Praktikum 6 – Sensoren und Vernetzung in der App

In diesem Praktikum sollen Sensordaten erfasst und mit einem Hintergrundsystem abgeglichen werden. In der Anwendung sollen App-Nutzer Preisinformationen und weitere Daten zu Tankstationen bereitstellen, auf die andere Nutzer dann zugreifen können.

Schritt 1: Projekt anlegen

Für die Entwicklung der Funktionalität wird ein neues Projekt (*EmptyActivity*) angelegt. Die Schritte dazu sind bekannt.

Es wird davon ausgegangen, dass alle Source-Dateien im Paket

```
package de.hsos.ma.andsensorretrofit
```

abgelegt werden.

Schritt 2: Einstellungen vornehmen

In der Gradle-Datei des Projektes sollten folgende Plugins konfiguriert sein:

```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
    id 'kotlin-android-extensions'
}
```

Wir benötigen hier die folgenden zusätzlichen *Dependencies*:

```
dependencies {
    …

    def lifecycle_version = "2.5.1"
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version"
    implementation "androidx.lifecycle:lifecycle-livedata-ktx:$lifecycle_version"

    implementation 'com.squareup.retrofit2:retrofit:2.7.2'
    implementation 'com.squareup.retrofit2:converter-gson:2.7.2'
    implementation 'com.squareup.okhttp3:okhttp:3.14.7'
}
```

Schritt 3: Layout anlegen

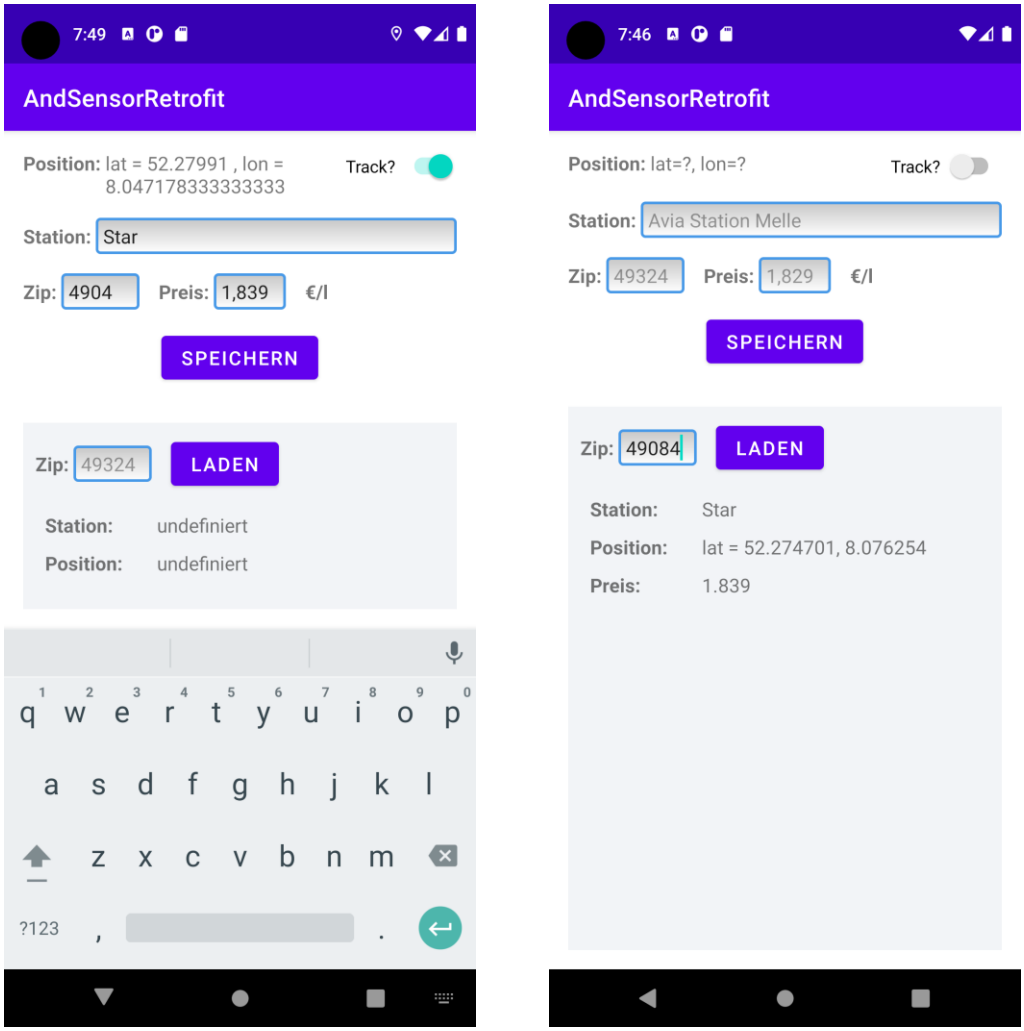
Für das Layout soll für die Eingabe-Elemente ein Style genutzt werden. Legen Sie die folgende Datei (Änderungen sind hier durchaus möglich) unter dem Namen **`edit_text_style.xml`** im Ressourcen-Ordner **`drawable`** ab:

```
<?xml version="1.0" encoding="utf-8" ?>

<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:thickness="0dp"
    android:shape="rectangle">
    <stroke android:width="2dp"
```

```
        android:color="#4799E8"/>
    <corners android:radius="3dp" />
    <gradient
        android:startColor="#C8C8C8"
        android:endColor="#FFFFFF"
        android:type="linear"
        android:angle="270"/>
</shape>
```

