SQLite

I. SQLite

1. SQLite란?

A. 클라이언트 어플리 케이션에 주로 사용하는 경량 내장형 DBMS

- 관계형 데이터베이스
- 안드로이드, ios, 그리고 웹 브라우저 등에서 사용 → 안드로이드의 경우 프레임워크에 기본 내장(바로 사용 가능)

B. 기기에 자료를 영구로 저장해야 할 경우 적용

- 휴대폰 내부에 파일로 DB가 만들어짐
- 라이브러리 형태로 호출하여 사용(클래스 import)

2. 사용 절차

1) SQLiteOpenHelper 상속 클래스 작성

```
class MyDBHelper : SQLiteOpenHelper(...) {
    MyDBHelper() // 생성자
    onCreate() // 테이블 생성
    onUPgrade() // DB 수정이 필요한 경우
}
```

2) helper 객체 생성

```
val helper = MYDBHelper(...)
```

3) helper를 사용하여 SQLiteDatabase 객체 획득

```
val writableDB = helper.writableDatabase // 쓰기 전용(insert, update, delete)
val readableDB = helper.readableDatabase // 일기 전용(select)
```

4) SQLiteDatabase 객체를 사용하여 Query 수행

```
- 메소드 사용 방식
writableDB.insert() // 자료 저장 시
writableDB.update() // 자료 수정 시
writableDB.delete() // 자료 삭제 시
writableDB.query() // 자료 검색 시

- SQL문 직접 작성 방식
writableDB.execSQL() // insert, update, delete 사용 시
readableDB.rawQuery() // select 사용 시
```

5) helper 객체(및 관련 객체) Close 수행

```
helper.close() // Cursor를 사용하였을 경우 Cursor.close() ← select 사용 시
```

2) ~ 5)은 DB사용이 필요한 곳마다 해당 코드 작성

II. DB 준비-SQLiteOpenHelper 클래스

1. SQLiteOpenHelper

1) 개념

A. 데이터베이스를 편리하게 사용할 수 있도록 도와주는 클래스 → 상속하여 사용

- 데이터베이스 저장 파일 생성
- 테이블 생성
- 테이블 업그레이드, 기본 샘플 데이터 추가 등

- SQLiteDatabase 객체 제공

B. 필수 재정의 메소드

- **생성자** : 사용할 DB 파일 명 및 DB 버전을 지정
- **onCreate()**
  - 사용할 테이블을 SQL을 사용하여 생성
  - 샘플이 필요할 경우 테이블 생성 후 샘플 추가 문장 작성
- **onUpgrade()**
  - 테이블 구조를 변경해야 할 필요가 있을 때 사용. 특별하게 재정의 하지 않아도 무방

    원래 테이블을 지우고 → onCreate() 호출

2) SQLiteOpenHelper 상속 클래스 작성

A. FoodDBHelper

```
class FoodDBHelper(context: Context?) : SQLiteOpenHelper(context, DB_NAME, null, 1) {
    val TAG ="FoodDBHelper
```

- 최초로 readableDatabase 또는 writableDatabase 사용시 호출
- DB의 테이블 생성 및 필요시 샘플 데이터 추가

```
    companion object {
        const val DB_NAME ="food_db"
        const val TABLE_NAME ="food_table"
        const val COL_FOOD ="food"
        const val COL_COUNTRY ="country"
    }
```

- DB 및 테이블, 테이블 컬럼명 등을 companion object로 생성하여 보관 → 해당 항목명의 사용 일관성을 위해
- 사용 예: FoodDBHelper.TABLE_NAME

```
    override fun onCreate(db: SQLiteDatabase?) { // 테이블 만듦
        val CREATE_TABLE =
            "CREATE TABLE ${TABLE_NAME} (${BaseColumns._ID} INTEGER PRIMARY KEY AUTOINCREMENT,"+
            "${COL_FOOD} TEXT, ${COL_COUNTRY} TEXT)"
        Log.d(TAG, CREATE_TABLE) // 확인
        db?.execSQL(CREATE_TABLE) // 실행
    }
```

- 최초로 readableDatabase 또는 writableDatabase 사용시 호출
- DB의 테이블 생성 및 필요시 샘플 데이터 추가
- BaseColumns 인터페이스의 _ID 속성(_id) 지정 → 다른 안드로이드 요소에서 필요한 경우가 있음
- AUTOINCREMENT : 자동 증가하면서 id 부여

```
    override fun onUpgrade(db: SQLiteDatabase?, oldVer: Int, newVer: Int) {
        val DROP_TABLE ="DROP TABLE IF EXISTS ${TABLE_NAME} TEXT"
        db?.execSQL(DROP_TABLE)
        onCreate(db)
    }
}
```

- 새로운 버전의 DB를 사용할 필요가 있을 때 사용

2. Helper 객체 생성

A. DB사용 부분에서 객체 생성

B. Helper 객체 생성 시의 동작

- 생성자에 의해 DB파일이 안드로이드 내부저장소에 생성
- 실기기는 보안 문제로 폴더 외부 접근이 안되므로 에뮬레이터 상에서만 확인 가능
- [Device File Explorer] 사용
  - 에뮬레이터 선택 후 [File Explorer] 선택

```
data/data/PACKAGE_NAME/databases/DATABASE_NAME
예) data/data/mobile.example.dbtest/databases/food_db
```

- DB파일은 컴퓨터에서 다운받을 수 있으며, 다운받은 파일은 SQLiteBrowser 프로그램을 사용하여 내용 확인 가능

3. SQLiteDatabase

A. Helper 클래스에 의해 관리되는 데이터베이스 클래스

- DB 작업 Query를 수행

B. Helper 클래스를 사용하여 획득

- 읽기 전용(select)

```
val myDB : SQLiteDatabase = myDBHelper.readableDatabase
```

- 읽기/쓰기 겸용(insert/update/delete)

```
val myDB : SQLiteDatabase = myDBHelper.writableDatabase
```

동시접근시(데이터 바꿀 때)를 대비해 나누어져 있음

C. SQL을 직접 사용하거나 관련 메소드를 사용하여 Query 수행

D. 모든 작업 수행 후 Helper 객체를 통해 반드시 ☆close() 하여 종료

Helper.close()

III. SQLiteDatabase Query

1. 데이터 삽입

1) Query 전용 함수 사용 or SQL 사용

- 전용함수 사용은 Content Provider 사용과 동일

2) insert() or execSQL()

- ContentValues 객체 생성 후 입력할 데이터 값 설정 후 insert() 사용 → 반환값이 있으므로 삽입 결과 확인 가능
- execSQL()를 사용하여 SQL 직접 작성 후 수행

3) 코드

A. 전용 함수 사용 방식 <얘로 많이 사용>

```
val db = helper.writableDatabase
val newRow = ContentValues()
newRow.put("food", food)
newRow.put("country", country)
db.insert("food_table", null, newRow)
helper.close()
```

- id는 auto로 부여되기 때문에 안써줘도 됨
- 수행 완료 후 helper.close() 호출

B. SQL 직접 사용방식 <눈에 잘 안들어옴>

```
db.execSQL("INSERT INTO food_table"+"VALUES (null, '된장찌개', '한국')")
```

- execSQL() 사용
- SQL 명령어를 직접 작성
- **☆ 문자열 표시 '' 사용 주의!!!! ★**

```
db.execSQL("insert into ${FoodDBHelper.TABLE_NAME}"+"values (null, ?, ?)",arrayof("된장찌개", "한국"))
```

- 매개변수 결합 방식
- execSQL(sql, 매개변수 배열)

2. 데이터 수정

1) update() or execSQL()

