**기술 문서 보고서**

모바일앱프로그래밍1 PROJECT – 6조

**헬스 케어 어플**



001분반 6조

2016117401 김기훈

2018116035 신진철

2018111175 심현윤

2018113185 이소정

**1. 인터페이스 구성**

해당 어플의 화면은 viewpager2, drawerlayout, navigation view, tablayout을 사용하여 구성하였다. 총 화면의 갯수는 4가지이다. 만보기 기능을 가진 홈 화면과 달력에서 원하는 날짜를 선택하여 해당 기간 동안의 걸음 수를 출력하는 기록 화면, 그리고 특정 조건에 해당하면 보상 뱃지를 획득할 수 있는데, 이 뱃지를 출력하는 보상화면과 초기 화면으로 클라이언트의 이름과 키, 몸무게 정보를 입력 받는 정보화면이 있다.

우선 초기 화면에서 클라이언트의 정보를 받아서 인텐트 기능을 이용하여 drawlayout 화면으로 정보 데이터를 전달해준다. 해당 구현 코드는 다음과 같다.

데이터 발신 액티비티

binding.done.setOnClickListener {  
 val intent=Intent(this,MainActivity::class.java)  
 intent.putExtra("height",binding.height.text.toString())  
 intent.putExtra("weight",binding.weight.text.toString())  
 intent.putExtra("name",binding.name.text.toString())  
 startActivity(intent)  
}

데이터 수신 액티비티

val navigationView=binding.navView  
navigationView.setNavigationItemSelectedListener(this)

override fun onNavigationItemSelected(item: MenuItem): Boolean=when(item.*itemId*) {  
  
 R.id.*info*->{  
 Toast.makeText(this,"키 : ${*intent*.getStringExtra("height")}cm " +  
 "몸무게 : ${*intent*.getStringExtra("weight")}kg", Toast.*LENGTH\_LONG*).show()  
 true  
 }  
 R.id.*aboutus* -> {  
 Toast.makeText(this, "We are Group 6", Toast.*LENGTH\_SHORT*).show()  
 true  
 }  
 R.id.*contact* -> {  
 Toast.makeText(this, "please contact 031-120",Toast.*LENGTH\_LONG*).show()  
 true  
 }  
 R.id.*infochange* -> {  
 finish()  
 true  
 }  
 else -> false  
}

이처럼 데이터를 수신하여 navigationview와 drawerlayout에 출력해주어야 하기 떄문에, setNavigationItemSelectedListener를 사용하여 내비게이션 뷰를 등록한 후에 onNavigationItemSeleted 함수를 사용하여 내비게이션 뷰 내의 특정 항목을 클릭했을 때, 인텐트를 통해 받아온 정보를 toast로 출력하도록 구현하였다. 정보수정인 R.id.infochange를 선택하면 다시 정보를 입력받는 초기화면으로 되돌아 갈 수 있다.

**2. 만보기 기능**

만보기 기능의 경우 안드로이드 스튜디오에서 제공하는 SensorEventListener를 사용하여 구현하였다. 이 중에서 만보기 기능을 위해 클라이언트의 움직임을 인지하는 센서는 크게 두 가지가 있다. TYPE\_STEP\_COUNTER와 TYPE\_STEP\_DETECTOR과 있는데, 이 두 센서는 작동 방법은 모두 동일하나 리턴 값만 다르다는 차이점이 있다. TYPE\_STEP\_COUNTER는 TYPE\_STEP\_DETECTOR와 달리 걷기 동작을 감지해 계속 리턴 값을 더해주기 때문에 우리 조는 TYPE\_STEP\_COUNTER 센서를 사용하기로 했다.

