, Tools zur Paketverwaltung und eine eigene Rendering-Engine. Das *Dart SDK* ist direkt im Flutter SDK enthalten und ermöglicht die Entwicklung in der Sprache Dart. Es enthält einen Compiler, die Standardbibliotheken und Entwicklungswerkzeuge. Flutter bietet leistungsfähige *Command Line Interface (CLI)*-Tools, um Projekte zu verwalten. Einige wichtige Befehle:

* *`flutter doctor`:* Überprüft die Installation und zeigt fehlende Komponenten an. Sie führen auf der Kommandozeile lediglich folgenden Befehl aus:

flutter doctor

* `*flutter create <projektname>`:* Erstellt ein neues Flutter-Projekt. Ein Beispiel:

flutter create my\_app

* *`flutter run`:* Startet die Anwendung in einem Emulator oder auf einem verbundenen Gerät

flutter run

* *`flutter pub get`:* Lädt Abhängigkeiten für ein Flutter-Projekt:

flutter pub get

* *`flutter build`:* Erstellt eine lauffähige APK- oder iOS-App für die Veröffentlichung.

flutter build

Nach der Installation können wir direkt mit dem Erstellen eines ersten Flutter-Projekts beginnen. Das kann über die Kommandozeile erfolgen:

flutter create my\_first\_app

cd my\_first\_app

Um das Projekt im Android-Emulator oder iOS-Simulator auszuführen muss dieser installiert und konfiguriert werden dann kann man die App von der Kommandozeile aus starten:

`flutter run`

Alternativ kann man die App auch aus der Entwicklungsumgebung Android Studio bzw. Visual Studio Code heraus beispielsweise als Web-Anwendung im aktuellen Browser ausführen (*Abbildung 2*)

Abbildung\_2.png

Abbildung 2: Hello World App (Web-Browser).

Untersuchen wir im Folgenden die Projektstruktur einer Flutter-App. Nach dem Erstellen eines Flutter-Projekts enthält der Projektordner verschiedene Verzeichnisse und Dateien:

* *my\_first\_app/:* Hauptprojektordner
* *android/:* Native Android-Projektstruktur
* *ios/:* Native iOS-Projektstruktur
* *lib/:* Hauptverzeichnis für Dart-Code
* *main.dart:* Einstiegspunkt der App
* *test/:* Unit-Tests für die App
* *assets/:* Bilder, Schriftarten, etc.
* *pubspec.yaml:* Konfigurationsdatei für Abhängigkeiten und Ressourcen
* *README.md*: Dokumentation zur App.

Der Einstiegspunkt einer Flutter-App ist die *main()*-Funktion, die in der Datei *lib/main.dart* definiert ist. Diese Funktion dient als Ausgangspunkt für die Ausführung der Anwendung. Innerhalb der *main()-*Funktion wird in der Regel die *runApp()-*Funktion aufgerufen, die das Haupt-Widget der Anwendung in den Widget-Baum einfügt und die Ausführung der Flutter-App startet. Ein Beispiel für eine minimale Flutter-App sieht wie folgt aus:

import 'package:flutter/material.dart';

void main() {

runApp(MyApp());

}

class MyApp extends StatelessWidget {

@override

Widget build(BuildContext context) {

return MaterialApp(

home: Scaffold(

appBar: AppBar(title: Text('Flutter App')),

body: Center(child: Text('Hallo, Flutter!')),

),

);

}

}

Sehen wir uns diese Zeilen des Quellcodes Schritt für Schritt an. Der Code definiert eine einfache Flutter-App mit einer App-Leiste (*AppBar*) und einem zentrierten *Text* („*Hallo, Flutter!*“). Die Anwendung basiert auf Material Design, dem User Interface-Framework von Google.

**Import der Flutter-Bibliothek:**

import 'package:flutter/material.dart';

* Dies importiert die Material Design-Bibliothek von Flutter.
* Diese Bibliothek enthält eine Vielzahl vorgefertigter Widgets für User Interface-Elemente wie Buttons, Texte, Layouts und vieles mehr.
* Alle Flutter-Apps basieren auf Widgets aus diesem oder einem anderen User Interface-Framework (Cupertino für iOS-Designs).