- ContentValues 객체에 변경할 값 및 조건 설정 후 update() 사용
- execSQL()를 사용하여 SQL 직접 작성 후 수행

2) 코드

A. 전용 함수 사용 방식 <얘로 많이 사용>

```
val db = helper.writableDatabase
val updateRow = ContentValues()
updateRow.put("country", "한국")
val whereClause="food=?"// 수정할 row 검색 조건-여러개일 경우 A=? and B=? ...
val whereArgs= arrayOf("된장찌개") // 검색 조건의 ?와 결합할
db.update("food_table" ,updateRow ,whereClause ,whereArgs )
helper.close()
```

- 배열인 이유 → food=? and country=? 이렇게 될 수 있어서

B. SQL 직접 사용방식 <눈에 잘 안들어옴>

```
db.execSQL("UPDATE food_table"+"SET country='한국' WHERE food='된장찌개' ")
```

- 매개변수 결합방식 사용 가능

3. 데이터 삭제

1) delete() or execSQL()

- 삭제대상 검색 조건 지정 후 delete() 수행
- execSQL 메소드를 사용하여 SQL 직접 작성 후 수행

2) 코드

A. 전용 함수 사용 방식 <얘로 많이 사용>

```
val db=helper.writableDatabas
val whereClause="food=?"
val whereArgs= arrayOf("된장찌개")
db.delete("food_table" ,whereClause ,whereArgs )
helper.close()
```

- 두 개 다 NULL 값 삽입시 테이블 데이터가 전부 지워짐 주의

B. SQL 직접 사용방식 <눈에 잘 안들어옴>

```
db.execSQL("DELETE FROM food_table WHERE food='된장찌개' ")
```

- 매개변수 결합방식 사용 가능

4. 데이터 검색

1) query() or rawQuery()

- 조건문 지정 후 query() 수행
- rawQuery 메소드를 사용하여 SQL 직접 수행
- Cursor(데이터 반환 집합에 대한 레퍼런스) 값을 반환
- 사용 후 Helper 및 Cursor는 반드시 close() 적용

2) 코드

A. 전용 함수 사용 방식 <얘로 많이 사용>

```
val db = helper.readableDatabase // 읽기전용 사용 가능
val columns = arrayof("_id","food","country") // 검색할 컬럼명, null일 경우 모든 컬럼
val selection ="food=?"// 검색조건(읽어올때 조건)
val selectArgs = arrayof("된장찌개") // 검색조건 ?에 결합할 값
val cursor = db.query("food_table", columns, selection, selectArgs, null,null,null,null)
                                                                    // groupby, having, orderby, limit
```

- columns, selection, selectArgs이 전부 null일 경우 = 모두 다 읽어 오고 싶다.

B. SQL 직접 사용방식 <눈에 잘 안들어옴>

```
val cursor = db.rawQuery("SELECT * FROM food_table"+"WHERE food='된장찌개'", null)
```

- 매개변수 결합방식 사용 가능

5. Cursor 〈dto〉

- select문에 의해 반환한 레코드의 집합을 가르킴
- Cusor.moveToNext() : 다음 레코드가 있으면 true, 없으면 false
- Cursor.getType(column_index)를 사용하여 값을 읽어옴

A. 코드

```
//val foodlist = ArrayList<FoodDto>() // DB 검색 결과를 DTO에 저장하여 List에 보관하고자 할 경우

with(cursor) { // with(T){ }: T 객체가 this 역할
      while (moveToNext()) {
      val id = getInt(getColumnIndex("_id"))
      val food = getString(getColumnIndex("food")) //
      val country = getString(getColumnIndex("country"))
            Log.d(TAG,"$id - $food ( $country )")
      //list.add(FoodDto(id, food, country)) // 결과를 DTO 객체에 저장후 list에 보관
      }
}
cursor.close()
helper.close() // 사용 완료 후 cursor, helper순으로 close() 호출
```

- **Cursor.getColumnIndex()** : 컬럼명으로 컬럼의 순서(index)를 알아냄
  Cursor.getColumnIndex(“food“) == 1

Room 활용

I. Room 설명

1. Room 개요

1) ROOM

- SQLite의 직접 사용이 아닌 추상화 계층을 적용하여 데이터베이스 활용을 지원하는 지속성 라이브러리
  - SQL 쿼리의 컴파일 시간 확인
  - Annotation을 통한 상용구(Boilerplate) 코드 최소화
  - 데이터베이스 이전 간소화

2. 주요 구성요소

1) Room Database (↳ SQLite open helper)

- 실제 데이터베이스를 관리
- 데이터베이스 접근점 제공
  → DB 및 DAO 생성

2) Room Entity (↳ Table & DTO)

- 데이터베이스 상의 테이블 표현
- 결과 레코드를 저장하는 용도로 사용
  → DTO 역할로도 사용

3) Room DAO → 쿼리 던지고 실행

- Database Access Object
- 데이터베이스 테이블에 select, insert, update, 그리고 delete를 수행하기 위한 메소드 정의

+ 인터페이스 만드로 Room에다 요청

II. 필요한 코드

1. 사전 준비

1) build.gradle 수정

- Room 관련 Dependency등 추가
- 필요에 따라 비동기 처리 라이브러리 등 Dependency 추가

A. build.gradle(Project:...)

```
plugins {
    ...
    id 'com.google.devtools.ksp' version '1.8.10-1.0.9' apply false
}
```

B. build.gradle(Module:...)

- Groovy 기준

```
dependencies {
    ...
    val room_version ="2.5.0"
    implementation "androidx.room:room-runtime:$room_version"
    annotationProcessor "androidx.room:room-compiler:$room_version"
    // optional - Kotlin Extensions and Coroutines support for Room
    implementation "androidx.room:room-ktx:$room_version" → 비동기 처리(Room과 Coroutine) 사용 시 필요
    // To use Kotlin annotation processing tool (kapt)
    // kapt "androidx.room:room-compiler:$room_version"
    // To use Kotlin Symbol Processing (KSP)
    ksp "androidx.room:room-compiler:$room_version"
}
```

2. Room Entity

1) 개요

- Room 지속성 라이브러리 사용 시 데이터를 표현(저장)하는 객체
- 클래스 선언과 동일한 테이블이 생성
- 인스턴스는 생성한 테이블의 데이터 행 하나를 표현
- 필요에 따라 생성자, ~~getter/setter~~, toString() 추가

2) Food 테이블 생성

- Annotation 사용

A. 기본

```
@Entity
data class Food ( // Entity 지정 , 테이블 명: Food
      @PrimaryKey
      val _id : Int, // 기본 키 지정

      val food : String,
      val country : String // 변수명으로 컬럼 생성
)
```

B. 상세

```
@Entity (tableName ="food_table") // 테이블 명 지정
data class Food (
      @PrimaryKey (autoGenerate =true) // 기본 키 값 자동 생성
      val _id : Int,

      @ColumnInfo(name ="food") // 컬럼 명 지정
      val food : String?,

      @ColumnInfo(name ="country")
      val country : String?,

      @Ignore
      var dummy : String? // 테이블 컬럼에서 제외
)
```

3. Room DAO

1) 개요

- ROOM 사용 시 데이터 접근 메소드를 정의하는 인터페이스
- Annotation에 따라 구현 클래스를 자동 생성
  - 편의 메소드: Query 작성 없이 삽입, 업데이트, 삭제 수행
  - 쿼리 메소드: 직접 Query를 작서앟여 테이블 접근

