[모바일프로그래밍] 프로젝트 - <날씨 알림이>

스마트ict융합공학과

201714286 조예린

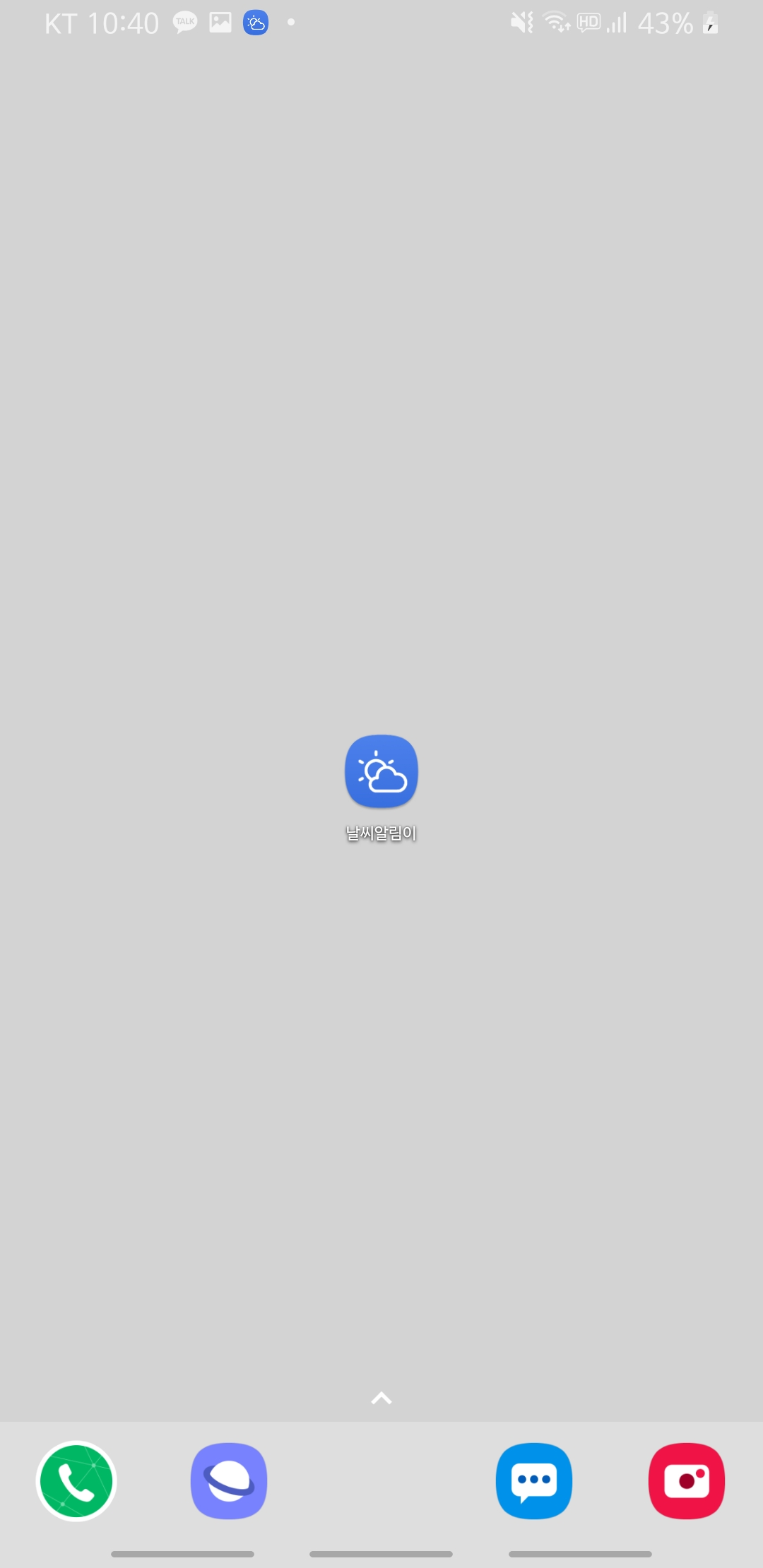
<날씨 알림이> 프로젝트 소개:

