01

Flutter

00. 필요 도구 설치

Flutter SDK 설치

- https://docs.flutter.dev/get-started/install/windows/mobile
  - Download and install
  - Flutter_windows_3.32.2-stable.zip
- 압축을 풀고 환경변수(path) 등록
  - ....₩flutter₩bin
- >flutter
- >flutter doctor

Android Studio 설치

- https://developer.android.com/studio/?hl=ko
  - SDK Tools 탭 - Android SDK Command-line tools 체크
- Plugin 설치
  - Flutter, Dart

01. Flutter란

- 구글에서 개발한 오픈소스 UI 툴킷

- 하나의 코드베이스를 사용하여 iOS, Andriod, 웹 데스크탑 (Windows, macOS, Linux) 애플리케이션 개발

- 특징
  - 다양한 플랫폼 지원
  - Dart 언어 사용
  - Widget 기반 구조
  - Hot Reload
  - 고성능 랜더링
  - 풍부한 블러그인

02. Widget

- 독립적으로 실행되는 작은 프로그램
- 주로 바탕화면 등에서 날씨나 뉴스 생활정보를 보여줌
- 그래픽이나 데이터 요소를 처리하는 함수를 가짐

03. Widget in Flutter

- UI를 만들고 구성하는 모든 기본 단위 요소
  - Text, Icon, Image, Text field...
- 눈에 보이지 않는 구조를 표현하는 요소들
  - Center, Colmun, Row, Padding...
- Widget의 종류
  - Stateless Widget
  - Stateful Widget
  - Inherited Widget

04. Widget Tree

- Widget들은 Tree구조로 설계되어 있음
- 하나의 Widget안에 다른 Widget들이 포함될 수 있음
- Widget은 부모 Widget과 자식 Widget으로 구성
- Parent Widget을 Widget Container라고도 함

04. Widget Tree

05. Stateless Widget

- 변하지 않는 상태를 가지는 Widget
- 상태가 고정되어 있고 빌드 시점 이후에는 변경되지 않음
- 화면에 데이터를 렌더링 하지만 사용자 상호작용이나 기타 이벤트에 의해 자체적으로 상태를 변경할 수 없음
- Flutter가 제공하는 대부분의 Widget

05. Stateless Widget

특징

- 불변성
  - Widget이 한번 생성되면 상태나 데이터가 변하지 않음
  - 모든 데이터는 Widget의 생성자에게 전달받아 화면에 표시
- 단순한 UI 구성
  - 정적 컨텐츠(Text, Icon, Button등)을 표시
- 효율적인 리렌더링
  - 상태가 없기 때문에 Flutter의 효율적 트리 리렌더링 메커니즘을 활용

06. 기초 Widget

- **MaterialApp**
  - 앱의 초기화면, 테마등 기본 설정 정의
  - 주 속성 : title, theme, home, routes...
- **Scaffold**
  - 한 페이지에 해당하는 Widget들의 모음
  - 주 속성 : appBar, body..
- **Text**
  - 문자열 출력
  - 주 속성 : style, textAlign ..
- **Column**
  - 자식 Widget들을 세로로 배치
  - 주 속성 : children, mainAxisAlignment..

06. 기초 Widget

- **Row**
  - 자식 Widget들을 가로로 배치
  - 주 속성 : children, mainAxisAlignment..
- **Center**
  - 자식 Widget을 가운데로 배치
  - 주 속성 : child
- **Padding**
  - 자식 Widget들의 여백을 설정
  - 주 속성 : padding , child
- **Divider**
  - 구분선 추가
  - 주 속성 : color, thickness, endIndent, height...
- **SizedBox**
  - 빈공간 설정
  - 주 속성 : height, width

06. Button Widget

- **ElevatedButton**
  - 음영 효과가 있어 떠 있는 듯한 입체감을 주는 버튼
  - 주 속성 : child, onPress, style..
- **TextButton**
  - 배경이 없는 간단한 버튼, 주로 텍스트만 표시
  - 주 속성 : child, onPress, style..
- **OutlinedButton**
  - 버튼 테두리가 강조된 형태의 버튼
  - 주 속성 : child, onPress, style..
- **IconButton**
  - 아이콘만 포함된 버튼으로 짧은 작업에 사용
  - 주 속성 : icon, iconSize, onPress, style..