만보기 값을 위해 사용할 변수는 다음과 같이 선언하였다.

var sm: SensorManager? = null

var sensor\_accelerometer: Sensor? = null

var myTime1: Long = 0

var myTime2: Long = 0

var x = 0f

var y = 0f

var z = 0f

var lastX = 0f

var lastY = 0f

var lastZ = 0f

val walkThreshold = 455 // 걷기 인식 임계 값

var acceleration = 0.0

var startTime: Long = 0

var currentTime: Long = 0

var elapsedTime = 0 //경과시간

var handler = ValueHandler()

var walkstart:String?= "0"

var initTime = 0L

var pauseTime = 0L

sm = getSystemService(*SENSOR\_SERVICE*) as SensorManager //센서에 접근

sensor\_accelerometer = sm!!.getDefaultSensor(Sensor.*TYPE\_ACCELEROMETER*) // 가속도 센서

SensorEventListener를 implement하게 되면 onSensorChanged()와 onAccuracyChanged() 함수를 오버라이드 할 수 있게 된다. onSensorChanged() 함수는 실제 센서의 작동과 관련된 함수이다. 센서가 동작을 감지하게 되면 이벤트를 발생하여 onSensorChanged() 함수에 값을 전달해준다. 따라서 이벤트가 발생하여 함수에 전달되는 값은 걸음 수를 나타내는 TextView에 전달해주었다. 해당 코드 구현은 다음과 같다.

override fun onSensorChanged(event: SensorEvent) {

if (event.sensor.*type* == Sensor.*TYPE\_ACCELEROMETER*) { //센서의 타입이 가속도 센서일 경우

myTime2 = System.currentTimeMillis()

val gab = myTime2 - myTime1 //시간차

if (gab > 90) {

myTime1 = myTime2

x = event.values[0]

y = event.values[1]

z = event.values[2]

acceleration = (Math.abs(x + y + z - lastX - lastY - lastZ) / gab \* 9000).toDouble() //이동속도공식

if (acceleration > walkThreshold) {

if(walkstart=="1"){

walkingCount += 1.0.toInt()

}

}

lastX = event.values[0]

lastY = event.values[1]

lastZ = event.values[2]

}

}

}

override fun onAccuracyChanged(sensor: Sensor, accuracy: Int) {}

// 스레드 클래스 생성

internal inner class BackgroundThread : Thread() {

var value = 0

var running = false

override fun run() {

running = true

while (running) {

value += 1

val message = handler.obtainMessage()

val bundle = Bundle()

bundle.putInt("value", value)

message.*data* = bundle

handler.sendMessage(message)

elapsedTime = currentTime.toInt() - startTime.toInt()

currentTime = System.currentTimeMillis()

try {

sleep(1000)

} catch (e: Exception) {

}

}

}

}

센서는 다른 앱을 사용하기 위해서 일시 중지 되거나 화면을 종료해도 계속 작동하고 있으며 전력을 소모하고 있다.

따라서 이를 방지하기 위해 앱을 완전히 종료하거나 다른 앱에 의해 일시 중지되는 경우, 센서의 작동을 중지 시키고 이벤트를 받아 다시 센서 관련 앱이 활성화 되었을 경우에 작동이 되도록 구현하였다. 이를 구현하기 위해 생명 주기 함수인 onPause()와 onResume()을 사용해주었다. 해당 구현 코드는 다음과 같다.

override fun onResume() {

super.onResume()

sm!!.registerListener(this, sensor\_accelerometer,

SensorManager.*SENSOR\_DELAY\_NORMAL*) //센서등록, 센서 읽어오는 속도

}

override fun onPause() {

super.onPause()

sm!!.unregisterListener(this) //센서 리스너 해제

}

**3. 공유 기능(스크린샷)**

## 공유기능을 구현하기 위해 Android Sharesheet 기능을 사용하였다. 해당 코드는 아래와 같다.

val shareIntent = Intent()  
shareIntent.*action* = Intent.*ACTION\_SEND*  
*shareIntent.putExtra(Intent.EXTRA\_STREAM*, Uri.fromFile(screenShot))  
shareIntent.*type* = "image/png"  
startActivity(Intent.createChooser(shareIntent, "공유하기"))

Intent의 ACTION\_SEND작업을 설정하고, 데이터를 공유하기 위해 스크린샷을 통해 이미지를 생성하고 Uri를 생성하였고, startActivity를 통해 공유 화면을 호출하였다.

