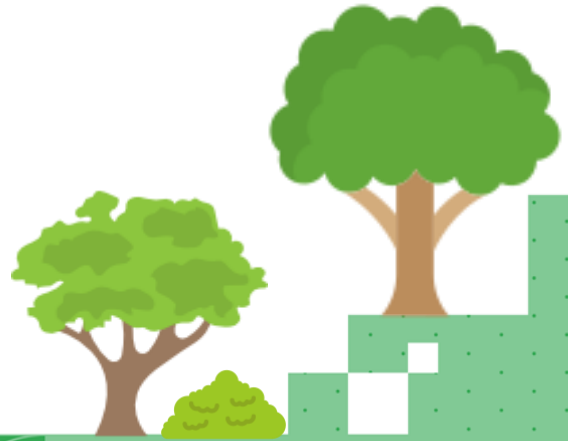
Google Developer Student Clubs
Hanyang University

Week 5.

김기진 (General - App Front)
컴퓨터소프트웨어학부20



Contents

 Json

 Future

 await, async

 http

API

- API는 애플리케이션 프로그래밍 인터페이스(Application Programming Interface)의 약자로,

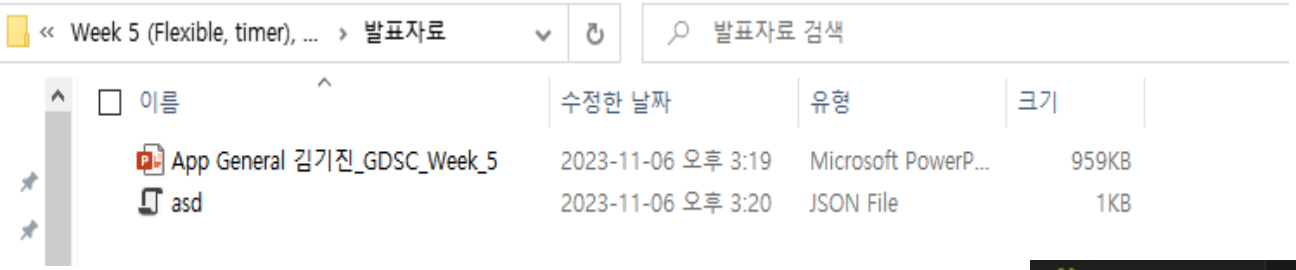
  두 소프트웨어 구성 요소가 서로 통신할 수 있게 하는 메커니즘

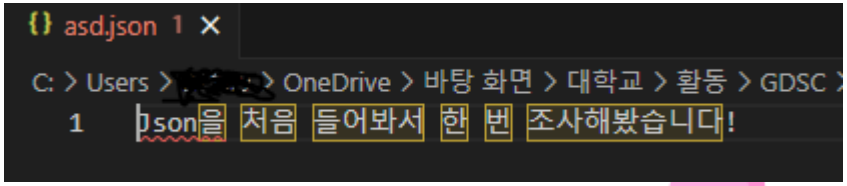
>> 다른 곳과 데이터 교환이 필요할 때 사용!

1. Json

Json(JavaScript Object Notation)을 칭하는 말로, 데이터의 한 형태

(.json) 형식으로 파일 저장 >> 그냥 **텍스트 파일**이라고 생각해도 무방!





1. Json

Json 파일의
텍스트 추출

추출한 텍스트
Map<String, dynamic>
자료형으로 변환

Map<String, dynamic>
자료형을 object (class)로
변환 >> fromJson 사용

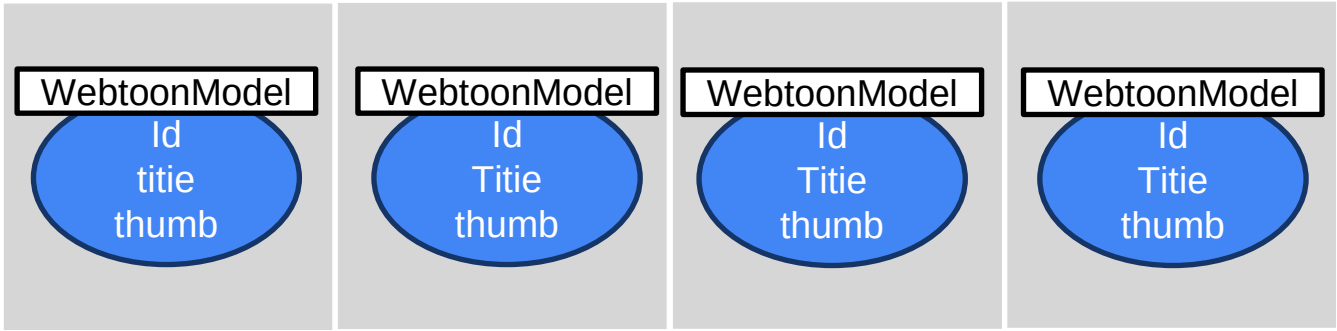
```
final url = Uri.parse('$baseUrl/$today'); // json 자료를 변환
```