**Die main-Funktion – Einstiegspunkt der App:**

void main() {

runApp(MyApp());

}

* Dies ist der Einstiegspunkt jeder Dart- und Flutter-Anwendung.
* Die *main*-Funktion wird als Erstes ausgeführt, wenn die App startet.
* Die Funktion *runApp()* startet die Anwendung und übergibt das Haupt-Widget (*MyApp*).
* *MyApp* ist das Haupt-Widget der App, das die gesamte Benutzeroberfläche definiert.

**Das Haupt-Widget MyApp (eine StatelessWidget-Klasse):**

class MyApp extends StatelessWidget {

@override

Widget build(BuildContext context) {

return MaterialApp(

home: Scaffold(

appBar: AppBar(title: Text('Flutter App')),

body: Center(child: Text('Hallo, Flutter!')),

),

);

}

}

* *MyApp* ist ein *Stateless Widget* (unveränderliches Widget).
* Es erbt von *StatelessWidget*, was bedeutet, dass es keinen internen Zustand hat.
* *@override*: Überschreibt die Methode *build(),* welche das User Interface des Widgets definiert.

**Das build-Methoden-Widget:**

@override

Widget build(BuildContext context) {

return MaterialApp(

home: Scaffold(

appBar: AppBar(title: Text('Flutter App')),

body: Center(child: Text('Hallo, Flutter!')),

),

);

}

* Diese Methode wird von Flutter aufgerufen, um das User Interface-Layout der App zu erstellen.
* Sie gibt ein Widget zurück, das die Struktur der App definiert.

**Das MaterialApp-Widget:**

return MaterialApp(

home: Scaffold(

* *MaterialApp:* Dies ist das Haupt-Widget für jede Flutter-App, die das Material-Design-Framework verwendet.
* Es stellt grundlegende User Interface-Funktionen bereit, wie Navigation, Themes und Lokalisierung.
* Das *home*-Attribut gibt das Haupt-Widget der App an (hier *Scaffold*).

**Das Scaffold-Widget – Das Grundgerüst der App**

Scaffold(

appBar: AppBar(title: Text('Flutter App')),

body: Center(child: Text('Hallo, Flutter!')),

),

* *Scaffold*: Stellt das Grundgerüst der App bereit; enthält typische Layout-Elemente wie die App-Leiste (*AppBar*) und den Body-Bereich.
* *appBar*: *AppBar(title: Text('Flutter App')):* Erstellt eine App-Leiste mit dem Titel "*Flutter App*"; *AppBar* ist ein vordefiniertes Widget, das eine Titelleiste im Material-Design-Stil anzeigt.
* *body: Center(child: Text('Hallo, Flutter!')):* Definiert den Hauptinhalt der App; *Center* ist ein Layout-Widget, das den enthaltenen Inhalt in der Mitte des Bildschirms platziert. *Text('Hallo, Flutter!')* zeigt eine einfache Textnachricht in der Mitte des Bildschirms an.

Dieser Code erstellt eine minimale Flutter-App mit folgenden Elementen: *MaterialApp* (als Container für die gesamte Anwendung); *Scaffold* (als Grundstruktur der Benutzeroberfläche); AppBar (für die obere Titelleiste mit "*Flutter App*") und einen zentrierten Text (*Text('Hallo, Flutter!'*)) als Hauptinhalt der App. Im nächsten Schritt erweitern wir dieses „*Hello World*“-Beispiel. Wir ergänzen einen *Button* im Screen. Dazu modifizieren wir den Quellcode wie folgt:

import 'package:flutter/material.dart';

void main() {

runApp(MyApp());

}

class MyApp extends StatelessWidget {

@override

Widget build(BuildContext context) {

return MaterialApp(

home: Scaffold(

appBar: AppBar(title: Text('Flutter App')),

body: Center(

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: [

Text('Hallo, Flutter!'),

SizedBox(height: 20), // Abstand zwischen den Widgets

ElevatedButton(

onPressed: () {

print('Button geklickt!');

},

child: Text('Klick mich!'),

),

],

),

),

),

);

}

}

Was wurde geändert?

* *Column*-Widget wurde hinzugefügt, um Text und Button untereinander anzuordnen.
* *ElevatedButton* wurde hinzugefügt, um Interaktion zu ermöglichen.
* *print('Button geklickt!')* gibt beim Klicken eine Nachricht in der Konsole aus.