Die App sollte folgendes Aussehen haben:



- Im oberen Bereich können (Sensor-) Daten erfasst werden und auf einen Server hochgeladen werden.
- Im unteren Bereich können Daten vom Server heruntergeladen werden.

Da die Erstellung des Layouts doch recht zeitaufwändig ist, wird dieses vorgegeben. Es befindet sich im Upload-Bereich des Lernraums zur Lehrveranstaltung.

Um einfacher auf die View-Elemente (Vermeidung von **`findViewById()`**) im Layout zugreifen zu können, verwenden wir die Kotlin-Extensions (zuvor als Plugin konfiguriert):

```
import kotlinx.android.synthetic.main.activity_main.*
```

Schritt 4: Tracking Funktionalität realisieren

Wir realisieren zunächst die Tracking-Funktion der App. Bei der Erfassung der Positionen von Stationen soll hier auf GPS-Daten zurückgegriffen werden.

In der *MainActivity* vereinbaren wir die folgenden Variablen:

```
private lateinit var locationManager: LocationManager    // Manager
private lateinit var tvGpsLocation: TextView             // Anzeige der Position
private var location: Location? = null                   // Letzte Position
private val locationPermissionCode = 2                   // Request Code für
Permission
```

Diese Variablen werden von den Methoden der *MainActivity* benötigt. Die Nutzung der GPS-Positionierung ist nur möglich, wenn vorher entsprechende Privilegierungen beim Nutzer eingeholt werden wurden, welches die nachfolgende Funktion leistet:

```
override fun onRequestPermissionsResult(
    requestCode: Int,
    permissions: Array<out String>, grantResults: IntArray
): Unit {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults)
    if (requestCode == locationPermissionCode) {
        if (grantResults.isNotEmpty() && grantResults[0] ==
PackageManager.PERMISSION_GRANTED) {
            Toast.makeText(this, "Permission Granted", Toast.LENGTH_SHORT).show()
        } else {
            Toast.makeText(this, "Permission Denied", Toast.LENGTH_SHORT).show()
        }
    }
}
```

Dazu sind im Manifest noch die folgenden Eintragungen vorzunehmen:

