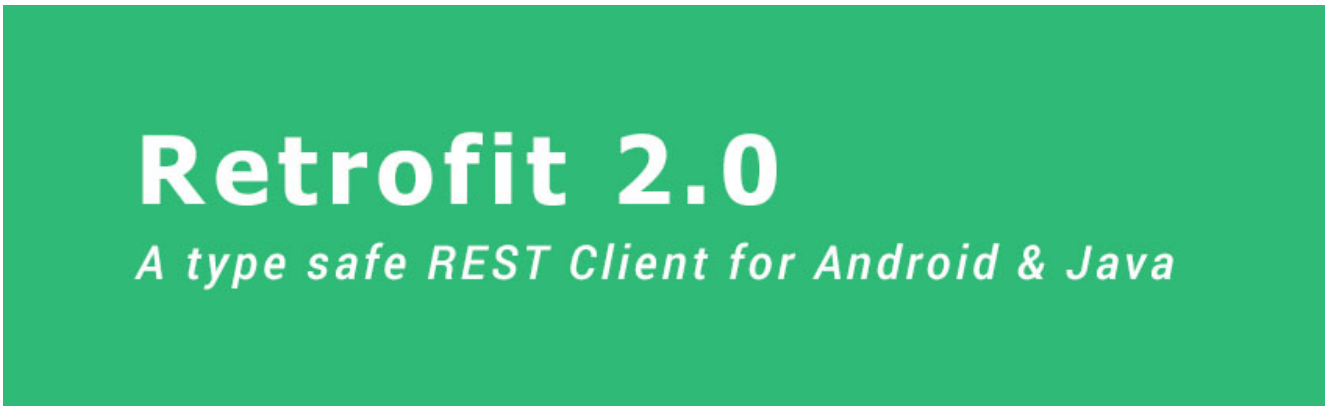
Android – Retrofit y Seeeduino Cloud



android – retrofit

Introducción

Y con este post llegamos a una de la librerías más útiles. Retrofit nos facilita el uso de llamadas rest, realizando las llamadas asíncronas sin que nosotros nos preocupemos de nada. Si además usamos GSON podemos obtener el resultado en una colección de objetos. La utilización de Retrofit en nuestros proyectos es muy fácil y ofrece un buen rendimiento. Una vez más, Square nos brinda una gran librería (ya hemos hablado anteriormente de Picasso).

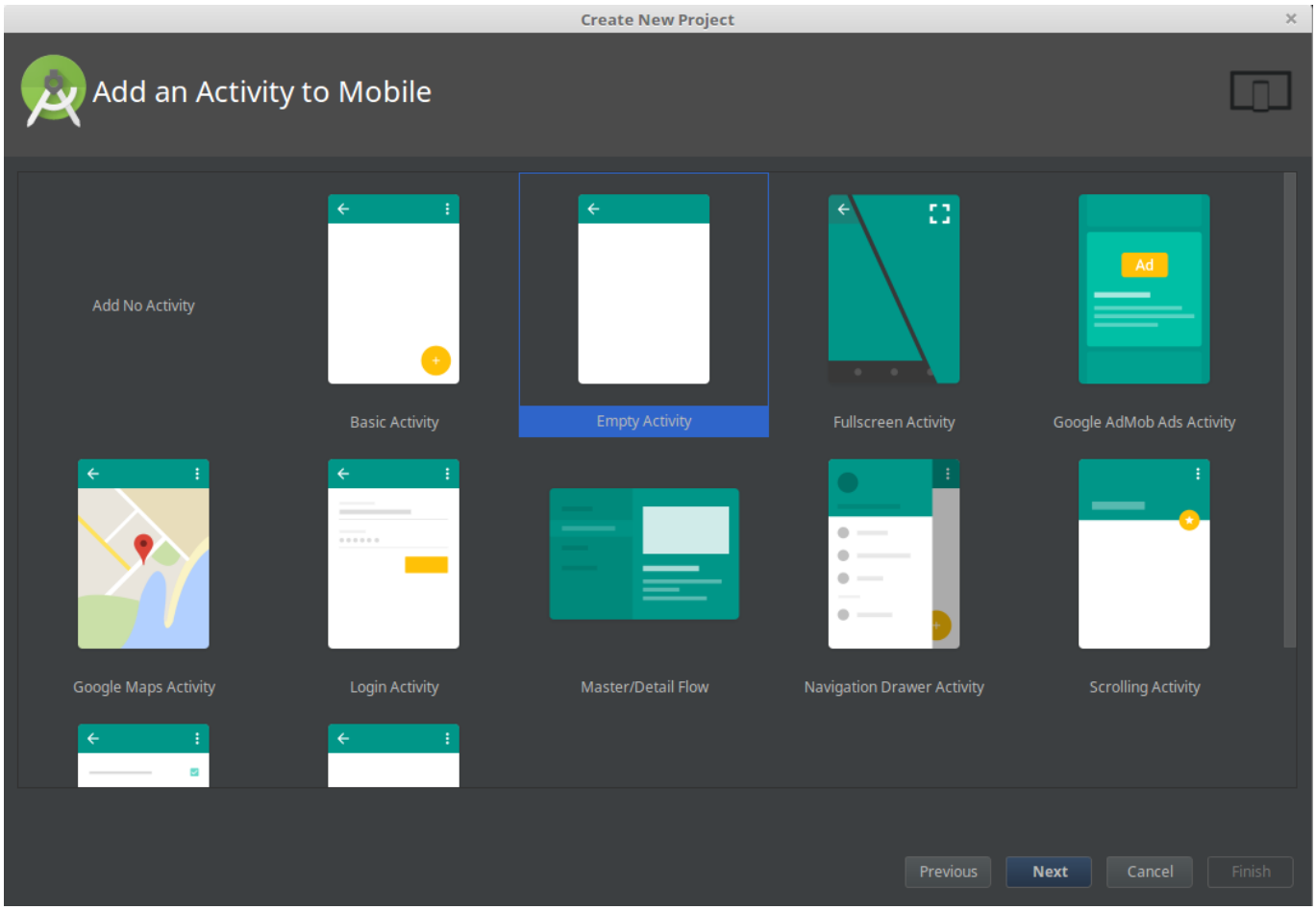
Este post es una continuación de los de seeeduino (Seeduino Cloud – Parte 1 , Seeeduino Cloud – Parte 2). Al finalizarlo podremos controlar nuestro ventilador des de una aplicación