Dieser Code definiert eine einfache Flutter-App mit einer Titelleiste und einem Text. Die Struktur basiert auf Material Design mit *MaterialApp*, *Scaffold* und *AppBar*. Die Benutzeroberfläche besteht aus einem zentrierten Text in einem *Center*-Widget. Die App kann leicht erweitert werden, indem interaktive Elemente wie Buttons oder Layouts hinzugefügt werden. Fassen wir diese ersten Erkenntnisse zusammen: Nach der Einrichtung der Entwicklungsumgebung kann mit der Entwicklung von Flutter-Apps begonnen werden. Flutter bietet eine leistungsstarke CLI, mit der sich Projekte einfach verwalten lassen. Durch die Nutzung des `*flutter doctor*`-Befehls können Installationsprobleme schnell erkannt und behoben werden. Der Code wird dabei in der Programmiersprache Dart geschrieben. Im nächsten Abschnitt geht es um die Flutter-Widgets, die für die Gestaltung der Benutzeroberfläche einer App verwendet werden. Auch diese werden in Dart programmiert.

Die Grundlagen von Flutter-Widgets

Widgets sind das Herzstück einer jeden Flutter-Anwendung. Sie definieren die Benutzeroberfläche und bestimmen, wie sich eine App visuell darstellt und interaktiv verhält. Flutter folgt einem vollständig widgetbasierten Ansatz, bei dem nahezu alles – von einfachen Texten bis hin zu komplexen Layouts – als Widget behandelt wird. Mit anderen Worten: „*Alles ist ein Widget*“.

Widgets sind die Bausteine einer Flutter-Anwendung und bilden die gesamte Benutzeroberfläche ab. Es gibt drei Arten von Widgets in Bezug auf Ihre Funktion.

* *Struktur-Widgets*: Bestimmen das grundlegende Layout der App (z. B. `Scaffold`, `AppBar`).
* *Steuer-Widgets*: Ermöglichen Interaktionen (z. B. `*Button*`, `*TextField*`).
* *Widgets für die Anzeige*: Statische oder dynamische Anzeige von Inhalten (z. B. `*Text*`, `*Image*`).

Ein einfaches Beispiel für ein Widget ist die Anzeige eines Textes:

import 'package:flutter/material.dart';

void main() {

runApp(

MaterialApp(

home: Scaffold(

body: Center(

child: Text('Hallo, Flutter!'),

),

),

),

);

}

Erklärung: `*MaterialApp*` ist die App-Struktur mit Material-Design; `*Scaffold* ist das Basislayout mit App-Bar, Body-Bereich usw.; `*Center*` zentriert das darunterliegende Widget und `*Text*` ist ein einfaches Widget, das einen *String* anzeigt.

Ein weiteres Unterscheidungsmerkmal ist die Unterteilung in *Stiefel* und *Stateless Widgets*:

* *Stateless Widgets:* Unveränderlich, keine dynamische Datenänderung.
* *Stiefel Widgets:* Besitzen einen internen Zustand, der sich ändern kann.

Stateless Widgets

*Stateless Widgets* sind unveränderlich, d. h. sie enthalten keinen Zustand und zeigen immer die gleichen Inhalte an. Ein Beispiel für ein Stateless Widget ist:

import 'package: flutter/material. Dart';

void main() {

rump(Mapp());

}

class Mapp extends StatelessWidget {

@override

Widget build(Build Context context) {

return MaterialApp(

home: Scaffold(

appBar: AppBar(title: Text('Stateless Widget')),

body: Center(child: Text('Ich bin statisch!')),

),

);

}

}

Erklärung: `*MyApp*` erbt von `*StatelessWidget*`, weil sich der Inhalt nicht ändern kann. Das Widget bleibt unverändert, egal wie oft es aufgerufen wird.

Stateful Widgets

*Stateful Widgets* speichern und verwalten einen internen Zustand, der sich ändern kann. Sie bestehen aus zwei Klassen:

* Die Hauptklasse (`*MyStatefulWidget*`): Erstellt das Widget.
* Der zugehörige Zustand (`*\_MyStatefulWidgetState*`): Verwaltet die Änderungen.

