WaffleStudio 2023
Rookies 21.5 Seminar

Android Seminar 2
2023.10.06 (금)

Instructor : 양주현 (@JuTaK97) / TA : 송동엽 (@eastshine2741)

오늘 배울 것

- Gradle 이란 무엇인가?
- 네트워크 통신을 하는 방법
- DI (Dependency Injection) 이란 무엇일까?
  - Dagger Hilt 사용법
- 비동기 프로그래밍 기초

Gradle (1)

- 우리의 앱은 누가 어떻게 빌드해 주고 있나요?

  초록색 ▶ 버튼만 누르면 뭔가 막 돌아가고 빌드되고 앱이 실행되었는데… How??

- 우리가 지금까지 사용해 본 라이브러리는 뭐가 있나요?

  RecyclerView, ViewModel, …

  라이브러리 코드는 그냥 import해서 사용했는데, 누가 어떻게 가져와 주나요?

- (플레이스토어에서 본 적이 있으실 겁니다) 앱의 최소 OS 버전은 어떻게 정하나요?

등등, 어디선가 많은 작업들을 해 주고 있습니다.

Gradle (2)

지금은 깊이 다루지 않겠습니다. 더 알고 싶으신 분은 여기로 〉 https://gradle.org/guides

먼저 우리의 앱의 프로젝트 구조를 살펴보면 “gradle” 이름이 붙은 게 몇 개 있는데…

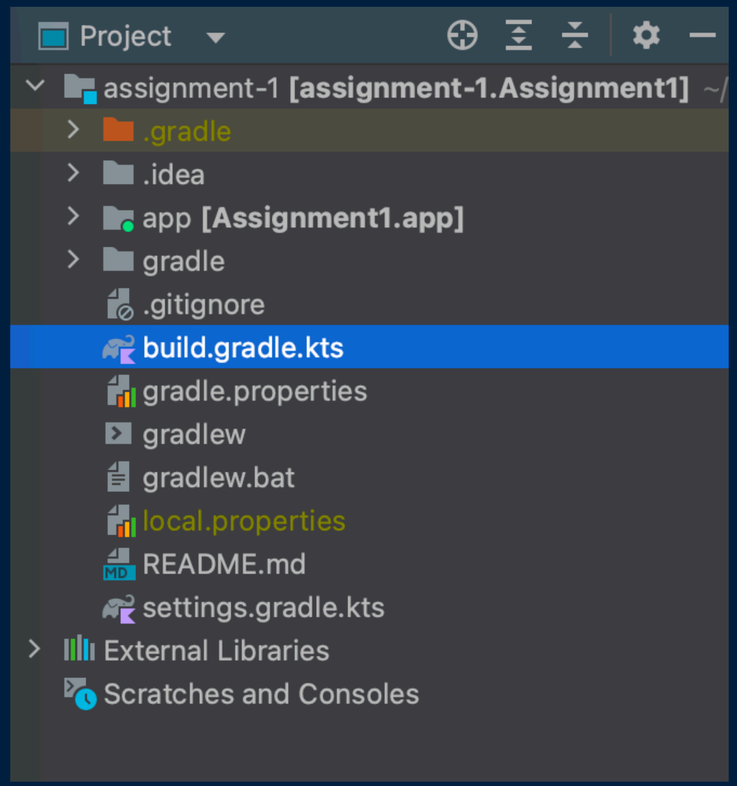
가장 root에는

- build.gradle.kts
- settings.gradle.kts 가 있고,

그리고 app 폴더에

- build.gradle.kts 가 있습니다.

지금은 이것들만 간단히 살펴봅시다!



Gradle (3)

```
// Top-level build file where you can add configuration options common to all sub-projects/modules.
plugins {
    id("com.android.application") version "8.2.0-beta01" apply false
    id("org.jetbrains.kotlin.android") version "1.9.0" apply false
}
```

우리의 앱은 아주 단순하기 때문에 “모듈” 이 아직은 단 하나이지만, 앱의 규모가 커지면 앱을 모듈화해서 여러 개로 쪼개게 됩니다. (지금은 여기까지만 알고 넘어갑시다!)

루트 디렉토리의 build.gradle 파일은 모든 모듈, 즉 프로젝트 전체에 대해 적용되는 설정을 정의하는 곳입니다.

com.android.application은 앱을 빌드하는 데 필요한 것들을 제공하는 플러그인이고, org.jetbrains.kotlin.android는 안드로이드 개발에 코틀린을 사용할 수 있도록 코틀린을 지원해 주는 플러그인입니다. (대략 이정도만 알고 넘어가도 충분합니다!)