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

Beim Einschalten des Trackings werden diese Privilegien abgefragt. Dies erfolgt nebenläufig, was zu einem Absturz führen kann, wenn vor Abschluss des Abfrage-Dialoges auf den *LocationManager* zugegriffen wird. Eine Gegenmaßnahme besteht darin, bei der ersten Abfrage der App kontinuierlichen Zugriff auf den GPS-Sensor zu geben.

```
private fun enableTracking() {
    if ((ContextCompat.checkSelfPermission(
            this,
            Manifest.permission.ACCESS_FINE_LOCATION
        ) != PackageManager.PERMISSION_GRANTED)
    ) {
        ActivityCompat.requestPermissions(
            this,
            arrayOf(Manifest.permission.ACCESS_FINE_LOCATION),
            locationPermissionCode
        )
    }
```

```
    locationManager = getSystemService(Context.LOCATION_SERVICE) as LocationManager
    locationManager.requestLocationUpdates(
        LocationManager.GPS_PROVIDER,
        5000, 5f, this
    )
}
```

Analog zum Einschalten gibt es auch eine Funktion zum Abschalten des Trackings:

```
private fun disableTracking() {
    tvGpsLocation.text = "lat = ?, lon = ?"
    this.location = null
    locationManager.removeUpdates(this);
}
h
override fun onLocationChanged(location: Location) {
    tvGpsLocation = findViewById(R.id.positionLabel)
    this.location = location
    tvGpsLocation.text =
        "lat = " + this.location?.latitude + " , lon = " + this.location?.longitude
}
```

Die Reaktion auf Positionsänderungen erfolgt in der Funktion **`onLocationChanged()`**. Hier wird die aktuelle Position in einer Variablen gespeichert und in einen *TextView* geschrieben.

Nach dieser Vorbereitung, kann in der Methode **`onCreate()`** der Aktivität das Tracking eingeschaltet werden. Hierzu dient der Switch-Button im Layout.

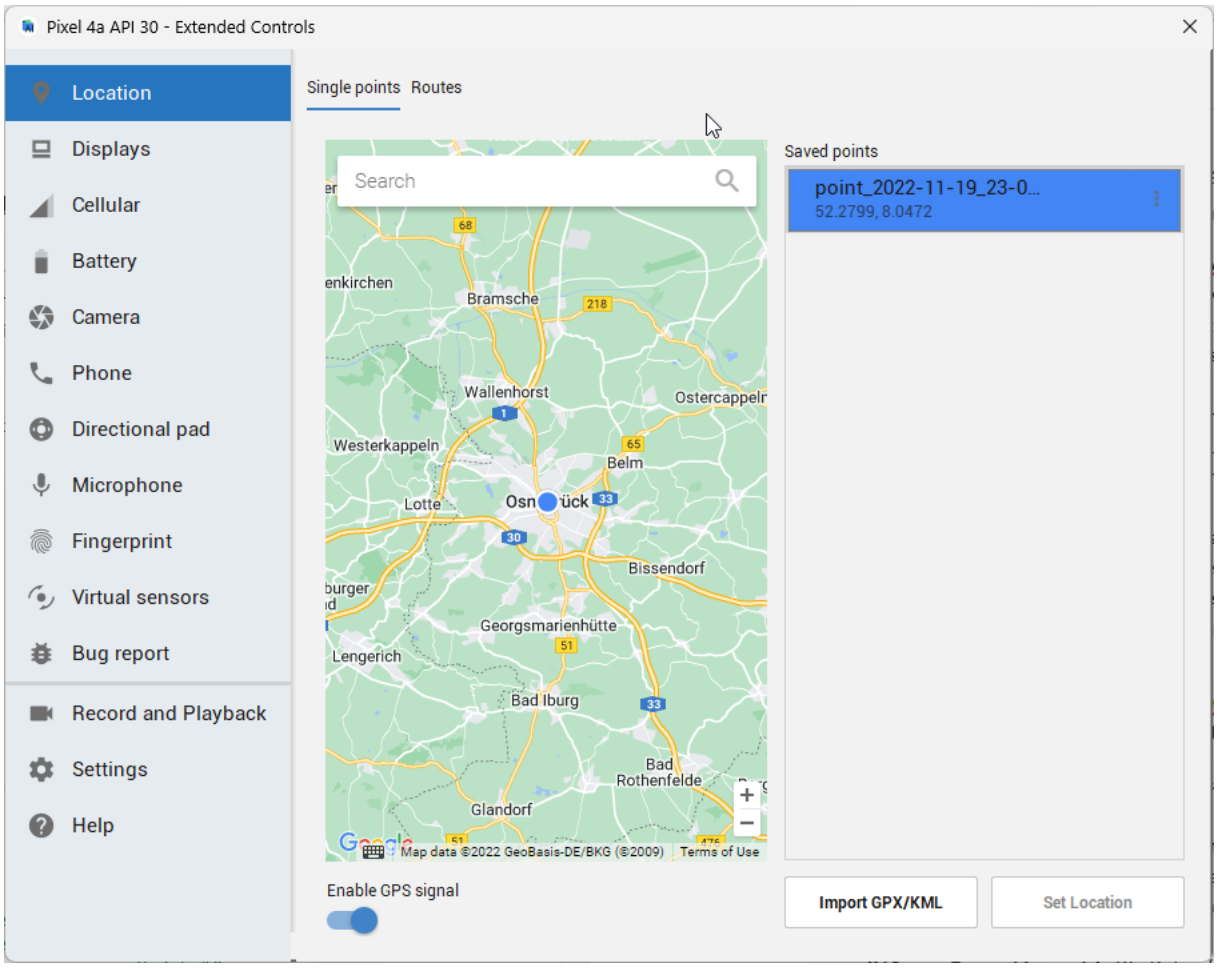
```
// Behandlung des Trackings
val trackingOnOffSwitch = findViewById<View>(R.id.trackingSwitch) as Switch
trackingOnOffSwitch.setOnCheckedChangeListener { buttonView, isChecked ->
    Log.v(TAG, "Track? " + isChecked)
    if (isChecked)
        enableTracking()
    else {
        disableTracking()
    }
}
```