Ein Beispiel für ein Stateful Widget mit Zähler-Button:

import 'package:flutter/material.dart';

void main() {

runApp(MyApp());

}

class MyApp extends StatelessWidget {

@override

Widget build(BuildContext context) {

return MaterialApp(

home: CounterWidget(),

);

}

}

class CounterWidget extends StatefulWidget {

@override

\_CounterWidgetState createState() => \_CounterWidgetState();

}

class \_CounterWidgetState extends State<CounterWidget> {

int counter = 0;

void increment() {

setState(() {

counter++;

});

}

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: Text('Stateful Widget')),

body: Center(

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: [

Text('Zähler: $counter', style: TextStyle(fontSize: 24)),

ElevatedButton(

onPressed: increment,

child: Text('Erhöhen'),

),

],

),

),

);

}

}

Erklärung: Die Klasse `*CounterWidget*` verwaltet ihren eigenen Zustand (`*counter*`).

* *`setState()`:* Wird aufgerufen, um den Zustand zu ändern und das User Interface zu aktualisieren.
* *‘onPressed()‘:* Erhöht den Zähler und aktualisiert den Bildschirm.

Eine Frage des Designs

Flutter bietet zwei Stile für Benutzeroberflächen. Das sind:

* Material Design (Google): `*AppBar*`, `*FloatingActionButton*`, `*TextField*`
* Cupertino (Apple): `*CupertinoButton*`, `*CupertinoNavigationBar*`, `*CupertinoAlertDialog*`

Sehen wir uns das an einem Beispiel an. Einen Material Design Button definieren wir mit folgenden Quellcodezeilen in Dart:

ElevatedButton(

onPressed: () => print('Button geklickt'),

child: Text('Material Button'),

)

Einen Button mit identischer Funktionalität, jedoch im Stil eines Cupertino Buttons wird durch folgenden Quellcode beschrieben:

CupertinoButton(

onPressed: () => print('Button geklickt'),

child: Text('Cupertino Button'),

)

Welche Unterschiede bzw. Gemeinsamkeiten gibt es in Fragen des Designs? Material Design Widgets passen sich optisch an Android und iOS an, d.h. wir können diese grundsätzlich auf beiden Systemen einsetzen. Auch eine Verwendung in Desktop- und Web-Apps ist problemlos möglich. Cupertino Widgets bieten natives iOS-Design, werden jedoch nicht automatisch für Android optimiert.

Strukturierung einer App mit Scaffold, AppBar, Buttons und Layouts

Ein Flutter-Layout besteht aus mehreren grundlegenden Widgets, die zusammen eine vollständige App bilden. Ein `*Scaffold*` ist das grundlegende Layout einer App, das typische Elemente wie *AppBar*, *FloatingActionButton* und einen *Body* enthält. Blicken wir in den Quellcode:

Scaffold(

appBar: AppBar(title: Text('Meine App')),

body: Center(child: Text('Hallo, Welt!')),

floatingActionButton: FloatingActionButton(

onPressed: () {},

child: Icon(Icons.add),

),

);

Wir haben es mit folgenden Bestandteilen zu tun:

* `*AppBar*`: Die Titelleiste der Anwendung.
* `*Body*`: Der Hauptinhalt der App.
* `*FloatingActionButton*`: Ein schwebender Aktionsknopf.

Flutter bietet mehrere Arten von Buttons. Das sind der *ElevatedButton*, ein erhabener, klickbarer Button; der *TextButton*, ein schlichter Textbutton ohne Hintergrund und ein *OutlinedButton*, d.h. ein Button mit Rahmen. Ein Beispiel für verschiedene Buttons sieht wie folgt aus:

Column(

children: [

ElevatedButton(onPressed: () {}, child: Text('Elevated Button')),

TextButton(onPressed: () {}, child: Text('Text Button')),

OutlinedButton(onPressed: () {}, child: Text('Outlined Button')),

],

)

Um Widgets anzuordnen, gibt es `*Column*`, `*Row*` und `*Stack*`. Dabei gilt folgendes. Ein `*Column*` ordnet Widgets vertikal an, ein `*Row*` ordnet die Widgets horizontal an und ein `*Stack*` platziert Widgets überlappend. Betrachten Sie beispielhaft die Arbeit mit einem *Column*:

Column(

mainAxisAlignment: MainAxisAlignment.center,

children: [

Text('Erstes Widget'),

Text('Zweites Widget'),

],

)

Wir können sagen: Widgets sind die Grundlage jeder Flutter-App und können entweder statisch (`*StatelessWidget*`) oder dynamisch (`*StatefulWidget*`) sein. Flutter bietet eine breite Auswahl an vorgefertigten Widgets für Material Design (Android) und Cupertino (iOS), wodurch plattformübergreifende Entwicklung effizient möglich ist. Das Grundgerüst einer App besteht aus `*Scaffold*`, `*AppBar*`, *Buttons* und *Layouts*, mit denen sich ansprechende Benutzeroberflächen gestalten lassen. Eine kompakte Übersicht über wichtige Basis Widgets liefert *Tabelle 1*.