Gradle (4)

app 폴더의 build.gradle은 해당 모듈의 빌드 구성 및 설정을 하는 곳입니다. 크게 보면 android {} 블록과 dependencies {} 블록으로 구성되어 있습니다.

android 블록에서는 앱의 기본 설정을 정의하고 있습니다.

- minSDK (앱이 지원하는 최소 OS), targetSDK와 compileSDK, 그리고 앱의 버전 코드를 설정합니다.

- buildType (디버그 모드 / 릴리즈 모드)을 설정할 수 있고, 릴리즈 모드로 빌드할 때 각종 앱 난독화, 최적화, 축소 등을 설정할 수 있습니다.

(지금은 그냥 그렇구나~ 하고 넘어가도 충분합니다!)

```
android {
    namespace = "com.example.assignment1"
    compileSdk = 34

    defaultConfig {
        applicationId = "com.example.assignment1"
        minSdk = 24
        targetSdk = 34
        versionCode = 1
        versionName = "1.0"
    }

    buildTypes {
        release {
            isMinifyEnabled = false
            proguardFiles(
                getDefaultProguardFile(
                    “proguard-android-optimize.txt"
                    ), “proguard-rules.pro"
            )
        }
    }
```

Gradle (5)

필요한 라이브러리들은 dependency 블록에 선언합니다. 개발하면서 필요한 라이브러리가 생기면, 버전과 함께 써놓으면 됩니다.
그러면 그 의존성은 누가 해결해 줄까요? (= 누가 필요한 코드을 찾아서 구해다 줄까요?)

settings.gradle 에는 프로젝트 구조(어떤 모듈들이 있는지…)를 정의하는 것과 더불어서, 각종 의존성(dependency)은 어떻게 해결해야 하는지 명시합니다.

즉 “라이브러리들은 여기서 찾아서 갖다 써!” 를 settings.gradle에 정의해 놓으면 gradle은 이를 참조해서 필요한 라이브러리를 다운로드 받아서 가져와줍니다.

라이브러리는 주로 온라인에 배포되어 있는데, maven 등의 저장소가 대표적입니다. 여러분이 첫 앱을 빌드할 때 잔뜩 다운받았던 것이 이것들입니다!

네트워크 통신 (1)

서버와 통신하기 위해 클라이언트는 무엇을 해야 할까요?

- 요청을 보낸다.
- 기다린다.
- 응답이 온다.
- 응답 데이터를 읽고 가공한다.
- (여기부턴 이미 배운 것) 데이터를 UI로 바꾸어 앱에 표시한다.

네트워크 통신 (2)

우리는 인터넷을 통해 통신하고, 서로 약속된 형식 (프로토콜)을 따라 데이터를 주고받습니다. 그리고 가장 대표적인 인터넷 통신 프로토콜이 바로 HTTP입니다.

HTTP 프로토콜 요청은 다음과 같은 핵심 구성 요소로 이루어져 있습니다.

- 메서드 :

  크게 GET, POST, PUT, DELETE의 4가지 메서드가 존재합니다. 서버가 어떤 작업을 수행할 지 지정하는 부분입니다.

- URL : 프로토콜(http, https 등), 호스트 (서버의 주소), 포트 번호, 경로 등의 정보입니다.

- 쿼리 : 추가적인 정보를 URL에 담아 전달하는 부분입니다. 주로 검색 request에서 사용됩니다.

- 바디 : 주고받는 실제 데이터입니다. 여러가지 데이터 형식을 사용할 수 있지만, 주로 JSON 형식을 사용합니다.

네트워크 통신 (3)

HTTP 프로토콜 응답은 다음과 같은 핵심 구성 요소로 이루어져 있습니다.

- Status Code

  성공 시 2XX (200 OK, 201 Created 등)

  실패 시 (클라 잘못) : 400 Bad Request, 404 Not Found, 403 Forbidden

  실패 시 (서버 잘못) : 500, 502 Bad Gateway, 503 Service Unavailable

- Body : HTTP 요청과 동일

네트워크 통신 (4)

시스템 프로그래밍 수업을 들으면 배우겠지만…
IO는 시스템의 영역입니다. 로우레벨의 시스템 함수를 이용해야 IO에 접근할 수 있습니다.