A. 예

```
@Dao
interface FoodDao { // DAO 지정
      @Insert // 편의 메소드
      fun insertFood(vararg food: Food)

      @Update // 편의 메소드
      fun updateFood(food: Food)

      @Delete // 편의 메소드
      fun deleteFood(food: Food)

      @Query("SELECT * FROM food_table") // 쿼리 메소드
      fun getAllFoods(): List<Food>
}
```

4. 편의 메소드

1) @Insert

- **메소드에 정의한 매개변수를 테이블에 삽입**
  - 매개 변수는 Entity 또는 Entity의 컬렉션(예: List<Food>)
  - 단일 매개변수일 경우 long 반환 (rowId)
  - 컬렉션일 경우 Long[] 또는 List<Long> 반환

A. 예

```
@Insert(onConflict = onConflictStrategy.REPLACE)
fun insertFoods(vararg foods : Food) : List<Long>

@Insert
fun insertTwoFoods(food1: Food, food2: Food)

@Insert
fun insertFoodAndFoods(food: Food, foods: List<Food>)
```

- REPLACE : 테이블에 동일한 ID의 행이 있을 경우 대체
- vararg : 가변인자(인자 개수 유동적 지정) = 여러개 넣기 가능, 다양한 형태로
- Long : 추가한 행의 rowId 리스트 반환

2) @Update

- **메소드에 정의한 매개변수 Entity의 동일 ID행을 찾아 수정 수행**

A. 예시

```
@Update
fun updateFood(food: Food) : Int

@Update
fun updateFoods(vararg foods: Food)
```

- Int : 수정한 행 개수 반환 가능

3) @Delete

- **메소드에 정의한 매개변수 Entity의 동일 ID행을 찾아 삭제 수행**

A. 예시

```
@Delete
fun deleteFood(food: Food) : Int

@Delete
fun deleteFoods(vararg foods: Food)
```

- Int : 삭제한 행 개수 반환 가능

5. 쿼리(Query) 메소드

A. 테이블의 데이터를 접근하거나 복잡한 삽입, 수정, 그리고 삭제 등을 사용할 때 사용

B. 예시

```
@Query("SELECT * FROM food_table") // Query 직접 작성
fun getAllFoods() : List<Food>

@Query("SELECT * FROM food_table WHERE _id = :id") // 메소드의 매개변수 사용, Query 내부에 작성 시 :매개변수 형식 사용
fun getAllFood(id: Int) : Food

@Query("SELECT * FROM food_table WHERE country IN (:country)")
fun getFoodsByCountry(country: List<String) : List<Food>// 컬렉션 사용 시

@Query("DELETE FROM food_table WHERE _id = :id") // Query 직접 작성 시 사용
fun deleteFoodById(id: Int)
```

6. ROOM Database

1) 개요

- 데이터베이스 설정 및 데이터 접근 지점을 제공하는 추상 클래스(RoomDatabase 상속)
  - Helper 역할 수행
  - DAO 생성

A. 메타 정보

```
@Database (entities = [Food::class], version =1)
abstract class FoodDatabase : RoomDatabase() {
      abstract fun foodDao() : FoodDao
}
```

① 사용할 엔터티들 전달
② 버전 → helper랑 비슷
③ 추상 클래스 : Room DB 상속
④ DAO 정의 + ⑤

B. 사용

```
val db : FoodDatabase = Room.databaseBuilder(
      applicationContext, FoodDatabase::class.java, "food_db"
).build() // DB생성
```

```
val foodDao : FoodDao = db.foodDao() // DB객체에서 foodDao 생성(자동으로 채워져서 나타남)
```

```
val foods = foodDao.getAllFoods() // DAO에서 정의한 메소드 사용
// insertFood(), updateFood(), deleteFood() 등
```

7. 고려 사항

1) ☆ Thread의 사용 ★

- 메인(UI) 스레드에서 ROOM DAO 메소드 사용을 제한
  → DB 접근 등 시간이 많이 걸리는 작업을 분리하도록 강제

A. 잘못된 코드

```
fun showAllFoods() {
      val foods: List<Food>= foodDao.getAllFoods() // 실행할 DAO 메소드

      for (food in foods) {
            Log.d(TAG, food.toString())
      }
}
```

```
onMainThreadExcepti
```

B. 별도의 스레드 생성 또는 비동기 쿼리 방식 사용

```
binding.btnSelect.setOnClickListener{
      Thread {
            showAllFoods()
      }.start() // Main Thread가 아닌 새로운 Thread를 생성하여 해당 Thread에서 실행
}
```

2) RoomDatabase에 Singleton 패턴 적용

ㄴ 객체를 하나만 유지하고자 할 때 적용하는 Design Pattern

- 데이터베이스 인트턴스는 많은 자원을 소비 → 객체를 하나만 생성하여 사용할 수 있도록 구현 개선 필요

A. 개선된 RoomDatabase

```
@Database (entities = [Food::class], version =1)
abstract class FoodDatabase : RoomDatabase() {
      abstract fun foodDao() : FoodDao

   companion object {
      @Volatile // MainMemory에 저장한 값 사용
      private var INSTANCE: FoodDatabase? =null

      // INSTANCE가 null이 아니면 반환 null이면 생성하여 반환(처음실행할때만 만듦)
      fun getDatabase(context: Context) : FoodDatabase {
         return INSTANCE ?: synchronized(this) { // 단일 스레드만 접근
            val instance = Room.databaseBuilder(
            context.applicationContext, FoodDatabase::class.java, "food_db"
            ).build()
            INSTANCE = instance
            instance
         }
      }
   }
}
```

B. 사용

```
db = FoodDatabase.getDatabase(this)
```

III. Coroutine 함수

1. 비동기(Asynchronous) 방식의 사용

A. DAO의 쿼리를 비동기 방식으로 구현(고려사항 1 Thread 대체)
→ 메소드이 반환 타입을 비동기 처리 클래스 사용으로 변경

B. 쿼리의 종류

- One-shot Write Query: insert, update, delete
- One-shot Read Query: select (한 번 실행)
- Observable Read Query: select (DB의 변경이 있을 때마다 실행)

C. 적용 가능 언어 및 프레임워크 옵션

쿼리 유형	Kotlin 언어 기능	RxJava	Guava	Jetpack 수명 주기
원샷 쓰기	**코루틴(suspend)**	Single<T>, Maybe<T> , Completable	ListenableFuture<T>	해당 사항 없음
원샷 읽기	**코루틴(suspend)**	Single<T>, Maybe<T>	ListenableFuture<T>	해당 사항 없음
관찰 가능한 읽기	**Flow<T>**	Flowable<T>, Publisher <T>, Observable<T>	해당 사항 없음	LiveData<T>

D. 코루틴(Coroutine) 사용 시: 비동기 처리 dependencies 추가

```
dependencies {
      …
      val room_version ="2.5.0"
      implementation "androidx.room:room-runtime:$room_version"
      annotationProcessor "androidx.room:room-compiler:$room_version"
      // optional - Kotlin Extensions and Coroutines support for Room
      implementation "androidx.room:room-ktx:$room_version" → 비동기 처리(Room과 Coroutine) 사용 시 필요
      // To use Kotlin annotation processing tool (kapt)
      // kapt "androidx.room:room-compiler:$room_version"
      // To use Kotlin Symbol Processing (KSP)
      ksp "androidx.room:room-compiler:$room_version"
}
```

2. 비동기 원샷 쿼리의 작성

A. ROOM은 Kotlin Coroutine을 지원

- build.gradle(Module:…) dependencies 추가

```
implementation "androidx.room:room-ktx:2.5.0"
```