|  |  |  |
| --- | --- | --- |
| Widget | Beschreibung | Quellcode |
| Text | Zeigt Text auf dem Bildschirm an. | Text('Hallo, Flutter!') |
| Icon | Zeigt ein Symbol aus den Material Icons an. | Icon(Icons.star, size: 50, color: Colors.yellow) |
| Container | Ein flexibles Widget für Layout und Styling. | Container(  width: 100,  height: 100,  color: Colors.blue, ) |
| SizedBox | Es ist eine einfache Box mit einer bestimmten Größe. Es kann verwendet werden, um Größenbeschränkungen für das untergeordnete Widget festzulegen oder eine leere SizedBox zwischen die beiden Widgets zu setzen, um etwas Platz dazwischen zu schaffen, | SizedBox(width: 50, height: 20) |
| Center | Zentriert das enthaltene Widget. | Center(child: Text('Zentriert')) |
| Align | Richtet das enthaltene Widget aus. | Align(alignment: Alignment.bottomRight, child: Text('Rechts unten')) |
| Column | Ordnet Widgets vertikal an. | Column(  children: [  Text('Erstes Widget'),  Text('Zweites Widget'),  ], ) |
| Row | Ordnet Widgets horizontal an. | Row(  children: [  Icon(Icons.star),  Icon(Icons.favorite),  ], ) |
| Expanded | Erweitert das Widget innerhalb eines Flex-Containers. | Expanded(child: Text('Erweiterbar')) |
| Flexible | Passt sich dynamisch an die Größe des Eltern-Widgets an. | Flexible(child: Text('Flexibles Widget')) |
| Stack | Überlagert mehrere Widgets. | Stack(  children: [  Image.asset('assets/hintergrund.png'),  Positioned(  bottom: 20,  child: Text('Überlagert'),  ),  ], ) |
| ListView | Erstellt eine scrollbare Liste. | ListView(  children: [  ListTile(  leading: Icon(Icons.task),  title: Text('Aufgabe 1'),  ),  ], ) |
| ElevatedButton | Ein erhöhter Button mit Schatten. | ElevatedButton(  onPressed: () {},  child: Text('Klick mich'), ) |
| Image | Zeigt ein Bild aus einer URL oder lokalen Quelle an. | Image.asset('assets/image.png') |
| Image.network | Lädt ein Bild aus dem Internet. | Image.network('https://example.com/bild.jpg') |
| Image.asset | Lädt ein Bild aus lokalen Assets. | Image.asset('assets/mein\_bild.png') |
| Scaffold | Das Grundgerüst einer Flutter-App. | Scaffold(  appBar: AppBar(title: Text('Meine App')),  body: Center(child: Text('Hallo Welt!')), ) |
| Form | Container für Formulare mit Validierungsmöglichkeiten. | Form(child: TextFormField()) |
| FormField | Einzelnes Eingabefeld mit Validierung in einem Form-Widget. | TextFormField(decoration: InputDecoration(labelText: 'Eingabe')) |
| RichText | Ermöglicht formatierte und gestylte Texte. | RichText(text: TextSpan(text: 'Fetter Text', style: TextStyle(fontWeight: FontWeight.bold))) |
| FutureBuilder | Ermöglicht den asynchronen Aufbau der UI. | FutureBuilder(future: myFuture, builder: (context, snapshot) => Text(snapshot.data.toString())) |
| StreamBuilder | Reagiert auf Datenströme und aktualisiert das User Interface dynamisch. | StreamBuilder(stream: myStream, builder: (context, snapshot) => Text(snapshot.data.toString())) |

Tabelle 1: Widgets in Flutter.

State Management in Flutter

Der Zustand (*State*) einer Anwendung bestimmt, wie sich die Benutzeroberfläche je nach Interaktionen oder externen Datenquellen verändert. In Flutter ist *State Management* ein zentrales Thema, da Flutter ein deklaratives User Interface-Framework ist, bei dem sich der Bildschirm nicht automatisch aktualisiert, wenn sich Daten ändern. Deshalb müssen Änderungen explizit verwaltet werden. In diesem Abschnitt wird erklärt, was *State* in Flutter bedeutet, welche gängigen Techniken zum State Management existieren und wie *setState()* für einfache Anwendungsfälle eingesetzt wird.