날씨 앱과 일정 앱을 합쳐 사용자가 일정에 해당하는 날짜와 시간, 그리고 장소를 등록하면 일정이 시간 순서대로 타임라인에 표시되며 그 시간대에 그 장소에 해당하는 날씨 정보가 함께 보여집니다. 앱을 새로 실행하거나 업데이트 버튼을 누르면 날씨 정보가 업데이트 되며 일정을 수정, 삭제할 수도 있습니다. 사용자가 앱을 계속 확인하지 않아도 일정에 해당하는 날씨 예보가 변경되면 상태바로 알림을 받을 수 있습니다. 동선 지도를 통해 순서대로 일정의 동선을 실시간 날씨예보와 함께 확인할 수 있습니다.

발표 영상 : <https://youtu.be/u5YRA1V2zV4>

실행 영상 : <https://www.youtube.com/watch?v=FZ9uXUtS870&feature=youtu.be>

github주소 : <https://github.com/uyttyu7532/TourWeatherReminder>

1. 아이콘 및 스플래시 화면
2. 타임라인 뷰



materialtimelineview(timeline recyclerview) 라이브러리를 사용하였습니다. 각 일정 사이에는 타임라인이 연결되어 있어 처음 일정 전과 마지막 일정 후에는 타임라인 선이 필요하지 않기 떄문에 처음 일정과 마지막 일정을 별도로 지정해주어야 합니다.

**<MainActivity.kt>**

object resetSchedule {  
 fun resetAdapter() {  
 if (*ScheduleList*.size > 0) {  
 *ScheduleList*[0].isFirstItem = true  
 *ScheduleList*[*ScheduleList*.size - 1].isLastItem = true  
 }  
 *timelineRecyclerAdapter* = TimelineRecyclerAdapter()  
 *recyclerView*.*adapter* = *timelineRecyclerAdapter  
 recyclerView*.*layoutManager* = LinearLayoutManager(*mContext*)  
  
 for (i in 0 *until ScheduleList*.size) {  
 if (!*ScheduleList*[i].isFirstItem) {  
 *timelineRecyclerAdapter*.addTimepoint(Timepoint())  
 }  
 *timelineRecyclerAdapter*.addSchedule(*ScheduleList*[i], *ScheduleList*[i].isFirstItem)  
 }  
 }  
 }

**<WeatherItemDelegateAdapter.kt>**

if (item.isLastItem) {  
 item\_weather\_timeline.position = MaterialTimelineView.POSITION\_LAST  
}  
if (item.isFirstItem) {  
 item\_weather\_timeline.position = MaterialTimelineView.POSITION\_FIRST  
}

어댑터에서는 각 일정 정보(일정 이름, 날짜 및 시간, 선택한 장소)와 해당하는 날씨 정보(날씨, 기온, 강수량)를 배경 색상, 아이콘과 함께 바인딩해줍니다. 또한 각 일정을 클릭하면 일정을 수정 / 삭제 할 수 있는 다이얼로그가 생성되어 보여집니다.