06. Button Widget

- **TextButton.icon**
  - icon과 텍스트의 조합을 표현하는 Widget
  - 주 속성 : icon, onPressed, label, style..
- **ElevatedButton.icon**
  - icon과 ElevatedButton의 조합을 표현하는 Widget
  - 주 속성 : icon, onPressed, label, style..

07. SnackBar

- 앱 화면 하단에 잠깐 표시되는 알람 Widget
- 주 속성 : content, action, backgroundColor, duration..
- 생성 예

```
ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(
                content: Text('This is a SnackBar!'),
                duration: Duration(seconds: 2),
));
```

08. Fluttertoast

- 앱에 간단한 알림 메시지를 잠시 나타나게 하는 Toast 메시지 라이브러리
- Fluttertoast 패키지 설치
  - fluttertoast: ^8.2.1
  - flutter pub get
- 생성 예

```
Fluttertoast.showToast(
        msg: "This is a Toast message",
        toastLength: Toast.LENGTH_SHORT,
        gravity: ToastGravity.BOTTOM,
        backgroundColor: Colors.black,
        textColor: Colors.white,
        fontSize: 16.0,
);
```

09. Navigator

- 화면 간 Navigation을 관리하는 Widget
- 페이지 전환, 스택 관리, 화면 뒤로가기 등을 처리
- 스택(Stack) 구조를 기반으로 동작
  - 화면을 스택에 추가하거나 제거하여 Navigation 흐름을 관리
- Route 관리
  - MaterialPageRounte를 사용해 새로운 페이지를 추가하거나 제거
- Named Routes 지원
  - 이름 있는 Route를 정의하여 Navigation을 간소화
- 메서드 : push, pop...

09. Navigator

생성 예

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

```
MaterialApp(
      initialRoute: '/',
      routes: {
        '/': (context)=>ScreenA(),
        '/b': (context)=>ScreenB(),
        '/c': (context)=>ScreenC(),
      },
    );
```

09. Container

- 레이아웃을 구성할 때 사용되는 Widget으로 특정 Widget을 감싸고 해당 Widget에 대해 배경색, margin, padding, size 등의 설정을 지정함

- 주 속성 : child, alignment, padding, margin, color, width, height, decoration 등..

09. Container

사용 예

```
Center(
  child: Container(
        width: 200,
        height: 100,
        padding: EdgeInsets.all(10),
        margin: EdgeInsets.all(20),
        decoration: BoxDecoration(
        color: Colors.blue,
        borderRadius: BorderRadius.circular(15),
       boxShadow: [
                    BoxShadow(
                          color: Colors.black.withOpacity(0.3),
                     blurRadius: 5,
                          offset: Offset(3, 3),
                ),
        ],
   ),
```

10. BoxDecoration

- Container와 같은 Widget의 배경, 테두리, 그림자, 모서리 반경 등을 꾸미는 데 사용되는 Decoration Class
- 주 속성 : color, border, borderRadius, boxShadow, image, shape..

11. GestureDetector

- 사용자의 다양한 제스처를 감지하고 처리할 수 있는 Widget
- 탭, 더블탭, 길게 누르기, 드래그, 플링 등이 포함

- 탭 (Tap)
  - onTap: 단일 탭 감지
  - onDoubleTap: 더블탭 감지
  - onSecondaryTap: 오른쪽 버튼(또는 보조 탭) 클릭 감지

- 길게 누르기 (Long Press)
  - onLongPress: 길게 누르기 감지
  - onLongPressStart, onLongPressEnd: 길게 누르기 시작 및 끝 이벤트 감지

11. GestureDetector

- 드래그 (Drag)
  - onPanStart, onPanUpdate, onPanEnd: 자유롭게 드래그하는 제스처 감지
  - onHorizontalDrag 및 onVerticalDrag: 특정 방향으로 드래그하는 제스처 감지
- 플링 (Fling)
  - onPanEnd: 드래그 후 빠르게 손을 떼는 동작
- 줌 및 핀치 (Scale)
  - onScaleStart, onScaleUpdate, onScaleEnd: 확대/축소 감지