## 또한 화면을 공유하기 위해 스크린샷 기능을 구현하였다.

fun ScreenshotButton(view: View?) {  
 val rootView = *window*.*decorView* //현재 화면 지정  
 val screenShot = ScreenShot(rootView)  
 //사진 저장  
 if (screenShot != null) {  
 sendBroadcast(  
 Intent(  
 Intent.*ACTION\_MEDIA\_SCANNER\_SCAN\_FILE*,  
 Uri.fromFile(screenShot)  
 )  
 ) //갤러리에 추가  
 }  
}

먼저 공유버튼을 클릭하면 현재 화면을 지정하고, 지정된 화면을 캡처하고 저장한다.

이 기능을 위해서는 권한 설정이 필요하기 때문에 manifest 파일에 아래의 코드를 첨부해주어야 한다

.

<uses-permission android:name="android.permission.READ\_EXTERNAL\_STORAGE" />  
<uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE" />

읽기, 쓰기 권한을 androidmanifest파일에 추가하고,

companion object {  
 private const val REQUEST\_EXTERNAL\_STORAGE = 1  
 private val PERMISSION\_STORAGE = *arrayOf*(  
 Manifest.permission.*WRITE\_EXTERNAL\_STORAGE*,  
 Manifest.permission.*READ\_EXTERNAL\_STORAGE*  
)  
  
 //권한 부여 확인  
 fun verifyStoragePermission(activity: Activity?) {  
 val permission = ActivityCompat.checkSelfPermission(  
 activity!!,  
 Manifest.permission.*WRITE\_EXTERNAL\_STORAGE*  
)  
 if (permission != PackageManager.*PERMISSION\_GRANTED*) {  
 ActivityCompat.requestPermissions(  
 activity,  
 PERMISSION\_STORAGE,  
 REQUEST\_EXTERNAL\_STORAGE  
 )  
 }  
 }  
}

verifyStoragePermission(this)  
val builder = StrictMode.VmPolicy.Builder()  
StrictMode.setVmPolicy(builder.build())

권한이 부여되었는지 확인하기 위해 위와 같은 함수를 작성하고 호출하였다.

다음으로 스크린샷 기능을 구현하기 위해 다음과 같은 코드를 사용하였다.

fun ScreenshotButton(view: View?) {  
 val rootView = *window*.*decorView* //현재 화면 지정  
 val screenShot = ScreenShot(rootView)  
 //사진 저장  
 if (screenShot != null) {  
 sendBroadcast(  
 Intent(  
 Intent.*ACTION\_MEDIA\_SCANNER\_SCAN\_FILE*,  
 Uri.fromFile(screenShot)  
 )  
 ) //갤러리에 추가  
 }  
}

fun ScreenShot(view: View): File? {  
 view.*isDrawingCacheEnabled* = true //캐시 읽기  
 val screenBitmap = view.*drawingCache* //비트맵으로 변환  
 val filename = "screenshot" + System.currentTimeMillis() + ".png" //스크린샷 파일 이름 설정  
 val file = File(  
 Environment.getExternalStorageDirectory().toString() + "/Pictures",  
 filename  
 ) //스크린샷 저장 경로 설정  
 var os: FileOutputStream? = null  
 try {  
 os = FileOutputStream(file)  
 screenBitmap.compress(Bitmap.CompressFormat.*PNG*, 90, os) //비트맵 > PNG파일  
 os.close()  
 } catch (e: IOException) {  
 e.printStackTrace()  
 return null  
 }  
 view.*isDrawingCacheEnabled* = false  
 return file  
}

먼저 공유 버튼을 누르면 현재 화면을 지정하고, 스크린샷 함수를 호출해 이미지를 저장한다.

스크린샷 함수에서는 화면을 비트맵으로 변환하고, PNG파일로 다시 변환해 지정한 파일 이름과 경로로 저장한다.

사진을 저장하지 않고 공유하면 간헐적으로 오류가 발생하였기 때문에 저장하고 공유하는 방식을 택하였고, 정상적으로 잘 작동하는 것을 확인할 수 있다.