Der *State* einer App beschreibt den aktuellen Zustand der Benutzeroberfläche, also welche Daten angezeigt werden und wie Widgets darauf reagieren. Der State kann durch Benutzereingaben, Netzwerkdaten oder andere Ereignisse verändert werden. Wir unterscheiden:

* *lokaler State:* Betrifft nur ein einzelnes Widget und kann mit *`setState()`* aktualisiert werden. Beispiele: Button-Klicks, Eingabefelder, Animationen.
* *globaler State (App-weit):* Wird von mehreren Widgets gemeinsam genutzt und erfordert ein State-Management-Tool wie Provider, Riverpod oder Bloc. Beispiele sind Benutzeranmeldung, Warenkorb oder API-Daten.

Flutter bietet verschiedene Möglichkeiten zur Verwaltung des State, abhängig von der Komplexität der Anwendung (*Tabelle 2*).

|  |  |  |  |
| --- | --- | --- | --- |
| Technik | Verwendungszweck | Schwierigkeit | Typ |
| setState() | einfache lokale UI-Updates | einfach | lokaler State |
| InheritedWidget | manuelles State-Management für komplexe Widgets | mittel | App-weiter State |
| Provider | offizielle Google-Empfehlung für globalen State | mittel | App-weiter State |
| Riverpod | Erweiterung von Provider mit besserer Modularität | mittel | App-weiter State |
| Bloc (Business Logic Component) | Trennung von UI und Business-Logik für große Apps | fortgeschritten | App-weiter State |

Tabelle 2: Möglichkeiten zur Verwaltung des State in einer Flutter App.

Wann sollte welche Technik genutzt werden:

* *setState()* für einfache Änderungen innerhalb eines Widgets.
* *Provider/Riverpod* für mittlere bis große Anwendungen mit gemeinsam genutztem State.
* *Bloc* für komplexe, skalierbare Apps mit sauberer Trennung von UI und Logik.

Für einfache State-Änderungen innerhalb eines Widgets wird `*setState()`* verwendet. Dabei wird das Widget neu gerendert, wenn sich der Zustand ändert. Im vorangegangenen Beispielen (siehe vorherige Abschnitte) haben wir bereits `*setState()`* verwendet. Im nächsten Abschnitt wird im Rahmen eines komplexeren Beispiel die Nutzung der Bibliothek *Provider* zur Verwaltung des Status demonstriert.

Erstellung einer einfachen Flutter-App

In diesem Abschnitt entwickeln wir eine einfache To-Do-App mit Flutter, die es dem Benutzer ermöglicht, Aufgaben hinzuzufügen, zu entfernen und abzuhaken. Dabei werden wichtige Konzepte wie State Management, Trennung von UI und Logik behandelt. Die App soll folgende Funktionen bieten: Aufgaben hinzufügen, Aufgaben als erledigt markieren und Aufgaben löschen. Erstellen Sie ein neues Flutter-Projekt in Android-Studio (*Abbildung 3*).

Abbildung\_3.png

Abbildung 3: Erstellen eines neuen Flutter-Projektes in Android Studio.

Wir ergänze bzw. modifizieren die generierte „*Hello World*“-App zu folgender Projektstruktur unterhalb des Ordners */lib/,* dem Hauptverzeichnis für Dart-Code:

* *`main.dart`:* Einstiegspunkt der App
* *`todo\_screen.dart`:* UI-Logik der App
* *`todo\_model.dart`:* Datenmodell für To-Do-Elemente
* *`todo\_provider.dart`:* State Management für die Aufgabenliste.

Das Datenmodell der App wird mit Hilfe der Klasse ‘*ToDo*‘ in der Datei `todo\_model.dart` erstellt:

class ToDo {

String title;

bool isDone;

ToDo({required this.title, this.isDone = false});

}

Erklärung: Erstellt eine *ToDo*-Klasse mit zwei Eigenschaften: `*title*` (Aufgabentext) und `*isDone*` (Status). Es wird ein Konstruktor definiert, welcher die Eigenschaft `*isDone*` standardmäßig auf den Wert `*false*` (Aufgabe noch nicht erledigt) setzt.