12. Drawer

- AppBar에서 menu 아이콘을 눌렀을 때 나타나는 슬라이드 아웃 패널로 사용
- Scaffold의 AppBar에서 햄버거 아이콘을 통해 Drawer를 열수 있음
- 주 속성 : child, ListView, ListTile, DrawerHearder

13. Stateful Widget

- 상태를 관리할 수 있는 Widget

- 상태를 유지 가능
  - 데이터가 변경될 때마다 UI를 업데이트하며, 이전 상태를 유지할 수 있음

- 분리된 상태 관리
  - StatefulWidget은 자체적으로 UI와 상태 관리 로직을 분리
  - StatefulWidget 클래스는 immutable이며, 상태 변경은 State 클래스에서 처리

13. Stateful Widget

StatefulWidget의 구조

- StatefulWidget 클래스 : Widget의 변하지 않는 값 및 구조를 정의.
- State 클래스 : 상태를 관리하며, UI를 업데이트하는 로직을 포함.

```
class MyStatefulWidget extends StatefulWidget {
      @override
      _MyStatefulWidgetState createState() => _MyStatefulWidgetState();
}

class _MyStatefulWidgetState extends State<MyStatefulWidget> {
      int _counter = 0;
      void _incrementCounter() {
         setState(() {
            _counter++; // 상태 업데이트
          }); }
      @override Widget build(BuildContext context) {
             return Scaffold( .....);
 }}
```

14. StatefulWidget의 생명주기(Lifecycle)

- **initState()**
  - Widget이 처음 생성될 때 호출.
  - 초기화 작업 수행 (ex: 애니메이션 초기화, API 호출 등).
  - 주의점
    - BuildContext가 필요한 초기화 작업은 initState()에서 하지 않고 didChangeDependencies()에서 한다.

14. StatefulWidget의 생명주기(Lifecycle)

didChangeDependencies()

- StatefulWidget의 State 객체가 의존하는 객체가 변경되었을 때 호출.(InheritedWidget과 연동된 상태가 변경)

```
final theme = Theme.of(context);
final locale = Localizations.localeOf(context);
final mq = MediaQuery.of(context);
```

- 위와 같이 상위 Widget의 contex를 통해 데이터를 가져와 State 데이터로 사용하는 경우 상위 Widget에서 해당 내용이 변경되면 현재 Widget의 didChangeDependencies()가 호출

- State가 처음 생성되고 initState() 호출 후
  - initState() → didChangeDependencies() 순서로 한 번 호출된 후 상황에따라 didChangeDependencies()는 더 호출될 수 있음

14. StatefulWidget의 생명주기(Lifecycle)

- **build()**
  - Widget의 UI를 렌더링하는 메서드.
  - 상태가 변경되면 setState()를 호출하여 build()를 다시 실행.

14. StatefulWidget의 생명주기(Lifecycle)

- **setState()**
  - 상태를 업데이트하고 UI를 다시 빌드.

- **deactivate()**
  - Widget이 트리에서 제거되기 전에 호출.

- **dispose()**
  - Widget이 영구적으로 제거될 때 호출.
  - 리소스 정리 (ex: 컨트롤러 해제, 타이머 정지 등).

15. LisView

기본 사용법

```
ListView(
  children: <Widget>[
    ListTile(title: Text('Item 1')),
    ListTile(title: Text('Item 2')),
    ListTile(title: Text('Item 3')),
  ],
)
```

동적 리스트 생성(ListView.builder)

```
ListView.builder(
  itemCount: items.length,
  itemBuilder: (context, index) {
    return ListTile(
      title: Text(items[index]),
    );
  },
)
```

15. ListView

ListTile

```
ListTile(
  leading: Icon(Icons.person),       // 왼쪽 아이콘
  title: Text('홍길동'),              // 제목
  subtitle: Text('개발자'),           // 부제목 (선택 사항)
  trailing: Icon(Icons.arrow_forward), // 오른쪽 아이콘
  onTap: () {
    print('클릭됨');
  },
)
```