B. 한 번만 실행하는 쿼리를 통해 데이터 접근

```
@Dao
interface FoodDao {
      @Query("SELECT * FROM food_table")
      fun getAllFoods() : List<Food>

      @Query("SELECT * FROM food_table WHERE country = :country")
      suspend fun getFoodsByCountry(country: List<String) : List<Food>

      @Insert
      suspend fun insertFood(vararg food: Food)

      @Update
      suspend fun updateFood(food: Food)

      @Delete
      suspend fun deleteFood(food: Food)
}
```

- suspend : Coroutine 사용 함수로 지정

3. 관찰 가능한 쿼리 작성 - Flow〈T〉

A. 쿼리에서 참조하는 테이블이 변경될 때마다 새로운 값을 표시 → 최신 정보 유지

```
import kotlin.coroutines.flow.Flow

@Dao
interface FoodDao {
      @Query("SELECT * FROM food_table")
      fun getAllFoods() : Flow<List<Food>>

      @Query("SELECT * FROM food_table WHERE country = :country")
      suspend fun getFoodsByCountry(country: List<String) : List<Food>// 일회성 쿼리 및 관찰 수행
```

- Flow<List<Food>>
  - 반환 타입을 Flow<T> 지정
  - 지속적인 관찰 수행

B. Flow<T>의 결과 확인

```
val foodFlow: Flow<List<Food>>= foodDao.getAllFoods()
foodFlow.collect { foods ->
      for (food in foods) {
           Log.d(TAG,food.toString())
      }
}
```

- collect()를 사용하여 수집한 결과 확인

4. Coroutaine 함수의 호출

A. Scope 지정

- Coroutine 함수(suspend함수, Flow 사용 등)들은 코루틴 스코프 내에서 실행
- CoroutineScope(Dispatchers.IO).launch {...} 사용

B. 예

```
fun addFood() {
      val food = Food(0, "된장찌개", "한국")
    CoroutineScope(Dispatchers.IO).launch {
        foodDao.insertFood(food)
    }
}
```

- suspend 함수 호출
- Flow<T> 반환 함수 호출 등에도 적용

5. 고려사항

1) Flow<T> 사용 시의 고려사항

- Flow<T>는 지속적으로 대상을 관찰하므로 변경이 있을 때마다 쿼리가 다시 실행됨
- 관심 결과와 관계없이 테이블의 변경이 있을 때마다 실행 되므로 불필요한 실행 발생
  → Flow<T>.distinctUntilChanged() 사용 : 변경이 있을 때만 쿼리를 실행하도록 변경한 Flow<T> 반환

2) 기존 Flow<T> 사용의 개선

```
val foodFlow: Flow<List<Food>>= foodDao.getAllFoods()
foodFlow.distinctUntilChanged().collect { foods ->
      for (food in foods) {
        Log.d(TAG, food.toString())
    }
}
```

- 변경이 있을 때만 쿼리를 수행하는 Flow로 변경

Android Application Architecture

I. Android App. Architecture

1. 개요

1) 아키텍처 구성의 원칙

A. 관심사의 분리(Separation of Concerns)

클래스는 정해진 역할만!!! eg. 액티비티는 화면 구성만 담당한다.

B. 데이터 모델과 UI 의 분리

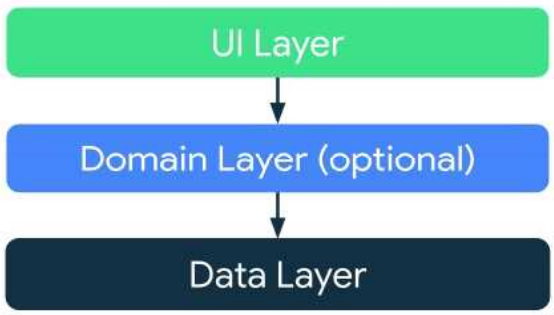
eg. 엑셀 데이터/ 그래프 → 표(데이터)를 고치면 그래프가 바뀜 ⇒ 데이터와 화면은 분리

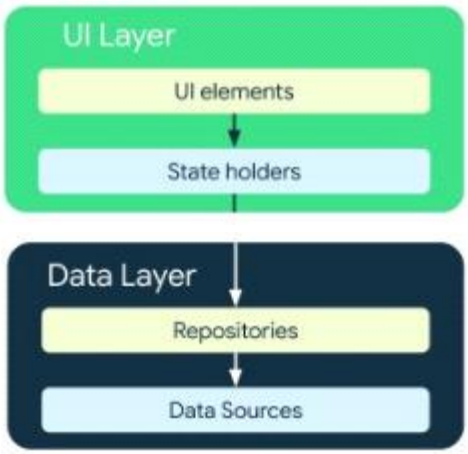
C. Data의 단일 소스 저장소(SSOT: Single Source of Truth) 사용

① DB는 DB대로 만들기 → ② 네트워크는 네트워크 대로 만들기 → ③ 각각한 이 두 개를 처리할 수 있는 놈 만들기

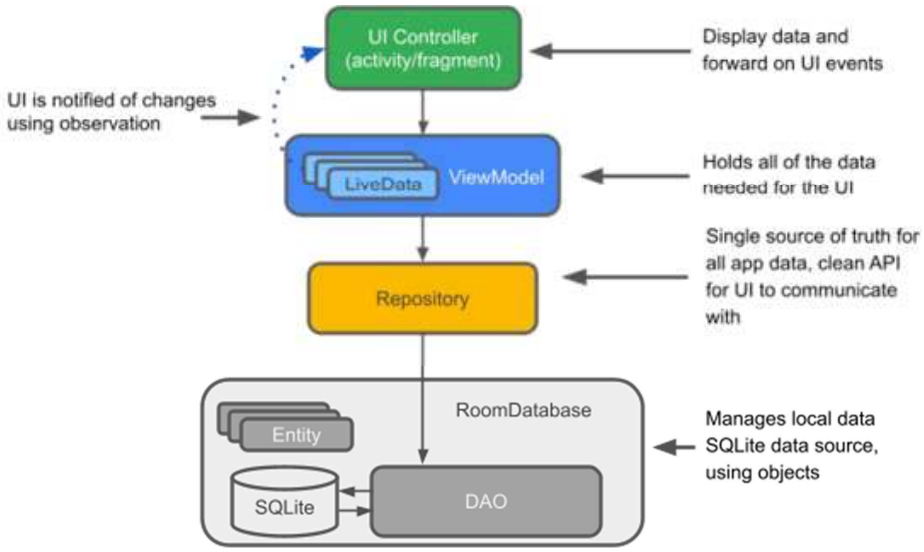
D. 단방향 데이터 흐름(UDF: Unidirectional Data Flow)

2) 권장 Architecture





2. 주요 구성요소



A. LiveData : 데이터의 변경을 관찰할 수 있는 데이터 보관 클래스

B. ViewModel : Data 와 UI 사이의 통신을 담당 및 데이터 지속 보유

C. Repository: 데이터 소스를 관리하는 개발자 구현 클래스

D. Entity : ROOM 사용 시 DB 테이블 표현 및 데이터 보관

E. RoomDatabase : 데이터베이스 관린, DAO 생성

F. DAO : 데이터 접근 객체

G. SQLite DB : 데이터 보관

II. 구현

1. 구현 준비

1) build.gradle(Project:...)

```
plugins {
    ...
    id 'com.google.devtools.ksp' version '1.8.10-1.0.9' apply false
}
```