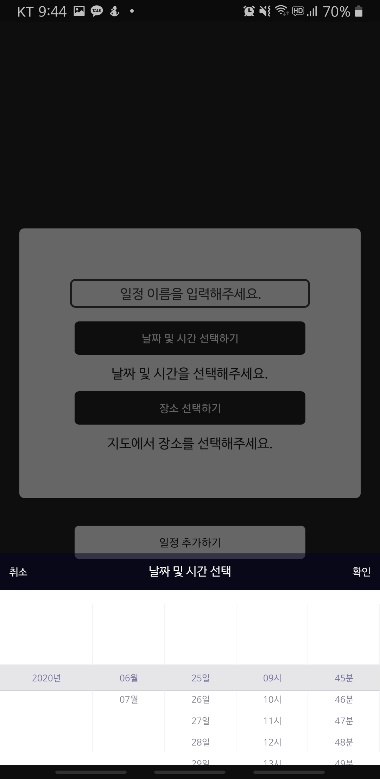
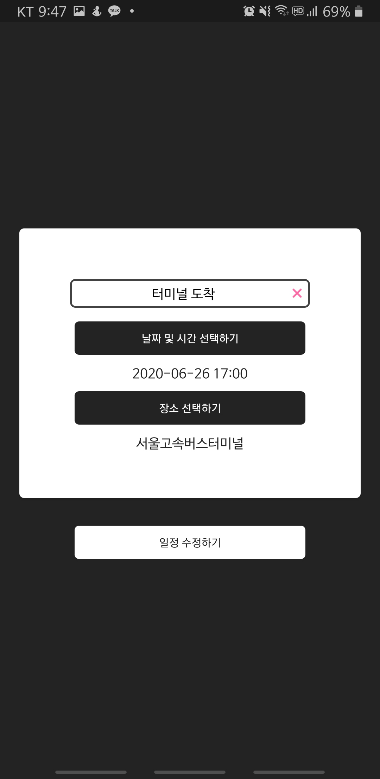
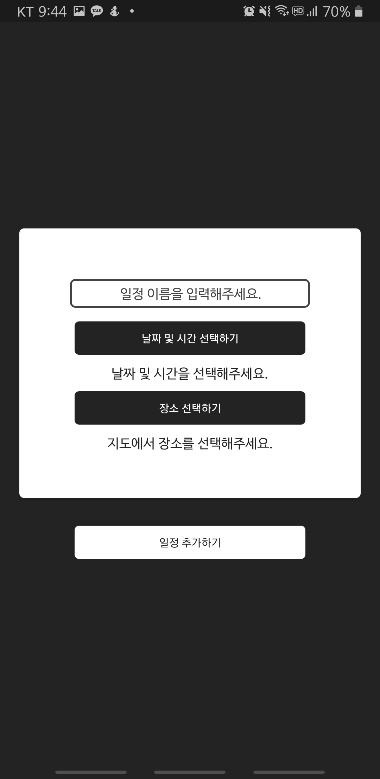
메인 화면으로 들어왔을 때, 메인 화면의 새로고침 버튼을 눌렀을 때, 메인 화면에서 화면을 밑으로 swipe했을 때, 등록한 주기의 시간마다 전체 일정의 날씨 정보가 업데이트 됩니다.

**<MainActivity.kt>**

fun updateAllSchedule() {  
 Handler().postDelayed(*Runnable* **{** *notificationContent* = ""  
 *notificationResultCnt* = 0  
 // 모든 날씨 정보를 다시 받아오는 작업  
 for (i in *ScheduleList*) {  
 MainAsyncTask(*applicationContext*).execute(i)  
 }  
 **}**, 1000  
 )  
}

1. 일정 추가 / 수정 화면

사용자가 일정 이름을 입력하고, 날짜 및 시간, 장소를 선택하여 일정을 추가 또는 수정할 수 있습니다. 일정 이름이 DB의 기본키로 저장되어 있기 때문에 같은 이름의 일정을 등록하면 일정이 수정됩니다. 일정을 클릭하여 수정 화면으로 들어온 경우 클릭한 일정 정보가 입력된 상태로 일정 추가 화면이 보여집니다.