15. ListView

Card

```
Card(
  child: Padding(
    padding: EdgeInsets.all(16.0),
    child: Text('이것은 카드입니다'),
  ),
)
```

- 재질 디자인(Material Design) 스타일의 카드 형태 UI를 만들 때 사용하는 Widget
- 그림자, 모서리 둥글기, 테두리, 마진 등 다양한 스타일을 지정할 수 있으며, 다른 Widget들을 감싸는 컨테이너 역할로 많이 사용
- 보통 Padding, Column, ListTile, Image 등과 함께 사용

주요 속성

- child, color, elevation, margin, shape, shadowColor, ..

16. Stack

- 자식 Widget들을 겹쳐서 배치할 수 있는 Widget
  - 첫 번째 자식이 가장 아래에 배치
  - 마지막 자식이 가장 위에 배치
- Positioned Widget
  - Stack 내부의 자식 Widget을 특정 위치에 배치
  - top, bottom, left, right

```
Stack(
      children: [
            Container(width: 200, height: 200, color: Colors.red),
            Positioned( top: 20, left: 20,
                  child: Container(width: 100, height: 100,
                        color: Colors.green
                  ), ),
            Positioned( bottom: 10, right: 10,
                  child: Container(width: 50, height: 50, color: Colors.blue),
             ),
      ], ),
```

17. RichText

- 텍스트 안에 다양한 스타일(굵기, 색상, 크기 등)을 혼합해서 표현할 수 있도록 해주는 Widget
- TextSpan이라는 객체를 사용해서 여러 조각의 텍스트를 계층적으로 구성하고, 각각에 개별 스타일을 지정

```
RichText(
  text: TextSpan(
    text: '안녕하세요, ',
    style: TextStyle(color: Colors.black, fontSize: 20),
    children: [
      TextSpan(
        text: '홍길동',
        style: TextStyle(fontWeight: FontWeight.bold, color: Colors.blue),
      ),
      TextSpan(
        text: '님!',
      ),],),)
```

18. Form

- 여러 개의 입력 필드(TextFormField 등)를 그룹으로 묶고, 그 상태를 통합적으로 관리할 수 있도록 도와주는 컨테이너 Widget
- 입력 필드(TextFormField 등) 여러 개를 하나의 그룹으로 묶어 유효성 검사(Validation) 와 상태 저장 등을 편리하게 할 수 있게 해 줌
- GlobalKey<FormState>를 통해 FormState에 접근할 수 있고 전체 입력 필드에 대해 validate()나 save() 등을 호출 가능하게 함

18. Form

GlobalKey<FormState>

- FormState에 접근하기 위해 사용
- FormState는 Form Widget의 상태를 관리
- validate(), save(), reset() 같은 메서드를 호출하려면 FormState에 접근해야 하는데, 이때 GlobalKey를 사용
- 여러 Widget 트리 사이에서 Form 상태를 유지하거나 참조할 수 있게 함

```
final GlobalKey<FormState> _formKey = GlobalKey<FormState>();

@override
Widget build(BuildContext context) {
  return Form(
    key: _formKey,
    child: Column(
      ...
    ),  );
}
```

18. Form

```
children: [
        TextFormField(
          validator: (value) { // TextFormField에 입력된 데이터
            if (value == null || value.isEmpty) {
              return '값을 입력하세요.'; //유효하지 않음
            }
            return null; //유효
          },
          onSaved: (value) {
            print("데이터 저장");
        ),
        ElevatedButton(
          onPressed: () {
            if ( _formKey.currentState!.validate() ) {
              // 모든 유효성 검사 통과
              print( " 폼 유효성 검사 성공 " );
             _formKey.currentState!.save();
            } },
          child: Text('제출'),
        ), ],
```

18. Form

- _formKey.currentState!.validate() 호출
  - 각 TextFormField의 validator() 순차 호출
  - 모두 null을 리턴/유효 → true 반환
  - 하나라도 유효하지 않으면 false 반환