Für die Verwaltung des Status eines Eintrages in der To-Do-Liste bauen wir ein State Management mit der Bibliothek *Provider* auf. Dazu legen wie die Klasse *ToDoProvider* in der Datei `*todo\_provider.dart*` an:

import 'package:flutter/material.dart';

import 'todo\_model.dart';

class ToDoProvider with ChangeNotifier {

List<ToDo> \_tasks = [];

List<ToDo> get tasks => \_tasks;

void addTask(String title) {

\_tasks.add(ToDo(title: title));

notifyListeners();

}

void toggleTask(int index) {

\_tasks[index].isDone = !\_tasks[index].isDone;

notifyListeners();

}

void removeTask(int index) {

\_tasks.removeAt(index);

notifyListeners();

}

}

Auch hier sind einige Erklärungen notwendig: Die Klasse verwaltet eine Liste von *ToDo*-Elementen. Die Methode *`addTask()`* fügt eine neue Aufgabe hinzu; *`toggleTask()`* markiert eine Aufgabe als erledigt bzw. nicht erledigt und *`removeTask()`* entfernt eine Aufgabe aus der Liste. Mit jeder Änderung der Einträge in der Liste wird die Methode *`notifyListeners()`* aufgerufen, welche das User Interface aktualisiert, wenn sich die Liste ändert. Wie funktioniert *notifyListeners()?* *notifyListeners()* ist Teil der *ChangeNotifier*-Klasse, die in Flutter zur Implementierung von State Management mit Provider genutzt wird.

Damit man die Provider-Bibliothek nutzen kann ist diese zu installieren. Wir ergänzen dazu in der Datei *pubspec.yaml* den folgenden Eintrag (Abhängigkeit):

dependencies:  
 flutter:  
 sdk: flutter  
 provider: ^6.1.2

Hinweis: *Pub* ist das offizielle Paketverwaltungssystem für Dart und Flutter. Es ermöglicht Entwicklern, externe Bibliotheken (Packages) und Werkzeuge zu verwalten, um die Funktionalität ihrer Anwendungen zu erweitern. Dabei können sowohl Open Source-Pakete aus dem offiziellen *Pub.dev*-Repository als auch eigene Pakete verwaltet werden. Die Verwaltung erfolgt über die Datei *pubspec.yaml.*

Widgets, die auf den veränderten Zustand angewiesen sind, lauschen (listen) auf Änderungen und werden neu gezeichnet. Das geschieht in der Klasse *ToDoScreen*, welche für die Darstellung des User Interfaces (Liste) zuständig ist:

import 'package:flutter/material.dart';

import 'package:provider/provider.dart';

import 'todo\_provider.dart';

class ToDoScreen extends StatelessWidget {

final TextEditingController \_controller = TextEditingController();

@override

Widget build(BuildContext context) {

final todoProvider = Provider.of<ToDoProvider>(context);

return Scaffold(

appBar: AppBar(title: Text('To-Do Liste')),

body: Column(

children: [

Padding(

padding: EdgeInsets.all(8.0),

child: TextField(

controller: \_controller,

decoration: InputDecoration(

labelText: 'Neue Aufgabe',

suffixIcon: IconButton(

icon: Icon(Icons.add),

onPressed: () {

if (\_controller.text.isNotEmpty) {

todoProvider.addTask(\_controller.text);

\_controller.clear();

}

},

),

),

),

),

Expanded(

child: ListView.builder(

itemCount: todoProvider.tasks.length,

itemBuilder: (context, index) {

final task = todoProvider.tasks[index];

return ListTile(

title: Text(

task.title,

style: TextStyle(

decoration: task.isDone ? TextDecoration.lineThrough : null,

),

),

leading: Checkbox(

value: task.isDone,

onChanged: (value) => todoProvider.toggleTask(index),

),

trailing: IconButton(

icon: Icon(Icons.delete),

onPressed: () => todoProvider.removeTask(index),

),

);

},

),

),

],

),

);

}

}

Erklärung: Der Code enthält ein Eingabefeld für neue Aufgaben; zeigt eine Liste von Aufgaben an, die mit `*ListView.builder*` erstellt wird; es gibt eine Checkbox zum Markieren von erledigten Aufgaben und einen Lösch-Button zum Entfernen einer Aufgabe. Beispielhaft lässt sich das Zusammenspiel der *Provider*-Bibliothek zur Verwaltung des Zustandes und der User Interface-Aktualisierung durch folgende Schritte beschreiben:

1. Der Benutzer tippt "Einkaufen" ins Textfeld
2. Klick auf den "Hinzufügen"-Button
3. *addTask("Einkaufen")* wird aufgerufen
4. *notifyListeners()* wird aktiviert
5. Das User Interface erkennt die Änderung und ruft *build()* neu auf
6. *ListView.builder* zeigt die aktualisierte Liste mit "Einkaufen" an.