2) build.gradle(Module:...)

```
plugins {
    ...
    id 'com.google.devtools.ksp'
}

dependencies {
    ...
    //    ROOM
    def room_version ="2.5.0"
    implementation "androidx.room:room-runtime:$room_version"
    annotationProcessor "androidx.room:room-compiler:$room_version"
    implementation "androidx.room:room-ktx:$room_version"
    ksp "androidx.room:room-compiler:$room_version“

    // viewModels 사용
    implementation "androidx.activity:activity-ktx:1.7.2"
    // implementation "androidx.fragment:fragment-ktx:1.6.1“

    // Lifecycle components
    def lifecycle_version ="2.6.2"
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version"
    implementation "androidx.lifecycle:lifecycle-livedata-ktx:$lifecycle_version"
    implementation "androidx.lifecycle:lifecycle-common-java8:$lifecycle_version"
}
```

2. Data Layer 구성

1) 데이터 소스를 포함하는 저장소로 구성

- 앱의 데이터를 한 곳에 모아 관리

2) Repository

- 데이터 소스르를 **관리**하는 클래스
- 예: FoodDao를 관리하는 FoodRepository
- DAO를 한 곳에 모아 관리함

A. 코드 (FoodRepository.kt)

```
class FoodRepository(private val foodDao: FoodDao) {
    val allFoods : Flow<List<Food>>= foodDao.getAllFoods()

    suspend fun addFood(food: Food) { // 원샷 또는 관찰가능한 쿼리 관련 멤버 선언
        foodDao.insertFood(food)
    }
}
```

- private val foodDao: FoodDao : FoodDao를 private 멤버로 선언
- Flow : import 시 Coroutine의 flow를 추가 해야 함
- suspend : 코루틴 적용을 위해 suspend 적용

3. Repository 객체 생성 ???????????

1) Repository 의 생성

- 앱이 실행되는 동안 계속 사용하여야 할 경우 Application 클래스로 지정
- 앱 실행 중 계속 사용 가능한 FoodRepository

A. FoodApplication.kt (앱 실행 중 계속 사용 가능한 FoodRepository)

```
class FoodApplication: Application() {
    val database by lazy {
        FoodDatabase.getDatabase(this)
    }

    val repository by lazy {
        FoodRepository(database.foodDao())
    }
}
```

- Application 상속 : 앱 끝날때까지 실행 될 수 있도록

B. FoodApplication 등록

AndroidManifest 에 기록

```
<application
      android:name=".FoodApplication"
      android:allowBackup="true"
```

- android:name=".FoodApplication" : 기본 Application에서 구현한 FoodApplication으로 대체

C. application 이라는 객체명으로 앱 전체에서 접근 가능 → 사용 시 형변환 필요

4. UI Layer 구성 (State holders)

1) ViewModel의 사용

- 비즈니스 로직 또는 화면 수준 상태 홀더
- UI의 상태 유지
- 비즈니스 로직에 대한 접근 권한 제공
- ViewModel 상속

A. FoodViewModel.kt

```
class FoodViewModel (var repository : FoodRepository) : ViewModel() {
    var allFoods: LiveData<List<Food>>= repository.allFoods.asLiveData()

    fun addFood(food: Food) = viewModelScope.launch {
        repository.addFood(food)
    } // 코루틴 범위를 viewModelScope 범위로 지정
}
```

- Flow<List<Food>> → LiveData<List<Food>> 로 변환
- LiveData<화면에 보여지는 데이터만 보관>
  - 데이터를 관찰가능한 데이터보관 클래스
  - 데이터변경이 발생하면 자동 반영

5. ViewModel 객체 생성

1)☆ FoodViewModel 생성 방법 ★

- Factory 사용 → ViewModelProvider.Factory 상속 및 create(Class<T>) : T 재정의

A. FoodViewModel.kt

```
class FoodViewModelFactory(private val repository: FoodRepository) : ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(FoodViewModel::class.java)) {
            @Suppress("UNCHECKED_CAST")
            return FoodViewModel(repository) as T // FoodViewModel 객체 반환
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
}
```

- private val repository: FoodRepository : repository 전달

2)객체 생성 : Activity의 변경 등에 상관 없이 유지

A. MainActivity.kt

```
class MainActivity : AppCompatActivity() {
    val viewModel : FoodViewModel by viewModels {
        FoodViewModelFactory((application as FoodApplication).repository)
    }
}
```

- FoodViewModel by viewModels : ViewModel delegate
- FoodApplication : application 멤버로 선언한 repository 사용

6. ViewModel을 사용한 데이터 접근

1)데이터 관찰 및 데이터 접근 수행

A. MainActivity.kt

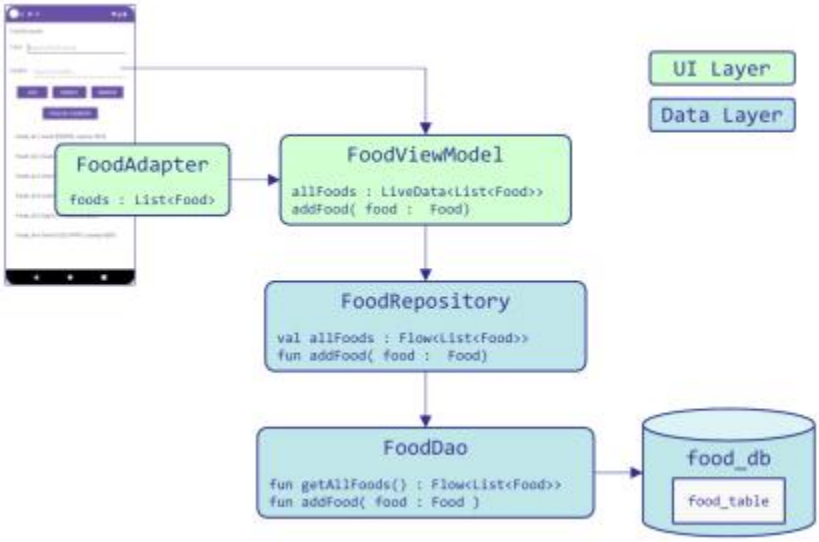
```
val viewModel : FoodViewModel by viewModels {
     FoodViewModelFactory( (application as FoodApplication).repository )
}

override fun onCreate(savedInstanceState: Bundle?) {
    ...
      viewModel.allFoods.observe(this, Observer {foods ->// LiveData.observe(context, Observer) 호출
        adapter.foods = foods adapter.notifyDataSetChanged()
        Log.d(TAG, "Observing!!!")
    })

      mainBinding.btnAdd.setOnClickListener {
        viewModel.addFood(
            Food(0, mainBinding.etFood.text.toString(),
            mainBinding.etCountry.text.toString()) // ViewModel 에 정의한 함수를 호출하여 repository 접근
        )
}
```

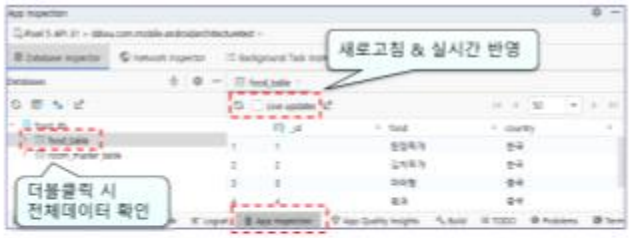
- observe : LiveData를 보고 잡다.....