**<AddActivity.kt>**

// 수정 모드라면 기존 일정 정보 표시하기  
fun isModify() {  
 addBtn.setText("일정 수정하기")  
 editTitleText?.setText(*modifyList*[0])  
 selectedPlaceText?.setText(*modifyList*[1])  
 latitude = *modifyList*[2].*toDouble*()  
 longitude = *modifyList*[3].*toDouble*()  
 timestamp = *modifyList*[4].*toLong*()  
 selectedDateText?.setText(*modifyList*[5])  
}

날짜 및 시간 선택은 TimePickerDialog 라이브러리를 사용하였습니다. 날씨 API가 현재로부터 7일 이후까지의 날씨 정보를 제공하기 때문에 일정 날짜는 현재부터 7일 이후의 날짜까지만 선택할 수 있습니다.

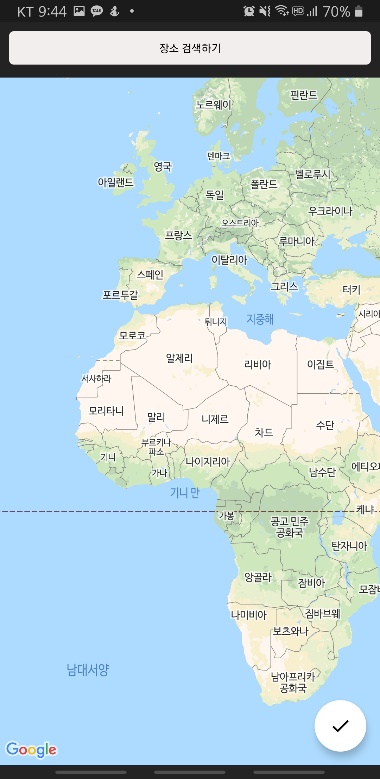
**<AddActivity.kt>**

// datePicker 다이얼로그 표시  
fun createDatePickerDialog()

// 선택한 시간 정보 저장 및 표시  
override fun onDateSet(  
 timePickerDialog: TimePickerDialog,  
 millseconds: Long  
)

// 선택한 시간 정보를 timestamp에서 날짜 형식에 맞춰 변경  
fun getDateToString(time: Long): String

사용자가 장소를 선택할 수 있는 방법은 두 가지입니다. VanillaPlacePicker 라이브러리를 이용하여 장소를 검색하여 선택할 수 있고, 지도를 이동, 확대 시켜 원하는 장소를 직접 클릭하여 선택할 수도 있습니다.



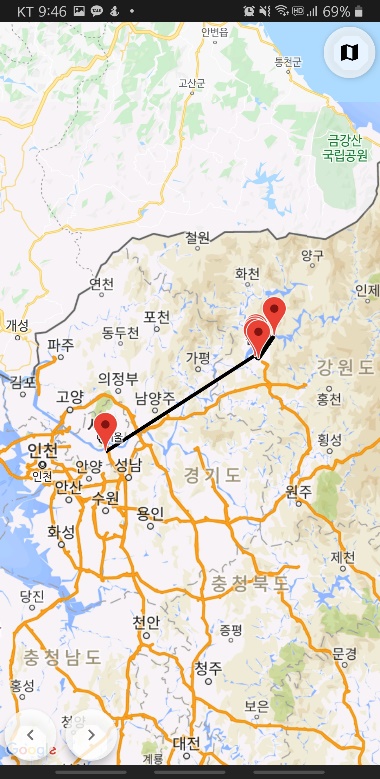
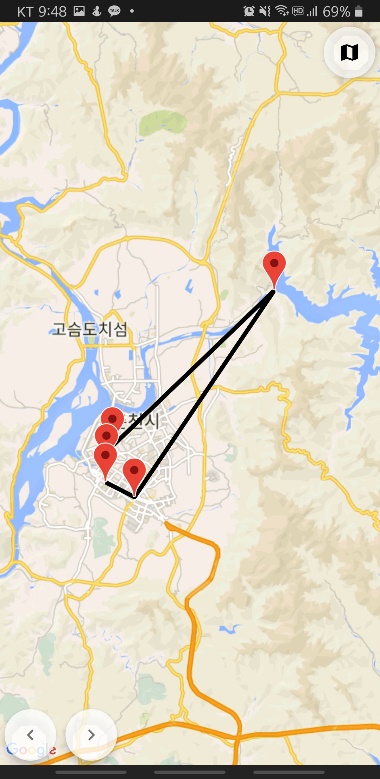
**<SearchPlaceActivity.kt>**