- _formKey.currentState!.save() 호출
  - 전제 조건: validate()에서 모든 필드가 유효해야 함
  - 각 TextFormField의 onSaved() 순차 호출
  - 사용자가 입력한 데이터를 변수나 모델에 저장

19. Provider

- 상태 관리 솔루션으로 상태를 관리하고 Widget Tree에 상태를 전달

- pubspec.yaml

```
dev_dependencies:
  flutter_test:
    sdk: flutter
  # ........
  flutter_lints: ^5.0.0
  provider: ^6.1.5
```

- dart file

```
import 'package:provider/provider.dart';
```

19. Provider

ChangeNotifier

- 상태를 관리하고 알림을 제공하는 클래스
- 상태 변경 시 notifyListeners()를 호출하여 상태를 사용하는 Widget들에게 공지(변경된 상태값으로 Widget을 다시 빌드해야 할 때)

```
class Counter extends ChangeNotifier {
  int _count = 0;

  int get count => _count;

  void increment() {
    _count++;
    notifyListeners(); // 상태 변경 알림
  }
}
```

19. Provider

ChangeNotifierProvider

- ChangeNotifier를 Widget 트리에 제공하는 역할
- 상태를 생성하고, 하위 Widget에서 이를 사용할 수 있도록 제공

```
void main() {
  runApp(
    ChangeNotifierProvider(
    create: (context) => SignUpProvider(),
    child: MyApp(),
    ),
  );
}
```

하위 Widget들이 상태값을 사용하는 방법

- **Counter** provider = context.watch<**Counter**>();
- provider.count, provider.increment();

19. Provider

MultiProvider

- 여러 개의 Provider를 등록하여 사용

```
void main() {
  runApp(
    MultiProvider(
      providers: [
        ChangeNotifierProvider(create: (context) => CounterProvider()),
        ChangeNotifierProvider(create: (context) => ToggleProvider()),
      ],
      child: MyApp(),
    ),
  );
}
```

20. final 과 const

final

- 한번만 값을 할당할 수 있음(Runtime 시점에 결정)
- final now = DateTime.now();
- final list1 = [1, 2];
- list1.add(3);
  - // 리스트 객체 자체는 final이지만, 내부 요소는 수정 가능

const

- 한번만 값을 할당할 수 있음(Compile 시점에 결정)
- const pi = 3.14159;
- 메모리 최적화도 됨(같은 값의 const 객체는 공유됨)
  - const a = [1, 2, 3];
  - const b = [1, 2, 3];

  - print(identical(a, b)); // true (같은 인스턴스)
- const list2 = [1, 2];
- list2.add(3); //error
- // 리스트도 내부 요소도 모두 불변

21. Font 적용

- https://fonts.google.com/ 에서 폰트 파일 다운로드
- 프로젝트폴더의 assets/font 안에 .ttf파일 복사
- **Pubspec.yaml**

```
fonts:
  - family: AstaSans
    fonts:
      - asset: assets/font/AstaSans-VariableFont_wght.ttf
      - asset: assets/font/AstaSans-ExtraBold.ttf
        weight: 700  //실제 코드에서 연결될 FontWeight.w700
  - family: Montserrat
    fonts:
      - asset: assets/font/Montserrat-Bold.ttf
        weight: 700
      - asset: assets/font/Montserrat-BoldItalic.ttf
        style: italic
        weight: 700
```

21. Font 적용

코드에서 적용

```
Text(“Hello World",
                style: TextStyle(
                      fontFamily: "Montserrat",
                      fontWeight: FontWeight.w700,
                         //pubspect.yaml에서 설정한 값과 일치
                      fontSize: 20,
                      fontStyle: FontStyle.italic
              ),),
```

02

Flutter Network

01. Future

- 비동기 작업을 나타내는 객체
- 현재 실행 중인 작업이 완료되기 전까지는 결과를 즉시 반환하지 않고, 결과를 나중에 제공
- 두 가지 상태:
  - 완료: 작업이 성공적으로 끝나고 결과를 반환.
  - 에러: 작업 중 문제가 발생하여 예외를 반환.
- 결과 처리 방법:
  - 결과가 준비되면 then을 사용하여 처리.
  - 에러가 발생하면 catchError를 사용하여 처리.