7. ☆ FoodDBExam 전체 Architecture ★



III. 참고 사항

1. Database Inspector



HTTP Communication

I. Android Network 개요

1. Connectivity Manager

1) 네트워크 사용 시 고려사항

- 네트워크 환경 조사 (eg. wifi 켜기)
- 사용 가능한 네트워크 환경 선택
- 네트워크 환경 변화 처리
- 네트워크 사용
- 네트워크 환경 조사
- ConnectivityManager 사용 → NetworkInfo 확인
- 필요 퍼미션 : <매니페스트>
  android.permission.ACCESS_NETWORK_STATE
- 주요 메소드(getter 사용값)
  - getAllNetworkInfo() : 기기가 제공하는 모든 Network 타입 조사
  - getActiveNetworkInfo() : 활성화 상태의 Network 타입 조사
  - getNetworkInfo(type: Int) : 특정 Network 타입 조사

2) 네트워크 환경 조사의 예

```
private fun getNetworkInfo() {
      val connMgr = getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
      var isWifiConn: Boolean =false
      var isMobileConn: Boolean =false
      connMgr.allNetworks.forEach { network ->// allNetworks = Network 배열 반환
            connMgr.getNetworkInfo(network)?.apply {
                  if (type == ConnectivityManager.TYPE_WIFI) {
                        isWifiConn = isWifiConn or isConnected
                  }
                  if (type == ConnectivityManager.TYPE_MOBILE) {
                        isMobileConn = isMobileConn or isConnected
                  }
            }
      }
      Log.d(TAG, "Wifi connected: $isWifiConn")
      Log.d(TAG, "Mobile connected: $isMobileConn")
}
```

- NetworkInfo 클래스
  - 네트워크 타입 정보를 표현하는 클래스
  - 현재 사용가능한 네트워크 확인

3) 네트워크 사용 가능 확인 예

```
private fun isOnline(): Boolean {  // 필수
      val connMgr = getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
      val networkInfo: NetworkInfo? = connMgr.activeNetworkInfo
      return networkInfo?.isConnected ==true
}
```

- ConnectivityManager.getActiveNetworkInfo()
  - 현재 활성화 상태의 네트워크 정보를 NetworkInfo 객체로 반환 <연결 상태>

4) 사용 Permission (AndroidManifest.xml 에 추가)

android.permission.ACCESS_NETWORK_STATEA

II. HTTP 통신

1. HTTP 통신

HTTP는 stateless로 상태를 유지 하지 않는다.

- Get (Read), Post(Create), Put(Update), Delete(Delete) ⇒ URL

1) HTTP 기반의 네트워크 활용

- JAVA 기반의 네트워크 관련 API 이용
- 필요 permission (AndroidManifest.xml)
  - android.permission.INTERNET → 기본
  - android.permission.ACCESS_NETWORK_STATE

2) HTTP Connection 설정

- URL 클래스를 사용하여 접속할 서버 url **설정**
- URLConnection (추상 클래스)를 사용하여 서버와의 **연결** 확보
  - HttpURLConnection 또는 HttpsURLConnection (구현 클래스) → 만든다

http일 때 https일 때

```
val strUrl = "https://cs.dongduk.ac.kr" // https 일 경우 HttpsURLConnection 사용
val url : URL = URL(strUrl)
val conn : HttpsURLConnection = url.openConnection() as HttpsURLConnection
                                ↳ 연결하라
```

conn : 정보 설정 저장

3) 네트워크 관련 추가 사항

A. 안드로이드 9.0이상부터 https만을 적용(권장)

접속 주소에 http를 사용할 경우 Exception 발생

B. 처리 방법 1 : 서버가 https 지원시 https://로 변경

C. 처리 방법 2 : 보안관련 설정 추가

- 방법1: res/xml/network....xml 파일 추가 후 AndroidManifest 에 지정
- **방법2: AndroidManifest 의 <application> 속성에 속성값 추가 → 얘 사용**
  - android:usesCleartextTraffic="true"

4) HTTP 요청 설정 → 연결전(통신 전)

HttpURLConnection 메소드	역할
setConnectTimeout (timeout : Int) //connectTimeout	1/1000 단위로 연결제한시간 설정, 0일 경우 무한 대기
setReadTimeout (timeout : Int) // readTimeout 1/1000	단위로 읽기 제한 시간 설정, 0일 경우 무한 대기
setUseCaches (newValue : Boolean) // useCashes	캐시 사용 여부를 지정한다.
setDoInput (newValue : Boolean) // doInput	입력을 받을 것인지를 지정한다.
setDoOutput (newValue : Boolean) // doOutput	출력을 할 것인지를 지정한다.
setRequestProperty (field : String, newValue : String)	요청 헤더에 값을 설정한다.
addRequestProperty (field : String, newValue : String)	요청 헤더에 값을 추가한다. 속성의 이름이 같더라도 덮어 쓰지는 않는다.

5) HTTP 요청 처리

HttpURLConnection or HttpsURLConnection	역할
setRequestMethod (method : String) // requestMethod	GET 또는 POST 설정, 디폴트는 GET
connect ()	통신 연결 (네트워크 트래픽 발생)
getResponseCode() : Int // responseCode	서버에 요청을 전달하고 응답 결과를 받음 • 정상적으로 요청 전달: HTTP_OK (200) • URL을 발견할 수 없음: HTTP_NOT_FOUND(404) • 인증 실패: HTTP_UNAUTHORIZED (401) • 서비스 사용 불가: HTTP_UNAVAILABLE (503)
getInputStream() // inputStream	서버에서 전달받은 데이터를 InputStream 타입으로 반환
disconnect()	서버와의 연결 종료

2. GET/POST

1) HTTP GET 요청

```
fun downloadText(url: String) : String? {
      var receivedContents : String? =null
      var iStream : InputStream? =null
      var conn : HttpsURLConnection? =null
      try {
            val url : URL = URL(url)
            conn = url.openConnection() as HttpsURLConnection  // 서버 연결 설정 - MalformedURLException
            conn.readTimeout =5000  // 읽기 타임아웃 지정 - SocketTimeoutException
            conn.connectTimeout =5000  // 연결 타임아웃 지정 - SocketTimeoutException
            conn.doInput =true  // 서버 응답 지정 - default
            conn.requestMethod ="GET"  // 연결 방식 지정 - or POST

            // conn.connect()  // 통신링크 열기 - 다른 메소드에서 내부 호출, 생략 가능
            val responseCode = conn.responseCode  // 서버 전송 및 응답 결과 수신

            if (responseCode != HttpsURLConnection.HTTP_OK) {
                  throw IOException("Http Error Code: $responseCode")
            }

            iStream = conn.inputStream  // 응답 결과 스트림 확인
            receivedContents = readStreamToString (iStream)  // stream 처리 함수를 구현한 후 사용
      }  catch (e: Exception) {  // MalformedURLException, IOExceptionl, SocketTimeoutException 등 처리 필요
                  Log.d(TAG, e.message!!)
      }  finally {
                  if (iStream !=null) { try { iStream.close()} catch (e: IOException) { Log.d(TAG, e.message!!) } }
                  if (conn !=null) conn.disconnect()
      }
      return receivedContents
}
```

- readStreamToString() : 직접 구현 해야 됨
- iStream.close(), conn.disconnect() : Stream과 Connection을 종료시켜야 함
  연결 → 사용 → **종료!!!**

2) InputStream의 String 변환 함수 구현(바뀌는 것 없이 그대로 사용)

```
private fun readStreamToString(iStream : InputStream?) : String {
      val resultBuilder = StringBuilder()

      val inputStreamReader = InputStreamReader(iStream)
      val bufferedReader = BufferedReader(inputStreamReader)

      var readLine : String? = bufferedReader.readLine()
      while (readLine !=null) {
            resultBuilder.append(readLine + System.lineSeparator() )
            readLine = bufferedReader.readLine()
      }

      bufferedReader.close()
      return resultBuilder.toString()
}
```

- Stream → Stream Reader → Buffer Reader : 효율을 위해서 사용

3) InputStream의 이미지 변환 방법

```
private fun readStreamToString(iStream : InputStream?) : Bitmap {
    val bitmap = BitmapFactory.decodeStream(iStream)
    return bitmap
}
```