매우 복잡하고 어렵기 때문에, 라이브러리를 사용합니다.

OkHttp 라이브러리는

HTTP 요청을 보내고 / 응답을 받고 / 커넥션을 관리하고 / 캐시, 인터셉터, 인증, 보안 등등…

이런 많은 (귀찮은) 것들을 대신 해줍니다.

직접 해봅시다!

```
//  build.gradle.kts (:app)
implementation("com.squareup.okhttp3:okhttp:4.10.0")
```

네트워크 통신 (5)

어플리케이션 전역적으로 사용하기 위해 Application에 OkHttpClient를 선언합니다.
Application을 아래와 같이 만든 후 AndroidManifest에 name을 등록해 줍니다.

```
// MyApplication.kt
class MyApplication: Application() {
    val client = OkHttpClient()
}
```

```
val request = Request.Builder()
    .url("http://ec2-13-209-69-159.ap-northeast-2.compute.amazonaws.com:8000/myapp/v1/")
    .build()
val response = (application as MyApplication).client.newCall(request).execute()
```

OkHttp를 이용하면 이렇게 간단하게 HTTP 통신을 구현할 수 있습니다.

네트워크 통신 (6)

그런데 항상 URL을 저렇게 넣어 줘야 하나요? POST, DELETE 등 메서드는요?
쿼리는요? 더 복잡한 데이터 클래스는 body로 어떻게 주고받죠?

-> 오픈 소스 라이브러리인 Retrofit을 사용하면 쉽게 RESTful API를 만들 수 있습니다.

앞서 body는 json이라는 데이터 형식으로 주고받는다고 했습니다. 따라서 data class와 json 간의 상호 변환이 필요합니다.

그리고 오픈 소스 라이브러리 Moshi는 서버에게 보낼 data class를 json 형식으로 직렬화 (serialization) 해 주고, 서버가 보내 준 json을 data class로 역직렬화 (deserialization) 해 줍니다.

네트워크 통신 (7)

1. 복잡한 HTTP 통신 (로우레벨) 을 간편하게 해주는 것 : OkHttp

2. API 인터페이스를 구현해 주는 것 : Retrofit

3. Data class와 JSON 간의 직렬화/역직렬화를 해 주는 것 : Moshi

필요한 dependency는 다음과 같습니다.

```
// OkHttp
implementation("com.squareup.okhttp3:okhttp:4.10.0")

// Retrofit
implementation("com.squareup.retrofit2:retrofit:2.9.0")
implementation("com.squareup.retrofit2:converter-moshi:2.9.0")

// moshi
implementation("com.squareup.moshi:moshi:1.14.0")
implementation(“com.squareup.moshi:moshi-kotlin:1.14.0")
```

네트워크 통신 구현 (1)

```
private val client = OkHttpClient.Builder().build() // OkHttpClient 객체 생성
private val moshi = Moshi.Builder()  // Moshi 객체 생성
    .add(KotlinJsonAdapterFactory()) // JSON 직렬화/역직렬화를 해 주는 Adapter 추가
    .build()
private val retrofit = Retrofit.Builder() // Retrofit 객체 생성
    .client(client) // okhttp 장착
    .baseUrl("http://ec2-13-209-69-159.ap-northeast-2.compute.amazonaws.com:8000/") // 서버 base url
    .addConverterFactory(MoshiConverterFactory.create(moshi)) // moshi 장착
    .build()
```

이제 네트워크 통신을 위한 도구는 모두 준비되었습니다.

우리가 사용할 API를 정의하고, 이 도구들을 이용해서 사용해 봅시다.

네트워크 통신 구현 (2)

```
data class Person(
    val name: String,
    val age: String,
)
```

요청을 보내면 오른쪽과 같이 생긴 정보를 보내 주는 API가 있습니다.

요청을 보낼 서버의 url이 “${base-url}/myapp/info” 일 때, 다음과 같이 인터페이스를 만들어 줍니다.

```
interface MyRestAPI {
    @GET(“/myapp/v1/info")
    fun getPerson(): Call<Person>
}
```

Call을 import해 올 때 주의해야 할 게, 정말 다양한 라이브러리에서 Call이라는 이름의 무언가를 갖고 있습니다. 반드시 Retrofit2의 Call을 import해야 합니다.