Schritt 5: Test der Tracking-Funktion

Die Teilfunktion der App kann nun schon getestet werden. Beim Einschalten des Switch-Buttons sollten im View die GPS-Koordinaten erscheinen. Das funktioniert auch im Emulator. Nützlich sind dabei die *Extended Controls*.

Z.B. ist es möglich, einen Ort zu suchen, sodann kann dieser Ort per *Set Position* gespeichert werden. Einen gespeicherten Ort kann man auswählen und per *Set Location* einen Positionswechsel emulieren. Bei eingeschaltetem Tracking erscheint nun die Position im View in Form von WGS-Koordinaten für Längen- und Breitengrad.

Über diesen Mechanismus können auch Routen emuliert werden, was hier aber nicht notwendig ist.



Schritt 6: Datenaustausch mit einem Backend

Für das Praktikum wird ein einfaches Backend mit einem REST-Interface bereitgestellt. Die Kommunikation erfolgt selbstredend über http(s). Die Funktion erschließt sich recht einfach, sofern man einmal einfache Routen aufruft. Beispiele:

http://ilexanu.site:8080/RESTStationProvider-1.0-SNAPSHOT/ProviderServlet/49084

liefert die folgende JSON-formatierte Nachricht (Liste von Tankstationen mit der Postleitzahl 49084):

```
[{"zip":49084,"cost":1.839,"latitude":"52.274701","name":"Star","longitude":"8.076254"},{"zip":49084,"cost":1.839,"latitude":"52.263978","name":"Westfalen","longitude":"8.065432"}]
```

Für den Datenaustausch nutzen wird die weit verbreitete Bibliothek Retrofit. Nehmen wir an, die Daten zu den Stationen werden in einer Klasse **`StationData`** (sog. *POJO*, hier als Java-Klasse, die vom Backend übernommen wurde) verwaltet:

```
package de.hsos.ma.andsensorretrofit;

import com.google.gson.annotations.SerializedName;

public class StationData {
    @SerializedName("name")
    private final String name;
```

```
    private final int zip;
    private final String latitude;
    private final String longitude;
    private final double cost;

    public StationData(String name, int zip, String latitude, String longitude,
double cost) {
        this.name = name;
        this.zip = zip;
        this.latitude = latitude;
        this.longitude = longitude;
        this.cost = cost;
    }
    public String getName() {
        return name;
    }
    public int getZip() {
        return zip;
    }
    public double getCost() {
        return cost;
    }
    public String getLatitude() {
        return latitude;
    }
    public String getLongitude() {
        return longitude;
    }
    public String toString() {
        String s = new String();
        s = s.format ("%s, %d, %s", name, zip, cost);
        return s;
    }
}
```