**4. 데이터베이스 기능**

안드로이드에서 데이터베이스를 사용하기 위해서는 SQLite를 사용한다. 외부 서버를 사용하지 않고 안드로이드 자체의 기능을 사용하는 것이기 때문에 SQLite를 사용하게 되었다.

var dbName = "final1.db"  
var tableName = "member"  
private lateinit var db: SQLiteDatabase

먼저 데이터베이스와 테이블의 이름을 정한 후, 다음과 같은 코드를 작성하여 데이터베이스를 생성한다.

db = this.openOrCreateDatabase(dbName, *MODE\_PRIVATE*, null)  
db.execSQL("CREATE TABLE IF NOT EXISTS $tableName(num integer primary key autoincrement, date text not null, count integer, km float);")

openOrCreateDatabase는 데이터베이스가 존재하지 않을 때 데이터베이스를 생성한다.

또한 SQL언어 CREATE TABLE IF NOT EXISTS 또한 테이블이 존재하지 않을 때 데이터베이스를 생성한다.

생성한 데이터베이스를 통해 데이터를 저장하고, 수정하기 위해서는 다음과 같은 코드를 사용한다.

//데이터 삽입  
override fun insert(date: String, count: Int, km: Float) {  
 if(date=="") {  
 val currentdate = this.date  
 db.execSQL("INSERT INTO $tableName(date, count, km) VALUES('$currentdate', '$count', '$km')")  
 } else {  
 db.execSQL("INSERT INTO $tableName(date, count, km) VALUES('$date','$count', '$km');")  
 }  
}

//데이터 탐색  
val cursor = db.rawQuery("SELECT \* FROM $tableName", null) {  
val buffer = StringBuffer()  
while (cursor.moveToNext()) {  
 val num = cursor.getInt(0)  
 val date = cursor.getString(1)  
 val count = cursor.getInt(2)  
 val km = cursor.getFloat(3)

}

//데이터 업데이트, 삭제

db.execSQL("UPDATE $tableName SET count=$*count* WHERE date=?", *arrayOf*(date))

db.execSQL("DELETE FROM $tableName WHERE date=?", *arrayOf*(date))

**5. 막대그래프(데이터 해시 값 받기)**

안드로이드 차트 기능을 제공하는 라이브러리인 MPAndroidChart를 사용하여 구현하였다.

3번 기능인 데이터베이스의 값을 끌어와서 최근 일주일 간의 걸음 수를 그래프로 나타내었다.

사용한 코드와 설명은 다음과 같다.

먼저 안드로이드 스튜디오 내에는 MPAndroidChart 라이브러리가 없기 때문에 setting.gradle 파일에 다음을 추가하고

allprojects {

repositories {

google()

mavenCentral()

jcenter()

maven { url "<https://jitpack.io>" }

}

}

모듈의 gradle 파일에 다음을 추가하였다.

dependencies {

implementation 'com.github.PhilJay:MPAndroidChart:v3.1.0'

}

이로써 BarChart를 사용할 수 있으며, 차트를 삽입할 xml layout에 다음과 같은 태그를 입력하였다.

<com.github.mikephil.charting.charts.BarChart

android:id="@+id/chart"

android:layout\_width="match\_parent"

android:layout\_marginTop="300dp"

android:layout\_height="190dp" />

해당 layout의 kt 파일 구현 코드는 다음과 같다.