네트워크 통신 구현 (3)

DTO (data transfer object)로 사용할 data class는 추가적인 표시를 해 줘야 합니다.

```
@JsonClass(generateAdapter = true)
data class Person(
    @Json(name = "name") val name: String,
    @Json(name = "age") val age: String,
)
```

이렇게 @ (annotation)을 달아 줘야 우리의 도구들이 이것을 인식하고 자동으로 필요한 나머지 코드들을 만들어 줄 수 있습니다.

마지막으로, 인터페이스를 Retrofit에게 전달해 주면 인터페이스를 그대로 구현해 줍니다!

```
// MyApplication.kt
val restAPI = retrofit.create(MyRestAPI::class.java)
```

네트워크 통신 구현 (4)

Application에 정의해 놓은 api를 가져와서 getPerson 함수를 부르면, Person 정보가 body에 들어 있는 HTTP 응답이 Call이라는 것에 래핑되어서 반환됩니다.

Call 객체에서 정보 (통신 성공이면 body 정보 등, 통신 실패면 에러 메시지 등)은 다음과 같이 enqueue 함수를 이용해 꺼낼 수 있습니다.

```
val api = (application as MyApplication).restAPI
val response = api.getPerson()
response.enqueue(object : Callback<Person> {
    override fun onFailure(call: Call<Person>, t: Throwable) {
        // 실패했을 때 할 동작
    }
    override fun onResponse(call: Call<Person>, response: Response<Person>) {
        val responseBody = response.body()
        // ...
    }
})
```

네트워크 통신 구현 (5)

실행해 봅시다! button의 click listener에서 api.getPerson을 불러 보면…

```
E FATAL EXCEPTION: OkHttp Dispatcher
  Process: com.example.assignment2, PID: 16728
  java.lang.SecurityException: Permission denied (missing INTERNET permission?)
      at java.net.Inet6AddressImpl.lookupHostByName(Inet6AddressImpl.java:150)
      at java.net.Inet6AddressImpl.lookupAllHostAddr(Inet6AddressImpl.java:103)
      at java.net.InetAddress.getAllByName(InetAddress.java:1152)
      at okhttp3.Dns$Companion$DnsSystem.lookup(Dns.kt:49)
```

너무나 친절한 메시지가 있습니다. AndroidManifest.xml에 다음을 추가해서 우리 앱이 인터넷 연결 권한을 갖게 하고, HTTP 통신을 할 수 있게 허용해 줍니다.

```
<uses-permission android:name="android.permission.INTERNET"/>

<application
    android:name=".MyApplication"
    ...
     android:usesCleartextTraffic="true">
```

Dependency Injection (1)

우리는 필요한 도구들(retrofit, moshi 등)을 Application에 변수로 선언해 놨습니다.
이 방식에는 몇 가지 문제점이 존재하는데,

1. 객체들이 Application에 종속되게 되고, Retrofit과 Application이라는 서로 관련없는 두 코드가 불필요하게 결합되게 됩니다.

2. Moshi 같은 것들은 딱 하나의 인스턴스만 있으면 되기 때문에 싱글톤 패턴을 사용하는 것이 좋은데, 지금같은 방식으로는 이들이 유일한 인스턴스로 존재하는 것을 보장할 수 없습니다.

3. 각 객체 간에 OkHttp, Moshi ➡️ Retrofit ➡️ MyRestAPI 라는 종속 관계가 존재하고 있는데, 단순하게 차례로 선언하는 방식으로는 객체 간의 의존성을 관리하는 것이 어렵습니다.

DI (Dependency Injection) 방식을 사용하면 이를 해결할 수 있고, 오픈소스 라이브러리인 Dagger Hilt 를 안드로이드에서 DI 도구로 주로 사용합니다.

Dependency Injection (2)

Dagger는 구글에서 개발한 DI 프레임워크이고, Hilt는 Dagger를 기반으로 만들어진 안드로이드 앱을 위한 DI 프레임워크입니다.