Wir benötigten ein API, um auf die Daten des Backends via http-Methoden zugreifen zu können. Genau das wird in der folgenden Klasse **`StationAPI`** beschrieben:

```
package de.hsos.ma.andsensorretrofit

import retrofit2.Call
import retrofit2.http.Body
import retrofit2.http.GET
import retrofit2.http.POST
import retrofit2.http.Url

interface StationAPI {

    @GET
    suspend fun getStationsForZip(@Url url: String): List<StationData>

    @GET
    suspend fun getStationForZip(@Url url: String): StationData

    @POST("stationData")
    suspend fun addStation(@Body stationData: StationData): StationData
}
```

Der lesende Zugriff per *http-GET* erfolgt in unserer Anwendung über die Angabe einer Postleitzahl (PLZ / zip), die in Form einer URL in der Anfrage an den Server geschickt wird. Wir können alle zu einer PLZ hinterlegten Stationen ausgeben (Bsp. für eine Route: http://..../49034) oder auf einen bestimmten, unter einem bestimmten Index (Bsp.: http://..../49034/1) in der Liste gespeicherten Eintrag. Das Hinzufügen von Eintragungen erfolgt über *http-POST* in der API-Methode **`addStation()`**. Zu beachten ist: Zugriffe auf das Netzwerk dürfen nicht im UI-Thead stattfinden, weshalb wir eigene Threads benötigen. Die entsprechende Dependency für die nutzbaren Bausteine hatten wir schon zu Anfang konfiguriert.

Mit diesem API instanziieren wir Service-Objekt (Singleton), welches den Service modelliert.

```
package de.hsos.ma.andsensorretrofit

import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory

object StationService  {
    val retrofit by lazy {
        Retrofit.Builder()

.baseUrl("http://ilexanu.site:8080/RESTStationProvider/ProviderServlet/")
            .addConverterFactory(GsonConverterFactory.create())
            .build()
            .create(StationAPI::class.java)
    }
}
```

Das Konzept ‚*Shared ViewModel*' hatten wir schon kennengelernt. Da die Zugriffe auf das Backend asynchron erfolgen müssen, um eine Blockierung des App-UI zu verhindern, können wir ein solches Modell zur Speicherung von asynchron eintreffenden Daten des Servers nutzen. *Observer* können dann auf beobachtete Änderungen reagieren und Views aktualisieren.

```
package de.hsos.ma.andsensorretrofit

import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope

import kotlinx.coroutines.launch
import retrofit2.Call
import retrofit2.Callback
import retrofit2.Response

class StationViewModel : ViewModel() {

    val myResponse: MutableLiveData<StationData> = MutableLiveData()
    val myResponseList: MutableLiveData<List<StationData>> = MutableLiveData()


    fun getStationsForZip(zip: Int) {
        viewModelScope.launch {
            myResponseList.value =
StationService.retrofit.getStationsForZip(zip.toString())
        }
    }

    fun getStationForZip(zip: String) {
        viewModelScope.launch {
```

```
            myResponse.value = StationService.retrofit.getStationForZip(zip)
        }
    }

    fun addStation(stationData: StationData) {
        viewModelScope.launch {
            myResponse.value = StationService.retrofit.addStation(stationData)
        }
    }
}
```

Nun sind die Vorbereitungen abgeschlossen. Die eigentlichen Zugriff auf den Server werden in **`onCreate()`** implementiert.