01. Future

사용법

```
Future<String> fetchData() async {
  await Future.delayed(Duration(seconds: 2)); // 2초 대기
  return "Hello, Future!";
}

void main() {
  fetchData().then((data) {
    print(data); // "Hello, Future!"
  }).catchError((error) {
    print("Error: $error");
  });
}
```

02. FutureBuilder

- Flutter에서 비동기 작업(Future)의 상태를 감시하고, 해당 상태에 따라 적절한 UI를 표시할 수 있게 해주는 Widget
- 데이터 로드 중에는 로딩 인디케이터를 표시하고, 데이터 로드가 완료되면 데이터를 화면에 표시하거나, 에러가 발생하면 에러 메시지를 표시
- FutureBuilder는 Future를 감시하여 상태 변화(waiting, done, error)에 따라 자동으로 UI를 업데이트
- 주요 속성
  - Future : 감시할 Future 객체
  - Builder : snapshot을 기반으로 UI를 빌드하는 콜백 함수
    - builder 함수는 매번 상태가 변경될 때 호출(waiting, done, error)

02. FutureBuilder

AsyncSnapshot

- builder 함수에 전달되는 snapshot 객체
- connectionState : 현재 비동기 작업의 상태
  - ConnectionState.none: Future가 아직 시작되지 않음
  - ConnectionState.waiting: Future가 대기 중(작업 진행 중)
  - ConnectionState.done: Future 작업 완료.
- data : Future 작업이 성공적으로 완료되면 반환되는 결과 데이터

builder 함수 호출 시점

- FutureBuilder가 처음 생성될 때
- future의 상태가 변경될 때(waiting -> done, eror)
- setState나 외부 상태 변화로 FutureBuilder가 재빌드될 때

02. FutureBuilder

기본 구조

```
FutureBuilder<T>(
  future: myFuture, // 비동기 작업 (Future<T>)
  builder: (BuildContext context, AsyncSnapshot<T> snapshot) {
    if (snapshot.connectionState == ConnectionState.waiting) {
      return Center(child: CircularProgressIndicator()); // 로딩 중
    } else if (snapshot.hasError) {
      return Center(child: Text('Error: ${snapshot.error}')); // 에러 발생
    } else if (snapshot.hasData) {
      return Text('Data: ${snapshot.data}'); // 데이터 로드 완료
    } else {
      return Center(child: Text('No data available')); // 데이터가 없는 경우
    }
  },
);
```

03. http

Package 설치

- **http: ^1.2.0**
- **import 'package:http/http.dart' as http;**

```
Future<void> fetchData() async {
  final url = Uri.parse('https://jsonplaceholder.typicode.com/posts/1');

  try {
    final response = await http.get(url);
    if (response.statusCode == 200) {
      print('Response data: ${response.body}');
    } else {
      print('Failed to load data. Status code: ${response.statusCode}');
    }
  } catch (e) {
    print('Error: $e');
  }
}
```

03. http

```
Future<void> sendData() async {
  final url = Uri.parse('https://jsonplaceholder.typicode.com/posts');
  final body = {'title': 'foo', 'body': 'bar', 'userId': '1' };

  try {
    final response = await http.post(
      url,
      headers: {
        'Content-Type': 'application/json',
      },
      body: http.jsonEncode(body),
    );

    if (response.statusCode == 201) {
      print('Response data: ${response.body}');
    } else {
      print('Failed to send data. Status code: ${response.statusCode}');
    }
  } catch (e) {
    print('Error: $e');
  }
}
```

04. factory

- **factory 키워드로 정의된 생성자는 클래스의 새 인스턴스를 반환하는 역할**
- **이 생성자는 반드시 다른 생성자를 호출하여 객체를 반환해야 함.**

```
factory Todo.fromJson(Map<String, dynamic> json) {
  return Todo(
    id: json['id'],
    title: json['title'],
    completed: json['completed'],
  );
}
```