val entry\_chart = ArrayList<BarEntry>() // 데이터를 담을 Arraylist  
var barChart = binding.chart as BarChart  
val barData = BarData() // 차트에 담길 데이터  
for (i in 0..5) {  
 entry\_chart.add(BarEntry((i+1).toFloat(), arr[i+1]))//데이터를 담아준다.  
}  
val barDataSet = BarDataSet(entry\_chart, " ") // 데이터가 담긴 Arraylist 를 BarDataSet 으로 변환한다.  
barDataSet.*color* = Color.rgb(199,177,153) // 해당 BarDataSet 색 설정 :: 각 막대 과 관련된 세팅은 여기서 설정한다.  
barDataSet.*valueTextSize*=13f  
barData.addDataSet(barDataSet) // 해당 BarDataSet 을 적용될 차트에 들어갈 DataSet 에 넣는다.  
barChart!!.*data* = barData // 차트에 위의 DataSet 을 넣는다.  
barChart!!.invalidate() // 차트 업데이트  
barChart!!.setTouchEnabled(false) // 차트 터치 불가능하게  
barChart!!.setDrawGridBackground(false)  
barChart!!.setDrawBorders(false)  
barChart!!.*axisLeft*.setDrawGridLines(false)  
barChart!!.*axisRight*.setDrawGridLines(false)  
barChart!!.*axisRight*.*isEnabled*=false  
barChart!!.*axisLeft*.*isEnabled*=false  
barChart!!.*xAxis*.*isEnabled*=false  
barChart!!.*xAxis*.setDrawAxisLine(false)  
barChart!!.setMaxVisibleValueCount(6)  
barChart!!.*description*.*isEnabled*=false  
barChart!!.*legend*.*isEnabled*=false

주석이 없는 그 외의 기능은 모두 차트의 속성에 관한 함수이다.

깔끔하게 차트를 구성하기 위해서 좌표는 모두 제거하였으며, 대신 막대그래프의 값을 확인할 수 있도록 차트 위에 작게 값을 적었다.

**6. 프래그먼트, 액티비티 간 데이터 상호작용 기능(번들, 인텐트, 트랜잭션)**

우선 프래그먼트에서 액티비티의 데이터를 사용하기 위해 interface를 생성하였다.

interface AccessDB {  
 public fun insert(date:String, count:Int, km:Float)  
 public fun inputHash(hash: HashMap<String, Int>)  
 public fun FindKmSum():Float  
 public fun inputFloatHash(hash: HashMap<String, Float>)  
 public fun toast(message: String)  
 public fun FindCountSum() : Int  
 public fun FindSnsValue() : Int  
 public fun FindidxSum() : Int  
}

인터페이스에 선언한 함수들은 모두 액티비티에서 override하여 함수의 내용을 추가해 주었다. 이후 프래그먼트에서 이 함수들을 사용하기 위해서는 다음과 같이 사용할 수 있다.

(예시)

(*activity* as AccessDB).inputHash(*hash*)

val km = (*activity* as AccessDB).FindKmSum()

다음으로 액티비티에서 프래그먼트에 직접 데이터를 보내기 위해 번들, 트랜잭션을 사용하였다. 코드는 아래와 같다.

//데이터 프래그먼트로 넘기기  
val initialFragment = InitialFragment()  
val bundle = Bundle()  
bundle.putInt("key", walkingCount)  
initialFragment.*arguments*=bundle  
val transaction=*supportFragmentManager*.beginTransaction()  
transaction.replace(R.id.*initialLayout*, initialFragment)  
transaction.commit()

이 코드는 먼저 번들에 키와 Value를 저장하고, 프래그먼트의 arguments에 번들을 넘긴다. 그 다음 프래그먼트에서 그 값을 받기 위해 transaction.replace로 프래그먼트를 새 프래그먼트로 교체한다.

또한 업적 달성 후 액티비티에서 프래그먼트를 새로고침하여 업적 달성 결과를 보이기 위해 초기화 함수를 생성하여 사용하였다. 코드는 다음과 같다.

fun refreshFragment(fragment: Fragment, fragmentManager: FragmentManager) {  
 var ft: FragmentTransaction = fragmentManager.beginTransaction()  
 ft.detach(fragment).attach(fragment).commit()  
}

트랜잭션을 이용하여 읽어온 프래그먼트를 detach->attach하여 새로고침하는 함수이다.

마지막으로 자정이 지나 데이터를 초기화하고 저장하기 위해서 다음과 같은 코드를 작성하였다.