Zunächst ist das *ViewModel* zu erzeugen:

```
val viewModel = ViewModelProvider(this).get(StationViewModel::class.java)
```

Dann erfolgt die Realisierung der Funktionalität zum Abruf von Daten:

```
val buttonLoad: Button = findViewById(R.id.loadStationData)
buttonLoad.setOnClickListener {
    if (zipEditLoad.text.toString().toIntOrNull() == null) {
        Toast.makeText(this, "Bitte Zip-Feld korrigieren.", Toast.LENGTH_SHORT).show()
    } else {
        val zip = zipEditLoad.text.toString().toInt()
        // einen Eintrag auslesen
        viewModel.getStationForZip(zip.toString() + "/1") // getStation()
        viewModel.myResponse.observe(this, Observer {
            if (it != null) {
                // TextViews setzen
                stationLabelLoadVal.setText(it?.name ?: "empty")
                val position_str: String = "lat = " + (it?.latitude ?: "?") +
                        ", lon = " + (it?.longitude ?: "?")
                positionLabelLoadVal.setText(position_str)
                costLabelLoadVal.setText(it?.cost.toString() ?: "0.0")
                Log.d(TAG, it?.toString() ?: "empty")
            }
        })
    }
}
```

Wesentlich komplexer ist das Speichern von Daten zu Stationen, da hier insbesondere auch fehlerhafte Formulardaten (diese sind dabei noch nicht einmal vollständig) ausgeschlossen werden müssen.

```
val buttonLoad: Button = findViewById(R.id.loadStationData)
buttonLoad.setOnClickListener {
    if (zipEditLoad.text.toString().toIntOrNull() == null) {
        Toast.makeText(this, "Bitte Zip-Feld korrigieren.", Toast.LENGTH_SHORT).show()
    } else {
        val zip = zipEditLoad.text.toString().toInt()
        // einen Eintrag auslesen
        viewModel.getStationForZip(zip.toString() + "/1") // getStation()
        viewModel.myResponse.observe(this, Observer {
            if (it != null) {
                // TextViews setzen
                stationLabelLoadVal.setText(it?.name ?: "empty")
                val position_str: String = "lat = " + (it?.latitude ?: "?") +
```

```
                    ", lon = " + (it?.longitude ?: "?")
                positionLabelLoadVal.setText(position_str)
                costLabelLoadVal.setText(it?.cost.toString() ?: "0.0")
                Log.d(TAG, it?.toString() ?: "empty")
            }
        })
    }
}
```

Schritt 7: Test der Kommunikation mit dem Backend

Die Gesamtapplikation kann nun getestet werden.

Die hier gezeigte Implementierung ist dabei nicht vollständig. Ist das Backend nicht erreichbar, so kann es beispielsweise zu einem Absturz der App kommen.

Optionale Aufgaben

- Im Rahmen der Unterlagen wurde die Erstellung einer Stand-alone App beschrieben. Diese kann in die *DriverHelp* App als Activity integriert werden.

- Das API und Backend sieht auch Funktionen zum Laden von Listen von Stationen vor, siehe Funktion **`getStationsForZip()`**. Diese könnten in einem *ListView* angezeigt werden. Über einen Adapter können beim Anklicken die ausführlichen Daten angezeigt werden.

- Um absehbare Abstürze der App bei fehlenden Netzwerkverbindungen oder Probleme im Backend bei der Verarbeitung von Daten zu verhindern, könnte die Netzwerkverbindung geprüft werden. Siehe dazu die Funktion **`retrieveURL()`** aus Vorlesung. Auch hier ist zu beachten: man darf die Abfrage nicht auf dem Main-/UI-Thread ausführen.

Informationen und Hilfen

- Android LocationManager : https://developer.android.com/reference/android/location/LocationManager
- Retrofit: https://square.github.io/retrofit/
- Retrofit dynamische URLs : https://square.github.io/retrofit/
- Retrofit unter Kotlin (grundlegendes Tutorial): https://dev.to/theimpulson/making-get-requests-with-retrofit2-on-android-using-kotlin-4e4c