- decodeStream(iStream) : Image 파일이 클 경우 오류가 발생할 수 있으므로 표시할 이미지의 크기 조절이 필요할 수 있음

4) 권한 오류 처리

A. Main Thread에서의 네트워크 사용 금지

- 3.0 버전부터 사용자 응답성 문제로 인해 **Main Thread(UI Thread)에서 네트워크 사용 금지** → 응답에 시간이 걸릴 경우 ANR(Application Not Responding) 발생 (강제 종료)

B. 해결 방법(임시 및 테스트 용)

- onCreate()실행 시 아래 코드 실행

```
val pol = StrictMode.ThreadPolicy.Builder().permitNetwork().build()
StrictMode.setThreadPolicy(pol)
```

- 테스트나 임시로 네트워크를 확인해볼 때만 사용할 것
- Thread 또는 AsyncTask 등을 사용하여 별도의 스레드에서 네트워크 기능을 구현하는 방법으로 교체 필요

C. 비동기 처리 필수! → Thread or Coroutine

5) URL 구성 시 매개변수(쿼리)를 URL에 추가

프로토콜://호스트주소[:포트번호]/[경로]/[파일]?{쿼리}&{쿼리}...

6) 쿼리 구성

- ?param_name=param_value&...
- HashMap<String, String>으로 구성하기가 용이

7) REST API의 경우

http://cs.dongduk.ac.kr/bbs/board5<명사>

8) POST

A. URL.openConnection URL.openConnection()을 사용하여 HttpsURLConnection 객체 획득

B. conn.doOutput conn.doOutput = true 추가 지정

C. key=value 형태로 전송 값 구성 후 getOutputStream()을 사용하여 전송 값 준비

D. HttpURLConnection 의 getResponseCode()를 통해 전송 및 응답 확인

E. getInputStream()을 통해 서버의 응답 확인

F. HttpURLConnection을 disconnect()

9) POST 코드

```
conn = url.openConnection() as HttpsURLConnection // 서버 연결 설정 - MalformedURLException
conn.readTimeout =5000  // 읽기 타임아웃 지정 - SocketTimeoutException
conn.connectTimeout =5000 // 연결 타임아웃 지정 - SocketTimeoutException
conn.doInput =true  // 서버 응답 지정 - default
// conn.requestMethod = "GET“

conn.doOutput =true  // 서버 전송 지정
conn.requestMethod ="POST"  // 연결 방식 지정 - POST
conn.setRequestProperty("content-type", "application/x-www-form-urlencoded; charset=UTF-8")
val params1 ="user=somsom"  // 전송 데이터 설정 key1=value1&key2=value2…
val params2 ="&dept=computer"
val outStreamWriter : OutputStreamWriter = OutputStreamWriter(conn.outputStream, "UTF-8")
val writer : BufferedWriter = BufferedWriter(outStreamWriter)
writer.write(params1)  // 전송 데이터를 OutStream에 기록
writer.write(params2)
writer.flush()  // 전송 데이터 출력

val responseCode = conn.responseCode  // 서버 전송 및 응답 결과 수신

if (responseCode != HttpsURLConnection.HTTP_OK) {
      throw IOException("Http Error Code: $responseCode")
}

iStream = conn.inputStream  // 응답 결과 스트림 확인
receivedContents = readStreamToString (iStream)  // stream 처리 함수를 구현한 후 사용
```

네트워크 기능 구현(XmlPullParser)

I. OpenAPI

1. OpenAPI 사용

1) Open API ⇒ WEB

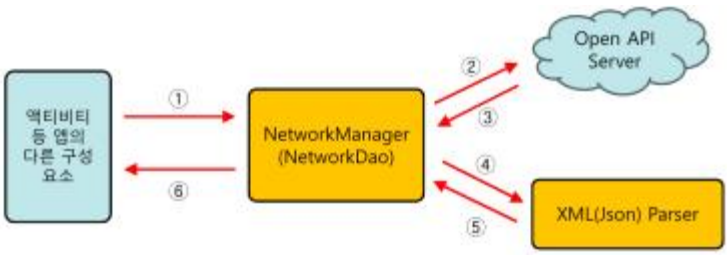
A. 개요

- Open Application Programming Interface
  - API: 플랫폼이나 웹서비스에서 갖고 있는 기능을 애플리케이션을 구현할 때 사용할 수 있도록 제공하는 인터페이스
- **일반적으로 REST 형식으로 제공**
  - GET 요청(URL)을 통해 XML 또는 Json 등의 형태로 결과를 받음
  - open API 접근 식별 유형
    - 고유의 key를 부여하여 URL 매개변수로 전송
    - Client ID/Client Secret을 사용하여 Header에 기록

B. Open API 서비스 예: 가입후 내키 발급받아 사용

- 영화진흥위원회 Open API: https://www.kobis.or.kr/kobisopenapi/homepg/main/main.do
- Naver Open API 사용: https://developers.naver.com/main/
- Google Open API 사용 : https://console.cloud.google.com/

2) Open API 처리 절차 및 구성 요소



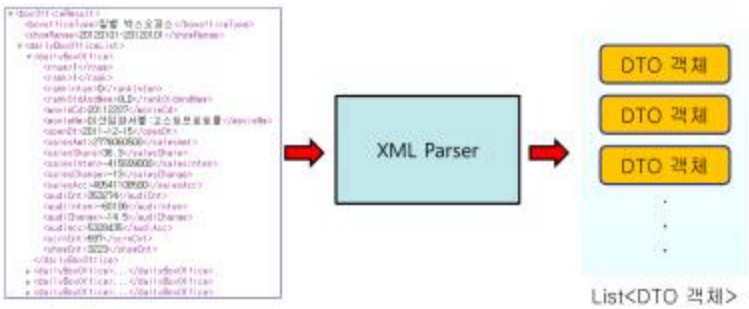
① OpenAPI 요청에 필요한 정보(검색어 등)를 전달하며 처리 요청
② OpenAPI 요청형식에 맞추어 정보 구성 후 서비스를 요청
③ 요청을 처리하여 응답을 전달(XML 또는 Json 응답)
④ 요청 결과 InputStream을 XML(Json) Parser에 전달
⑤ Parsing 수행 후 각 항목을 DTO로 변환하여 List<DTO> 형태로 변환
⑥ List<DTO> 결과를 요청한 구성요소에 반환

II. XML Parsing 구현

1. Parser의 구현 예 1

A. XML 문서를 읽어 하나의 요소를 하나의 DTO에 저장하는 작업 반복

B. 생성한 DTO는 List 등의 컬렉션에 저장



2. 아이템 저장을 위한 DTO 정의

A. 아이템 저장을 위한 DTO

```
data class Movie( // Data 클래스 생성
    var rank: Int?,
    var title: String?,
    var openDate: String?
)
```

- 필요한 경우 _id 등의 필드 추가
- DB와 같이 사용할 경우 Entity로 선언

- XML 문서의 한 항목 정보 → 하나의 DTO에 저장
- 한 항목이 포함한 태그(속성) → DTO 객체의 하나의 멤버 변수에 저장
- 하나의 XML 문서 내의 모든 항목들 → List 등 하나의 컬렉션에 저장

3. XMLPullParser 주요 요소(객체명이 parser일 경우)

A. parser.name

- getName() 의 반환 값, TAG의 이름 반환 (읽고 있던 놈에서)

B. parser.eventType

- getEventType()의 반환 값, 현재 읽은 항목에 따른 유형 반환
- START_TAG(<>), END_TAG</>, TEXT 등
  - 3가지로 많이 구성됨