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

class WebtoonModel {
  final String title, thumb, id;

  WebtoonModel.fromJson(Map<String, dynamic> json)
      : title = json['title'],
        thumb = json['thumb'],
        id = json['id'];
  // name constructor
}
```

Google Developer Student Clubs
Hanyang University

1. Json

[{"id":"814543","title":"너는 그냥 개그만화나 그려라","thumb":"https://image-comic.p



2. Future

Future<‘자료형’> : return 때 ‘자료형’ or error의 값을 가짐

>>> 비동기 동작을 위해 사용

2. Future

● 동기 처리
: 모든 동작을 코드 차례대로 수행

>> api를 통해 데이터를 가져오는 동안 그 다음 코드는 진행되지 않고 정지됨

● 비동기 처리
: 어떤 동작이 수행 중이더라도 다음 코드를 수행

>> api를 통해 데이터를 가져오는 동안, 그 다음 코드를 수행
>> 사용자에게 코드가 수행 중에 있다고 확인시킬 수 있음!

2. Future

FutureBuilder
: 비동기 처리를 위해 사용하는 widget

parameter : future, builder

future : Future type으로 주로 어떤 시간이 걸린 후에 데이터가 생기는 값을 많이 사용

builder : contex와 snapshot이 있는데, snapshot으로 future 의 상태에 접근 가능

```
body: FutureBuilder(
  future: webtoons, // FutureBuilder에서 따로 await할 필요 없음
  builder: (context, snapshot) {
    //context는 상위 위젯트리의 정보, snapshot은 future의 결과값으로 future의 상태에 접근할 수 있는 parameter
    if (snapshot.hasData) {
```

3. await, async

await 키워드를 사용하기 위해서는 async 함수여야 하고 async 함수이기 위해선 Future 값을 return 해야 함!

```dart
13  static Future<List<WebtoonModel>> getTodaysToons() async {
14    // await를 실행하기 위해 함수를 asynchronous function으로 
15    List<WebtoonModel> webtoonInstances = [];
16    final url = Uri.parse('$baseUrl/$today'); // json 자료를 
17    final response = await http.get(url);
```

```dart
static Future<WebtoonDetailModel> getToonById(String id) async {
  final url = Uri.parse('$baseUrl/$id');
  final response = await http.get(url);
```

```dart
static Future<List<WebtoonEpisodeModel>> getLatesetEpisodesById(
    String id) async {
  List<WebtoonEpisodeModel> episodesInstances = [];
  final url = Uri.parse('$baseUrl/$id/episodes');
  final response = await http.get(url);
```

3. await, async

async
: 비동기 처리를 위한 키워드로 특별한 역할은 없음,,

>> 해당 함수가 Future이고 특정 수행 뒤에 값이 바뀔 수 있음을 명시해주는 역할!

3. await, async

await
: 이 키워드가 있는 함수는 해당 부분이 완료될 때 까지 코드 진행 x

>> await 수행 동안 대기중인 함수는 다른 Future 값을 return함

>> await 수행이 완료된 후 함수는 또 다른 Future 값을 return 할 수 있음

3. await, async

```
body: FutureBuilder(
  future: webtoons, // FutureBuilder에서 따로 awa
  builder: (context, snapshot) {
    //context는 상위 위젯트리의 정보, snapshot은 f
    if (snapshot.hasData) {
      return Column(
        children: [
          const SizedBox(
            height: 50,
          ), // SizedBox
          Expanded(child: makeList(snapshot)),
        ],
      ); // Column
    }
    return const Text('Loading....');
  },
), // FutureBuilder
```

데이터가 있으면 Column return

데이터가 없으면 Text return

4. HTTP

HTTP
: 웹끼리 정보를 주고 받을 수 있도록 하는 규칙

4. HTTP

요청 (request)

사용자(유저)

응답 (response)

수용자(서버)

4. HTTP

유저는 여러 가지 명령어로 서버에 파일을 요청, 등록 등의 동작을 할 수 있음

>> PUT : 서버에 데이터 등록

>> GET : 서버에 있는 데이터 요청

4. HTTP

서버의 응답 상태를 표시하는 Status Code가 존재

>> 200(OK)
: 정상적으로 유저에게 데이터 보냄

>> 400(Bad Request)
: 이상한 요청이라 서버에서 뭘 바라는지 모름
(명령어를 잘못 썼다거나!)

>> 404(Not Found)
: 유저가 요청한 파일이 서버에 없음

4. HTTP