필요한 dependency는 아래와 같습니다.

```
// build.gradle.kts (:app)
plugins {
    id("kotlin-kapt")
    id("dagger.hilt.android.plugin")
}

dependencies {
    implementation(“com.google.dagger:hilt-android:2.47")
    kapt("com.google.dagger:hilt-android-compiler:2.47")
}
```

```
// build.gradle.kts (:Project)
buildscript {
    dependencies {
        classpath("com.google.dagger:hilt-android-gradle-plugin:2.47")
    }
}
```

Dependency Injection (3)

NetworkModule.kt 같은 파일을 새로 만들고 다음과 같이 선언해 줍니다.

```
@Module
@InstallIn(SingletonComponent::class) // 앱 전체 수명 주기 동안 단일 인스턴스 (싱글톤)
class NetworkModule {
    @Provides // OkHttpClient 타입의 객체를 어떻게 만드는 지 dagger에게 알려 준다.
    fun provideOkHttpClient(): OkHttpClient {
        return OkHttpClient.Builder().build()
    }
}
```

Dagger 모듈에 대해서 더 자세히 공부하기 : [링크]

Dependency Injection (4)

마찬가지로, 앞서 살펴본 의존성 관계 순서대로 하나씩 @Provides를 작성해 줍니다.

```
@Provides
fun provideMoshi(): Moshi {
    return Moshi.Builder().add(KotlinJsonAdapterFactory())
        .build()
}
```

```
@Provides
fun provideOkHttpClient(): OkHttpClient {
    return OkHttpClient.Builder().build()
}
```

```
@Provides
fun provideRetrofit(
    okHttpClient: OkHttpClient,
    moshi: Moshi,
): Retrofit {
    return Retrofit.Builder().baseUrl("server-url")
        .client(okHttpClient)
        .addConverterFactory(MoshiConverterFactory.create(moshi)
        .build()
}
```

```
@Provides
fun MyRestAPI(
    retrofit: Retrofit,
): MyRestAPI {
    return retrofit.create(MyRestAPI::class.java)
}
```

Dependency Injection (5)

Dagger가 안드로이드 컴포넌트들을 알아보고 의존성을 주입할 수 있도록 Activity와 Application에도 어노테이션이 필요합니다.

이제 api가 필요한 곳에서 다음과 같이 주입시킬 수 있습니다.

```
@AndroidEntryPoint
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    @Inject
    lateinit var api: MyRestAPI
```

```
@HiltAndroidApp
class MyApplication: Application()
// MyApplication 안에 선언했던 것들은 이제 모두 지웁니다.
```

이제 dagger가 각 객체의 의존성을 파악하고, 컴파일 타임에 객체 생성 및 주입 코드를 자동으로 생성해 줍니다.

Async Programming (1)

다시 처음으로 돌아와서…

서버에게 요청을 보내고 응답이 오기까지는 시간이 걸립니다.

앞 슬라이드의 코드를 다시 보면,

서버와 통신하기 위해 클라이언트는 무엇을 해야 할까요?

- 요청을 보낸다.
- 기다린다.
- 응답이 온다.
- 정보를 읽고 가공한다.
- (여기부턴 이미 배운 것) 데이터를 UI로 바꾸어 앱에 표시한다!

```
val request = Request.Builder().url("https://server-url/").build()
val response = (application as MyApplication).client.newCall(request).execute()
```

이 코드를 실제로 setOnClickListener {} 안에서 수행하면 크래시가 발생합니다.

```
E FATAL EXCEPTION: main
  Process: com.example.assignment2, PID: 20268
  android.os.NetworkOnMainThreadException
      at android.os.StrictMode$AndroidBlockGuardPolicy.onNetwork(StrictMode.java:1675)
      at java.net.Inet6AddressImpl.lookupHostByName(Inet6AddressImpl.java:115)
      at java.net.Inet6AddressImpl.lookupAllHostAddr(Inet6AddressImpl.java:103)
```

Async Programming (2)

우리가 작성한 코드는 기본적으로 UI 스레드에서 실행됩니다.

UI 스레드는 터치 이벤트 등의 유저 상호작용과 UI 그리기 등을 담당합니다. UI 스레드에서 네트워크 관련 작업을 진행하려고 하면, 시스템은 NetworkOnMainThreadException을 발생시킵니다. 또한, UI 스레드에서 무한 루프가 돌거나 시간이 오래 걸리는 작업을 진행하면 ANR(애플리케이션 응답 없음)이 발생합니다.

예를 들어, 서버의 응답을 오른쪽과 같이 기다리면 ANR이 발생합니다.

```
val connection = makeIORequest(req)
while (true) {
  if (connection.isArrived()) {
    break
  }
  // wait...
}
val response = connection.response
```

하지만 IO는 일반적으로 많은 시간이 소요됩니다.

- 외부(서버 or DB)에서 처리 시간이 길 수도 있고,
- 인터넷이 느릴 수도 있습니다.

어떤 방법이 있을까요?