C. parser.text

- getText() 의 반환 값, TAG 사이의 Text 값

D. parser.next()

- XML 문서의 다음 항목으로 이동 ⇒ 무조건 다음으로 이동

E. parser.nextTag()

- XML 문서의 다음 TAG 로 이동. 이동 했는데, **TAG 가 아닐 경우 예외 발생**

F. parser.require(TAG_Type, namespace, TAG)

- 지금 읽고 있는 부분의 TAG의 유형 및 종류 확인
- 예측했던 태그가 아니면 예외 발생

4. XMLPullParser의 생성

```
class BoxOfficeParser {
    private val ns: String? =null
    companion object { // ① 어떤 TAG를 잘라낼지 함수로 정함
        val FAULT_RESULT ="faultResult"    // OpenAPI 결과에 오류가 있을 때에 생성하는 정보를 위해 지정
        val DAILY_BOXOFFICE_TAG ="dailyBoxOffice"
        val RANK_TAG ="rank"
        val TITLE_TAG ="movieNm"
        val OPEN_DATE_TAG ="openDt"
    }
    @Throws(XmlPullParserException::class, IOException::class)
    fun parse(inputStream: InputStream?) : List<Movie> {
        inputStream.use { inputStream ->
            val parser : XmlPullParser = Xml.newPullParser()
            /*Parser 의 동작 정의, next() 호출 전 반드시 호출 필요*/
            parser.setFeature(XmlPullParser.FEATURE_PROCESS_NAMESPACES, false)
            /* Paring 대상이 되는 inputStream 설정 */
            parser.setInput(inputStream, null)
            /*Parsing 대상 태그의 상위 태그까지 이동*/
            while (parser.name !="dailyBoxOfficeList") { // ② 파싱하는 내용 중에 내가 잘라 내려고 하는 태그의 윗부분 태그까지이동
                parser.next()
            }
            return readBoxOffice(parser)
        }
    }
```

5. 세부 parsing 함수1

1) ☆ 대상 항목 탐색 ★

```
@Throws(XmlPullParserException::class, IOException::class)
private fun readBoxOffice(parser: XmlPullParser) : List<Movie> {
      val movies = mutableListOf<Movie>() // DTO를 저장할 리스트 생성
      parser.require(XmlPullParser.START_TAG, ns, "dailyBoxOfficeList") // 현재 TAG를 확인. 아닐 경우 예외발생
      while(parser.next() != XmlPullParser.END_TAG) {
        if (parser.eventType != XmlPullParser.START_TAG) {
            continue
             }
        if (parser.name == DAILY_BOXOFFICE_TAG) {  // 항목을 표현하는 TAG를 찾았을 경우 세부 내용 Parsing
            movies.add( readDailyBoxOffice(parser) )
        } else {
            skip(parser) // 대상 태그가 아닐 경우 다음으로 이동
        }
    }
      return movies // 생성한 List<DTO> 반환
}
```

- parser.eventType : 현재 Parser가 읽은 유형의 종류 확인

6. 세부 parsing 함수 2

1) 항목의 관심 태그 부분 parsing

```
@Throws(XmlPullParserException::class, IOException::class)
private fun readDailyBoxOffice(parser: XmlPullParser) : Movie {
    parser.require(XmlPullParser.START_TAG, ns, "dailyBoxOffice")
      var rank : Int? =null
    var title : String? =null
    var openDate: String? =null

      while (parser.next() != XmlPullParser.END_TAG) {
        if (parser.eventType != XmlPullParser.START_TAG) {
            continue
             } when (parser.name) {
            RANK_TAG -> rank = readTextInTag(parser, RANK_TAG).toInt()
            TITLE_TAG -> title = readTextInTag(parser, TITLE_TAG)
            OPEN_DATE_TAG -> openDate = readTextInTag(parser, OPEN_DATE_TAG)
            else -> skip(parser) // 관심 TAG가 아닐 경우 다음으로 이동
        }
    }
    return Movie (rank, title, openDate) // 필요한 정보를 다 읽은 후 DTO로 저장하여 반환
}
```

- readTextInTag : 관심 TAG를 찾았을 경우. 태그 사이의 TEXT 확인
- toInt() : 필요한 경우 형 변환

7. 세부 parsing 함수 3

1) TEXT 읽기 함수

```
@Throws(IOException::class, XmlPullParserException::class)
private fun readTextInTag (parser: XmlPullParser, tag: String): String {
    parser.require(XmlPullParser.START_TAG, ns, tag)
      var text =""
    if (parser.next() == XmlPullParser.TEXT) {
        text = parser.text  // 현재 TEXT 읽기
        parser.nextTag() // 다음 TAG읽기, TAG가 아닐 경우 예외 발생
    }
    parser.require(XmlPullParser.END_TAG, ns, tag)
    return text
}
```

2) 다음 TAG로 넘어가기 함수

```
@Throws(XmlPullParserException::class, IOException::class)
private fun skip(parser: XmlPullParser) {
    if (parser.eventType != XmlPullParser.START_TAG) {
        throw IllegalStateException()
    }
    var depth =1
      while (depth !=0) {
       when (parser.next()) {
           XmlPullParser.END_TAG -> depth--
           XmlPullParser.START_TAG -> depth++
       }
    }
}
```

III. Coroutine 적용

1. Coroutine을 적용한 Network 사용

A. Coroutine 사용

```
        mainBinding.btnSearch.setOnClickListener {
            val date = mainBinding.etDate.text.toString()

            CoroutineScope(Dispatchers.Main).launch{
                val def = async(Dispatchers.IO) { // 네트워크 요청 + 응답 받기
                    var movies : List<Movie>? =null
                    try {
                        movies = networkDao.downloadXml(date)
                    } catch (e: IOException) {
                        Log.d(TAG, e.message?: "null")
                        null
                    } catch (e: XmlPullParserException) {
                        Log.d(TAG, e.message?: "null")
                        null
                    }
                    movies // 저장
                }
                adapter.movies = def.await() //결과를 받아옴
                adapter.notifyDataSetChanged()
            }
        }
```

- CoroutineScope(Dispatchers.Main).launch : Main Thread 상에 Coroutine 실행
  - 반환값 없음
- async(Dispatchers.IO)
  - IO Thread에 Coroutine 실행 후 실행 결과 반환
  - Deffered<T>
- def.await()
  - async의 반환값을 deffered.await()으로 확인 대기
  - 결과 전달 시 수행

B. 코루틴이 수행될 스레드의 종류

- Default : 시간이 많이 걸리는 작업
- IO Thread : DB, Network
- Main Thread : 일반적인 UI

2. 흐름 정리

A. 안드로이드 앱에서 코루틴을 사용하여 네트워크 요청하기

① 메인 액티비티에서 네트워크 요청
② 코루틴(IO Thread)를 사용하여 네트워크 요청 처리
③ 결과를 받아와 화면에 출력

B. xml 다운로드 및 파싱 과정

① 네트워크 매니저를 통해 xml 다운로드
② 다운로드한 xml에서 원하는 항목 파싱
③ 파싱한 결과를 뷰티오?로 화며에 표시

C. 내가 대충 정리

① 어떤 tag를 질라낼지 함수로 정함
② 파싱하는 내용중에 내가 질라 내려고하는 태그의 윗부분의 태그까지 이동
③ 맞는 항목의 태그인지 검사하고 내부에 있는 TAG를 파싱하는 함수 호출
④ 내가 관심있는 TAG를 만나면 값을 읽는 작업
⑤ dto로 읽어 반환