var intent = VanillaPlacePicker.Builder(this)  
 .with(PickerType.AUTO\_COMPLETE)  
 .setPickerLanguage(PickerLanguage.ENGLISH)  
 .build()  
startActivityForResult(intent, KeyUtils.REQUEST\_PLACE\_PICKER)

**<MapsActivity.kt>**

// 지도에서 직접 클릭하여 장소 선택  
fun initMapListener() {  
 mMap.setOnMapClickListener **{** mMap.clear()  
 latitude = **it**.latitude  
 longitude = **it**.longitude  
 mMap.addMarker(  
 MarkerOptions().position(**it**)  
 .icon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.*HUE\_VIOLET*))  
 .title("이 위치로 선택하시겠습니까?")  
 ).showInfoWindow()  
 **}**}

1. 동선 지도 화면

일정을 한 눈으로 확인할 수 있게 지도에 동선을 추가하였습니다. 초기 화면으로 들어오면 시간 순서대로 전체 동선을 볼 수 있습니다.

<RouteActivity.kt>

// 전체 동선에 맞게 화면 범위 조정

val padding = 300 // offset from edges of the map in pixels  
var bounds = builder.build()  
cu = CameraUpdateFactory.newLatLngBounds(bounds, padding)  
googleMap.animateCamera(cu);

지도 위의 이전 또는 다음 버튼을 클릭하면 이전 또는 다음 일정의 장소로 카메라가 이동합니다. 또한 일정 마커를 클릭하면 일정 정보와 함께 날씨를 확인할 수 있습니다.

<RouteActivity.kt>

var markeroptions = MarkerOptions()  
 markeroptions  
 .position(  
 LatLng(  
 *ScheduleList*[i].latitude!!,  
 *ScheduleList*[i].longitude!!  
 )  
 )  
 .title("${i + 1}번째 일정")  
 .snippet(*ScheduleList*[i].weather + "\t" + *ScheduleList*[i].rain + "\t" + *ScheduleList*[i].title + "\t" + *ScheduleList*[i].date)  
  
 markerList.add(markeroptions)  
 builder.include(markeroptions.*position*)  
 }  
  
 val adapter = InfoWindowAdapter(this)  
 googleMap.setInfoWindowAdapter(adapter)

// 동선 표시  
lineOptions!!.add(  
 LatLng(  
 *ScheduleList*[i].latitude!!,  
 *ScheduleList*[i].longitude!!  
 )  
)

googleMap.addPolyline(lineOptions)

**<InfoWindowAdapter.kt>**

infoWeather.setImageResource(

weather.get(marker.*snippet*.*split*("\t")[0].*substring*(0,2))!!)  
  
infoRain.*text* = marker.*snippet*.*split*("\t")[1] + "mm/h"  
infoNum.*text* = marker.*title*infoTitle.*text* = marker.*snippet*.*split*("\t")[2]  
infoDate.*text* = marker.*snippet*.*split*("\t")[3]

1. 날씨 데이터 관련

openweathermap 에서 제공하는 One Call API를 사용하였습니다. 위치 정보(위도, 경도)와 함께 API를 호출하면 현재로부터 2일 동안의 시간별 예보, 7일동안의 일별 예보, GMT기준 시차 정보 등을 json 형태로 제공받을 수 있습니다.

과거의 일정에 대해서는 날씨 정보를 새로 가져오지 않으며, 등록한 일정이 현재로부터 2일 이내라면 시간별 예보를 파싱하고, 이후의 일정이라면 일별 예보를 파싱하여 사용자에게 보여줍니다.

