

01

Flutter

01. Flutter란

- 구글에서 개발한 오픈소스 UI 킷
- 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

MyApp: 논리적 앱 이름

MaterialApp: 앱의 전반적인 구조와 설정을 정의

MyHomePage: 논리적 페이지 이름

Scaffold: 앱의 하나의 화면에 해당하는
Widget들을 포함

05. Stateless Widget

- 변하지 않는 상태를 가지는 위젯
- 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 위젯

05. Stateless Widget

■ 특징

■ 불변성

- 위젯이 한번 생성되면 상태나 데이터가 변하지 않음
- 모든 데이터는 위젯의 생성자에게 전달받아 화면에 표시

■ 단순한 UI 구성

- 정적 콘텐츠(Text, Icon, Button등)을 표시

■ 효율적인 리렌더링

- 상태가 없기 때문에 Flutter의 효율적 트리 리렌더링 메커니즘을 활용

06. 기초 Widget

■ MaterialApp

- 앱의 초기화면, 테마 등 기본 설정 정의
- 주 속성 : title, theme, home, routes...

■ Scaffold

- 한 페이지에 해당하는 위젯들의 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. Fluttoast

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

```
Fluttoast.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
 - `MaterialPageRoute`를 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 위젯의 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, DrawerHeader

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** : 위젯의 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()

- 위젯이 처음 생성될 때 호출.
- 초기화 작업 수행 (ex: 애니메이션 초기화, API 호출 등).

■ didChangeDependencies()

- 위젯의 의존성이 변경되었을 때 호출.
- 주로 InheritedWidget과 연동된 상태가 변경되었을 때 사용.

■ build()

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

14. StatefulWidget의 생명주기(Lifecycle)

■ setState()

- 상태를 업데이트하고 UI를 다시 빌드.

■ deactivate()

- 위젯이 트리에서 제거되기 전에 호출.

■ dispose()

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

15. ListView

- 스크롤 가능한 위젯으로, 여러 위젯을 나열할 때 사용
- **ListView.builder**
 - 스크롤 중 필요한 아이템만 동적으로 빌드하여 성능을 최적화.

```
ListView.builder(  
  itemCount: 100, // 아이템 개수  
  itemBuilder: (context, index) {  
    return ListTile(  
      title: Text('Item $index'),  
    );  
  },  
)
```

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. Provider

- 상태 관리 솔루션 중 하나로, 간단하면서도 강력한 방식으로 상태를 관리하고 Widget Tree에 상태를 전달
- **ChangeNotifier**
 - 상태를 관리하고 알림을 제공하는 클래스
 - 상태 변경 시 `notifyListeners()`를 호출하여 상태를 사용하는 Widget들에게 공지

```
class Counter extends ChangeNotifier {  
  int _count = 0;  
  
  int get count => _count;  
  
  void increment() {  
    _count++;  
    notifyListeners(); // 상태 변경 알림  
  }  
}
```

17. Provider

■ ChangeNotifierProvider

- **ChangeNotifier**를 위젯 트리에 제공하는 역할
- 상태를 생성하고, 하위 **Widget**에서 이를 사용할 수 있도록 제공

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

■ 하위 **Widget**들이 상태값을 사용하는 방법

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

17. Provider

■ MultiProvider

- 여러 개의 Provider를 등록하여 사용

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

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 `then`을 사용하여 처리.
 - 에러가 발생하면 `catchError`를 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, error)
- 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'],  
  );  
}
```