```dart
static Future<WebtoonDetailModel> getToonById(String id) async {
  final url = Uri.parse('$baseUrl/$id');
  final response = await http.get(url);
  if (response.statusCode == 200) {
    final webtoon = jsonDecode(response.body);
    return WebtoonDetailModel.fromJson(webtoon);
  }
```

```dart
final response = await http.get(url);
```

유저가 http 규칙으로 url에 데이터 요청 (get)

```dart
if (response.statusCode == 200) {
```

서버가 보낸 응답에 있는 Status Code로
정상적으로 데이터를 받았는지 확인

Week 5.

과제

Google Developer Student Clubs
Hanyang University

1번 문제

`async/await` 를 사용하여 로딩 구현해보기

1. Flutter 화면에 버튼을 추가합니다.
2. 버튼을 클릭하면 `async/await` 를 사용하여 로딩을 구현해보세요.

ex) 버튼을 누르면 "로딩중" -> 로딩이 완료되면 "로딩 완료" 메시지를 화면에 표시

강의 #6.5 5:50~5:56 부분처럼 만드시면 됩니다.

작성한 코드를 복사하여 코드블록 태그 안에 넣어주세요.

```
class MyButton extends StatefulWidget {
  const MyButton({super.key});

  @override
  _MyButtonState createState() => _MyButtonState();
}

class _MyButtonState extends State<MyButton> {
  String buttonText = '로딩 시작';

  Future<void> loadButton() async {
    setState(() {
      buttonText = '로딩중...';
    });

    await Future.delayed(const Duration(seconds: 3));

    setState(() {
      buttonText = '로딩 완료';
    });
  }

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: loadButton,
      child: Text(buttonText),
    );
  }
}
```


Thanks to ChatGPT..

```dart
class MyButton extends StatefulWidget {
  const MyButton({super.key});

  @override
  _MyButtonState createState() => _MyButtonState();
}

class _MyButtonState extends State<MyButton> {
  String buttonText = '로딩 시작';

  Future<void> loadButton() async {
    setState(() {
      buttonText = '로딩중...';
    });

    await Future.delayed(const Duration(seconds: 3));

    setState(() {
      buttonText = '로딩 완료';
    });
  }

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: loadButton,
      child: Text(buttonText),
    );
  }
}
```

ElevatedButton : 입체감 있는 버튼 widget

버튼을 누를 시 3초동안 로딩중…

이후 로딩완료 나오도록 설정

2번 문제

`FutureBuilder` 를 사용하여 로딩 구현해보기

1. Flutter 화면에 버튼을 추가합니다.
2. 버튼을 클릭하면 `FutureBuilder` 를 사용하여 로딩을 구현해보세요.

ex) 버튼을 누르면 "로딩중" -> 로딩이 완료되면 "로딩 완료" 메시지를 화면에 표시

강의 #6.5 5:50~5:56 부분처럼 만드시면 됩니다.

작성한 코드를 복사하여 코드블록 태그 안에 넣어주세요.

```
class _MyButtonState extends State<MyButton> {
  Future<String>? futureData;

  Future<String> createFutureData() async {
    await Future.delayed(const Duration(seconds: 3));
    return '로딩 완료';
  }
```

```
: FutureBuilder<String>(
    future: futureData,
    builder:
        (BuildContext context, AsyncSnapshot<String> snapshot) {
      if (snapshot.connectionState == ConnectionState.waiting) {
        return const Text('로딩 중..');
      }
      throw Error();
```


Thanks to ChatGPT..

```dart
class _MyButtonState extends State<MyButton> {
  Future<String>? futureData;

  Future<String> createFutureData() async {
    await Future.delayed(const Duration(seconds: 3));
    return '로딩 완료';
  }
```

```dart
Widget build(BuildContext context) {
  return Column(
    mainAxisAlignment: MainAxisAlignment.center,
    children: <Widget>[
      (futureData == null)
          ? ElevatedButton(
              child: const Text('로딩 시작'),
              onPressed: () {
                setState(() {
                  futureData = createFutureData();
                });
              },
            )
          : FutureBuilder<String>(
              future: futureData,
              builder:
                  (BuildContext context, AsyncSnapshot<String> snapshot) {
                if (snapshot.connectionState == ConnectionState.waiting) {
                  return const Text('로딩 중..');
                }
                throw Error();
              }
```

초기 상태는 '로딩 시작'

이후 버튼을 누르면 createFutureData 실행

createFutureData가 null이 아니게 되므로 밑에 있는 FutureBuilder 실행

Waiting에 있으면 '로딩 중' 출력

createFutureData가 완료되면 '로딩완료'를 return 해서 return 값이 바뀌어 화면에 출력