날씨 정보는 총 8가지로, 각 날씨에 해당하는 색상과 아이콘 정보가 저장되어 있습니다.

**<ParsingWeather.kt>**

class MainAsyncTask(context: Context) : AsyncTask<ScheduleEntity, Unit, ScheduleEntity>() {

// 과거 일정인지, 2일 이내 일정인지 아닌지 확인 후 파싱함수 호출

override fun doInBackground(vararg params: ScheduleEntity): ScheduleEntity?

// 사용자가 등록한 일정 정보와 파싱한 날씨 정보를 합쳐서   
// title존재 여부에 따라 DB에 일정 추가 또는 수정  
override fun onPostExecute(result: ScheduleEntity)

// 파싱된 날씨 정보 중 가장 가까운 시간의 날씨 정보 반환  
fun findMinDiffDt(WeatherArray: ArrayList<WeatherData>): WeatherData

// 일별 날씨 파싱  
fun parseDaily(json: JSONObject, tempTime: String)

// 시간별 날씨 파싱  
fun parseHourly (json: JSONObject)

일정에 등록된 장소 기준으로 시차를 적용하여 날씨 정보를 가져오며, 일별 예보의 경우 아침, 낮, 저녁, 밤으로 나누어 날씨 정보가 제공되기 때문에 일정에 등록된 시간에 따라 다른 시간대의 날씨 정보를 가져오도록 하였습니다. 시간별 예보와 일별 예보 모두 파싱된 날씨 중 가장 가까운 시간의 날씨를 가져옵니다.