Async Programming (3)

해결책 1. 새 스레드를 생성

UI 스레드가 아닌 다른 스레드를 만들고, 그 곳에서 IO를 진행할 수 있습니다.

```
binding.text.setOnClickListener {
    thread {
        val request = Request.Builder().url(“server-url").build()
        val response = OkHttpClient().newCall(request).execute()
    }
```

하지만 스레드는 비싼 자원이기 때문에 각 네트워크 요청마다 스레드를 만드는 것은 큰 낭비입니다.
또한 스레드를 “잘” 사용하고 관리하는 것은 아주 어려운 일입니다.

발생하는 문제들 : Race condition, 데드락, context switch로 인한 성능 저하, 코어 수의 한계 등등… 궁금하면 찾아보세요

Async Programming (4)

해결책 2. 콜백 (CallBack)

앞서 Retrofit의 예제에서는 Call<Person> 으로 응답을 받아서 enqueue 함수에 onFailure과 onFailure 두 개의 함수를 넣어주었습니다.

따라서 api.getPerson()은 client.newCall(request).execute() 과는 다르게, UI 스레드를 틀어막는 (blocking) 함수가 아닙니다. 시간이 지나고 응답이 왔을 때 나중에 수행할 콜백 함수를 전달하기 때문입니다.

```
binding.text.setOnClickListener {
    val response = api.getPerson()
    response.enqueue(object : Callback<Person> {
        override fun onFailure(call: Call<Person>, t: Throwable) {
            Toast.makeText(this@MainActivity, "fail ㅠㅠ", Toast.LENGTH_SHORT).show()
        }
        override fun onResponse(call: Call<Person>, response: Response<Person>) {
            Toast.makeText(this@MainActivity, response.body().toString(), Toast.LENGTH_SHORT).show()
        }
    })
```

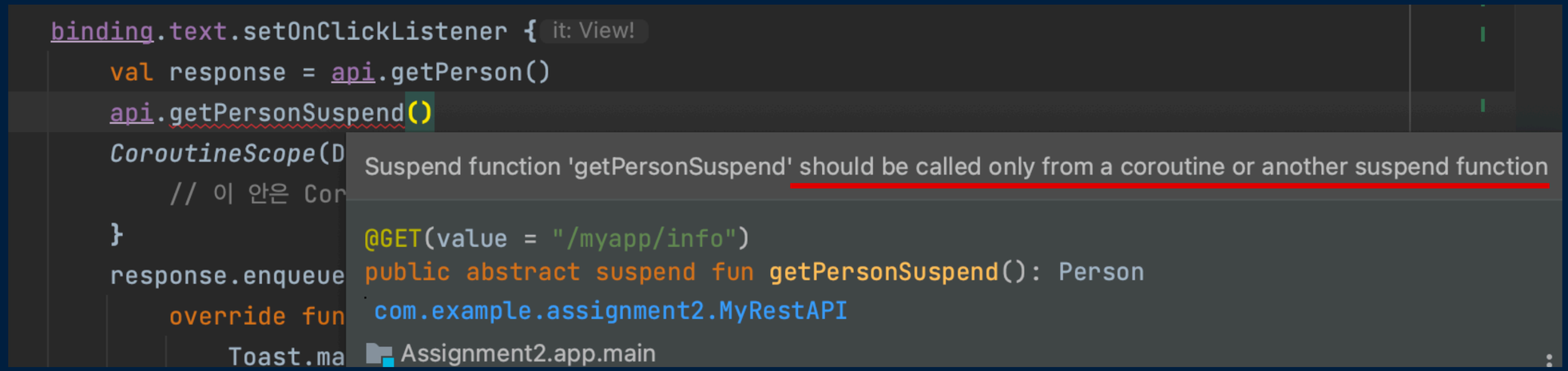
Async Programming (5)

해결책 3. 코루틴 (Coroutine)

코루틴은 비동기 작업을 아주 편하게 사용할 수 있는 도구입니다.

함수를 suspend fun 으로 만들면, 이 함수는 이제부터 “중단 가능한 함수”가 됩니다. 중단 가능한 함수는 일반적인 함수와 조금 다르게 컴파일되고, “잠시 멈췄다가” “나중에 이어서” 실행할 수 있게 됩니다.

따라서 suspend fun은 아무 데서나 사용할 수 없습니다.