Das Ergebnis: Die neue Aufgabe erscheint sofort auf dem Bildschirm.

Als letztes werfen wir noch einen Blick in die Datei `*main.dart`* unserer App:

import 'package:flutter/material.dart';

import 'package:provider/provider.dart';

import 'todo\_provider.dart';

import 'todo\_screen.dart';

void main() {

runApp(

ChangeNotifierProvider(

create: (context) => ToDoProvider(),

child: MyApp(),

),

);

}

class MyApp extends StatelessWidget {

@override

Widget build(BuildContext context) {

return MaterialApp(

debugShowCheckedModeBanner: false,

home: ToDoScreen(),

);

}

}

Beachten Sie dazu die folgende Erklärung: *`ChangeNotifierProvider`* stellt `*ToDoProvider*` für die gesamte App bereit; *`MaterialApp`* startet die App mit dem *`ToDoScreen()`* und der Zusatz *`debugShowCheckedModeBanner: false`* entfernt das Debug-Banner beim Ausführen der App. Starten wir die App, beispielsweise als Web-Applikation im aktuellen Browser (*Abbildung 4*) oder alternativ auf einem Android-Emulator (*Abbildung 5*).

Abbildung\_4.png

Abbildung 4: Beispiel App (To-Do-Liste) als Web-App.

Abbildung\_5.png

Abbildung 5: Beispiel App (To-Do-Liste) als Android-App.

Debugging

Auch auf diesen Bereich der Entwicklung möchten wir einen kurzen Blick werfen. Flutter bietet leistungsstarke Debugging-Tools. Konkret sind es:

* Flutter DevTools: *`flutter pub global activate devtools`*
* *Hot Reload & Hot Restart:* zur schnellen Aktualisierung des Codes
* *`print()`* für direkte Debugging-Ausgaben
* *debugPrint()`* für Log-Ausgaben
* Breakpoint-Debugging in Visual Studio Code oder Android Studio

Wir haben eine To-Do-App mit Flutter entwickelt, die zeigt, wie man State Management mit Provider verwendet und eine saubere Trennung von User Interface und Logik erreicht.

Schneller Einsteigen

Sie wollen mehr von Flutter erfahren? Dann ist unser **Einstiegs-Webinar** in Flutter genau das richtige für Sie Buchen Sie es unter [3]. Den Quellcode zum Beispiel finden Sie ebenfalls dort zum Download.

./.

Fazit und Ausblick

Die Kombination von Dart und Flutter bietet eine leistungsstarke Plattform zur Entwicklung plattformübergreifender Anwendungen. Durch das widgetbasierte UI-System ermöglicht Flutter eine konsistente Gestaltung von Benutzeroberflächen für verschiedene Betriebssysteme. Die Hot-Reload-Funktion steigert die Effizienz der Entwicklung erheblich, während das State-Management mit Provider eine robuste Methode zur Steuerung der Anwendungslogik bietet. Im Artikel haben wir die Grundlagen von Flutter erläutert, die Architektur, die Einrichtung der Entwicklungsumgebung sowie die Erstellung einer einfachen App mit State-Management demonstriert. Besonders hervorzuheben ist die einfache Skalierbarkeit, da Widgets wiederverwendet und UI-Elemente dynamisch angepasst werden können. Damit ermöglicht Flutter sowohl für kleine als auch komplexe Anwendungen eine hohe Performance und Flexibilität. Im nächsten Artikel werden wir Best Practices und fortgeschrittene Konzepte rund um die Programmierung mit der Sprache Dart behandeln.

*Dr. Veikko Krypczyk ist begeisterter Entwickler und Fachautor. Er bietet Seminare und Workshops zu unterschiedlichen Themen der Softwareentwicklung. Weitere Informationen zu diesen und anderen Themen der IT und Angebote zu Seminaren und Workshops finden Sie unter https://larinet.com.*

Links & Literatur

[1] https://docs.flutter.dev/resources/architectural-overview

[2] <https://docs.flutter.dev/get-started/install>

[3] <https://larinet.com>