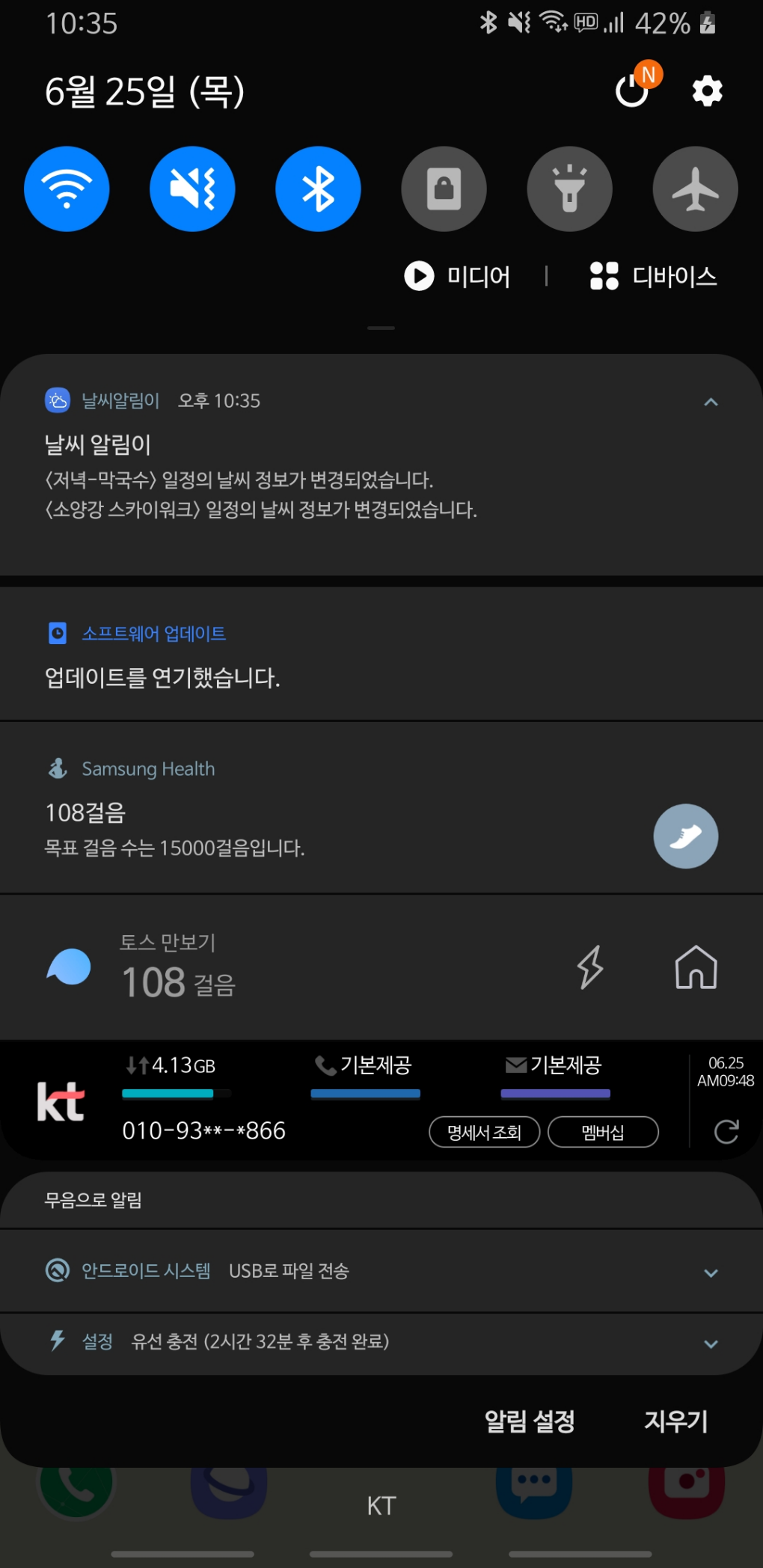
**<ParsingWeather.kt>**

// 등록된 장소의 시차 정보 가져오기  
fun getGMTOffset(json: JSONObject): Long

// 아침/낮/저녁/밤 시간 정보 확인  
fun tempTime(date: String): String

1. 날씨 변경 상태바 알림

JobService를 이용하여 일정 주기마다 전체 날씨를 업데이트하여 변경된 내용이 있다면 상태바 알림이 생성됩니다. 변경된 내용이 없다면 상태바 알림이 생성되지 않습니다.



**<MainActivity.kt>**

// 일정주기마다 JobScheduler 실행

val js = getSystemService(Context.JOB\_SCHEDULER\_SERVICE) as JobScheduler  
 val serviceComponent = ComponentName(this, MyJobService::class.java)  
 val jobInfo = JobInfo.Builder(22, serviceComponent)  
 .setRequiredNetworkType(JobInfo.NETWORK\_TYPE\_UNMETERED)  
 .setPeriodic(TimeUnit.MINUTES.toMillis(15))   
 .build()  
 js.schedule(jobInfo)

**<ParsingWeather.kt>**

// 날씨 정보가 변경되면 변경 내용 추가

if (isTitle.weather != null && isTitle.weather != scheduleEntity.weather) {  
 *notificationContent* += "${isTitle.title} 날씨 정보가 변경되었습니다.\n"  
}

// 모든 날씨 정보가 업데이트되면 알림 생성 / 업데이트 시간 바꾸기

if (*notificationResultCnt* == *ScheduleList*.size) {  
 *makeNotification*()  
 val current = LocalDateTime.now()  
 val formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")  
 val formatted = current.format(formatter)  
 *updatetime*.setText(formatted)  
}

**<Notification.kt>**

// 상태바 알림 생성

fun makeNotification()

// 알림 채널 생성  
fun createNotificationChannel(  
 context: Context, importance: Int, showBadge: Boolean,  
 name: String, description: String  
)