Async Programming (6)

코루틴은 여러 방법으로 만들 수 있고, 가장 일반적인 방법은 다음과 같습니다.

```
CoroutineScope(Dispatchers.IO).launch {
    api.getPersonSuspend()
}
```

Dispatchers.Main은 이 코루틴을 IO 스레드에서 실행한다는 의미입니다. UI를 업데이트하는 메인 스레드와 다른 스레드에서, 즉 백그라운드에서 네트워크 통신 함수를 수행하게 됩니다.

만약 여기서 UI 작업을 하면 어떻게 될까요? Toast를 띄워봅시다.

```
CoroutineScope(Dispatchers.IO).launch {
    val response = api.getPersonSuspend()
    Toast.makeText(this@MainActivity, response.toString(), Toast.LENGTH_SHORT).show()
}
```

Async Programming (7)

```
E FATAL EXCEPTION: DefaultDispatcher-worker-2
  Process: com.example.assignment2, PID: 20848
  java.lang.NullPointerException: Can't toast on a thread that has not called Looper.prepare()
      at com.android.internal.util.Preconditions.checkNotNull(Preconditions.java:167)
      at android.widget.Toast.getLooper(Toast.java:187)
```

크래시가 발생합니다. UI 스레드가 아닌, DefaultDispatcher-worker-2라는 이상한 스레드에서 토스트를 띄우려고 했기 때문입니다.

아래와 같이 withContext를 이용하면 코루틴 내에서 수행되는 스레드를 바꿀 수 있습니다.

```
CoroutineScope(Dispatchers.IO).launch {
    val response = api.getPersonSuspend()
    withContext(Dispatchers.Main) {
        Toast.makeText(this@MainActivity, response.toString(), Toast.LENGTH_SHORT).show()
    }
}
```

Async Programming (8)

코루틴 속에서 네트워크 통신을 하고 정보를 받아오고, 우리는 그 데이터에 따라 UI를 그립니다.

```
CoroutineScope(Dispatchers.IO).launch {
    val response = api.getPersonSuspend()
    withContext(Dispatchers.Main) {
        binding.text.text = "이름 : ${response.name}, 나이 : ${response.age}"
    }
}
```

복습) MVVM 구조는 위처럼 뷰와 로직이 합쳐져 있는 형태를 권장하지 않습니다. UI에 그릴 정보를 어떻게 받아오는지 View는 몰라야 합니다. 우리는 LiveData를 사용해서 이를 분리했습니다.

ViewModel에는 viewModelScope라는 뷰모델의 생명 주기에 묶여 있는 특별한 CoroutineScope가 존재합니다. 이를 활용하면 다음과 같이 MVVM 구조에서 비동기 함수를 사용할 수 있습니다.

Async Programming (9)

```
  // Activity의 onCreate
 viewModel.data.observe(this) {
     binding.text.text = "이름 : ${it.name}, 나이 : ${it.age}"
 }
 binding.text.setOnClickListener {
     viewModel.fetchPersonAndUpdateText()
 }
```

```
  // ViewModel
fun fetchPerson() {
    viewModelScope.launch(Dispatchers.IO) {
        val person = api.getPersonSuspend()
        withContext(Dispatchers.Main) {
            _data.value = person // _data는 뷰모델이 갖고 있는 MutableData
        }
    }
}
```

참고 자료들

- Gradle 가이드
  - 내용이 방대합니다. 저도 잘 모릅니다. 지금 알 필요는 없습니다. ~~스터디 환영~~
- Dagger 기본사항, Android 앱에서 Dagger 사용
  - 역시 내용이 방대합니다. 세미나에서 설명한 것으로 충분하긴 하나, 더 알면 무척 좋습니다. ~~역시 스터디 환영~~
- Hilt를 사용한 종속 항목 삽입 : 오늘 세미나 내용이 포함되어 있습니다.
- 코루틴 튜토리얼 : 쉬운 codelab입니다. 따라해 보면 좋습니다.
- 코틀린 기초 : 공식 가이드 문서입니다.
- 수명 주기 인식 구성요소와 함께 Kotlin 코루틴 사용
  - LiveData와 viewModelScope의 발전된 활용 형태를 공부할 수 있습니다.

과제 공지

과제 3: 단어장 앱 만들기

- 과제 테마 : Network(Http), DI, Asynchronous Programming (Coroutine)

- 과제 3 기한 : 10월 26일 목 자정 (목~금 넘어가는 밤)

- 과제 링크 : https://github.com/wafflestudio/seminar-2023-android-assignment/blob/assignment3/assignment-3/README.md