//날짜 지나면 초기화 부분  
val intentFilter = IntentFilter(Intent.*ACTION\_DATE\_CHANGED*)  
val receiver = object : BroadcastReceiver() {  
 override fun onReceive(p0: Context?, p1: Intent?) {  
 toast("날짜가 변경되었습니다.")  
 //db에 데이터 삽입  
 …

//초기화

walkingCount=0  
  
 binding.chronometer.*text*="00:00"  
 myTime1= 0  
 myTime2= 0  
 acceleration = 0.0  
 startTime = 0  
 currentTime = 0  
 elapsedTime = 0 //경과시간  
 walkstart = "0"  
 initTime = 0L  
 pauseTime = 0L  
  
 //데이터 프래그먼트로 넘기기  
 ...  
 }  
}  
registerReceiver(receiver, intentFilter)

우선 intentFilter의 ACTION\_DATE\_CHANGED 설정을 사용하여 날짜가 바뀌었다는 것을 인식하면 onReceive를 호출하여 그 안에 있는 코드를 실행한다. 이 어플리케이션에서는 우선 그날의 데이터를 저장한 후, 데이터를 초기화시킨다. 앞서 말한 액티비티에서 프래그먼트로 데이터를 넘기는 기술을 사용하여 데이터를 송신한다.

**7. 달력 기능**

먼저 달력 기능의 범위를 설정하기 위해 dateRangePicker라는 변수를 만들어주었다.

val dateRangePicker =MaterialDatePicker.Builder.dateRangePicker()

.setTitleText("검색 기간을 골라주세요")

.build()

dateRangePicker.show(childFragmentManager, "date\_picker")

그 후 달력 UI에서 입력 받은 날짜 정보들을 selection?.first ?:0에서 처음 날짜를 입력받아 startDate에 들어가며 selection?.second ?: 0에서 마지막 날짜를 입력받아 endDate에 들어가게 된다. 이 부분에서 startcheck, endcheck 변수를 만들었고 rangedate는 오늘 날짜를 입력받는 데이터이다. 그래서 startcheck, endcheck변수와 rangedate를 비교하여 rangedate를 넘어가는 변수가 하나라도 있으면 틀리다는 텍스트가 나오도록 구현하였다.

//날짜 클릭해서 받는 부분  
binding.btn.setOnClickListener **{**  
(*activity* as AccessDB).inputHash(*hash*)  
 (*activity* as AccessDB).inputFloatHash(*kmhash*)  
 val dateRangePicker =  
 MaterialDatePicker.Builder.dateRangePicker()  
 .setTitleText("검색 기간을 골라주세요")  
 .build()  
 dateRangePicker.show(*childFragmentManager*, "date\_picker")  
 dateRangePicker.addOnPositiveButtonClickListener(object : MaterialPickerOnPositiveButtonClickListener<Pair<Long, Long>> {  
 override fun onPositiveButtonClick(selection: Pair<Long, Long>?) {  
 val calendar = Calendar.getInstance()  
 var startDate="0"  
 var endDate="0"  
 var startcheck=0  
 var endcheck =0  
 var sum = 0  
 var kmsum = 0.0f  
  
 calendar.*timeInMillis* = selection?.first ?: 0  
 startDate = SimpleDateFormat("yyyy-MM-dd").format(calendar.*time*).toString()  
 startcheck=SimpleDateFormat("yyyyMMdd").format(calendar.*time*).*toInt*()  
 calendar.*timeInMillis* = selection?.second ?: 0  
 endcheck = SimpleDateFormat("yyyyMMdd").format(calendar.*time*).*toInt*()  
 endDate = SimpleDateFormat("yyyy-MM-dd").format(calendar.*time*).toString()  
  
 if(startcheck <= rangedate && endcheck <= rangedate){  
 binding.starttext.*text* = startDate  
 binding.endtext.*text*=endDate  
 sum = walkadd(startDate,endDate)  
 kmsum = kmadd(startDate, endDate)  
 binding.walktxt.*text*=sum.toString()  
 binding.kmtxt.*text*=kmsum.toString()  
 }  
 else{  
 binding.starttext.*text* = "false range"  
 binding.endtext.*text*="false range"  
 binding.walktxt.*text*="false data"  
 binding.kmtxt.*text*="false data"  
 }  
 }  
 })  
}