**<MyJobService.kt>**

override fun onStartJob(params: JobParameters): Boolean {  
  
 *ScheduleList*.clear()  
 val appDatabase = AppDatabase  
 *CoroutineScope*(Dispatchers.IO).*launch* **{** var it = appDatabase?.getInstance(*applicationContext*)?.DataDao()?.getData2()  
 it?.*forEach* **{** *ScheduleList*.add(**it**)  
 **}** *notificationContent* = ""  
 *notificationResultCnt* = 0  
 // 모든 날씨 정보를 다시 받아오는 작업  
 for (i in *ScheduleList*) {  
 MainAsyncTask(*applicationContext*).execute(i)  
 }  
 **}** return true  
}

1. DB

개별 사용자는 각자 다른 정보를 필요로 하기 때문에 SQLite에 Room ORM을 적용한로컬 데이터베이스를 사용하였습니다. RoomDatabase는 싱글톤 방식을 이용하여 하나의 인스턴스만을 사용하도록 하였습니다.

DB에는 ScheduleEntity 정보가 저장되어 있습니다.

**<ScheduleEntity.kt>**

@Entity(tableName = "Schedule")  
data class ScheduleEntity(  
 var weather: String?,  
 @PrimaryKey  
 var title: String,  
 var date: String,  
 var timestamp: Long,  
 var temp: Float?,  
 var rain: Float?,  
 var latitude: Double?,  
 var longitude: Double?,  
 var place: String,  
 var isFirstItem: Boolean = false,  
 var isLastItem: Boolean = false  
) : ViewType {  
  
 override fun getViewType(): Int = ViewType.ITEM  
  
}

사용한 쿼리 정보는 다음과 같습니다. DB에 접근하는 함수를 호출할 때는 MainThread가 아닌 다른 쓰레드를 이용해야 합니다. 저는 Coroutine을 이용해 비동기적으로 함수를 호출하여 사용하였습니다.

이 때 반환값을 LiveData로 설정하면 Observe를 통해 데이터의 변경을 감지할 수 있기 때문에 일정 추가 / 수정 시 리스트를 업데이트하는 따로 코드를 작성하지 않고 MainActivity에서 데이터 변경이 감지되면 DB에서 일정 데이터들을 다시 가져와 리스트를 생성했습니다.

**<DataDao.kt>**

@Dao  
abstract class DataDao : BaseDao<ScheduleEntity> {  
  
 // 시간 순서대로 DB에 저장된 데이터 가져오기(LiveData)  
 @Query("SELECT \* FROM Schedule ORDER BY timestamp ")  
 abstract fun getData(): LiveData<List<ScheduleEntity>>  
  
 // 시간 순서대로 전체 DB에 저장된 데이터 가져오기  
 @Query("SELECT \* FROM Schedule ORDER BY timestamp ")  
 abstract fun getData2(): List<ScheduleEntity>  
  
 // title로 ScheduleEntity 찾기 (수정/ 추가모드 위해 데이터 존재 여부 확인)  
 @Query("SELECT \* FROM Schedule WHERE title = :title")  
 abstract fun getScheduleByTitle(title: String): ScheduleEntity  
  
 @Delete  
 abstract fun deleteSchedule(schedule: ScheduleEntity)  
  
  
 @Update(onConflict = OnConflictStrategy.*REPLACE*)  
 abstract fun updateSchedule(schedule: ScheduleEntity)  
  
 @Insert  
 abstract fun insertSchedule(schedule: ScheduleEntity)  
}

**<WeatherItemDelegateAdapter.kt>**

val appDatabase = AppDatabase  
*CoroutineScope*(Dispatchers.IO).*launch* **{** appDatabase?.getInstance(*context*)?.DataDao()  
 ?.deleteSchedule(item)  
**}**

**<MainActivity.kt>**

val appDatabase = AppDatabase  
appDatabase?.getInstance(*applicationContext*)?.DataDao()?.getData()?.observe(this,  
 androidx.lifecycle.*Observer* **{** *ScheduleList*.clear()  
 **it**?.*forEach* **{** *ScheduleList*.add(**it**)  
 **}** resetAdapter()  
 **}**)