Android.

- Creando la estructura del proyecto
- Configuración
- Creando nuestras clases Pojo
- Creando una instancia de Retrofit
- Preparando end points
- Preparando nuestro activity

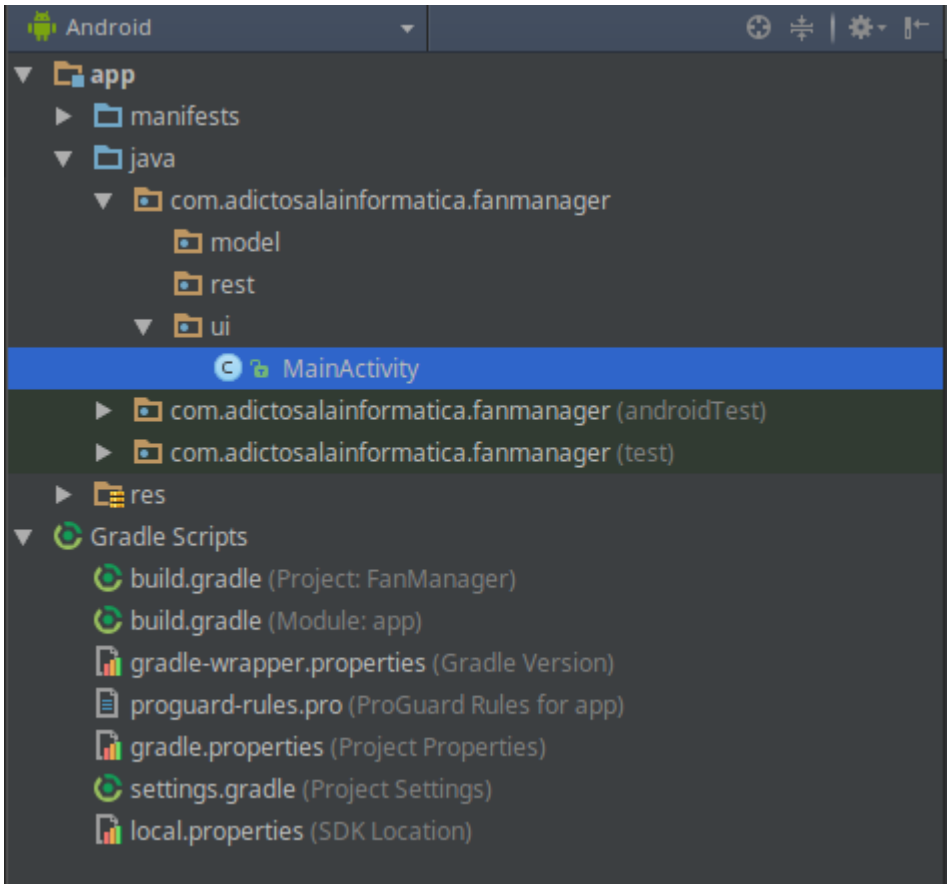
Creando el proyecto y su estructura

Crearemos un proyecto con un empty activity



A continuación crearemos una nueva estructura de packages. Esto es una costumbre, no es ningún estándar y cada developer utiliza la que más le conviene o resulta más fácil de usar, para organizar su código. Muy probablemente, cambiará durante el proceso desarrollo y aquí cada unos tiene sus

preferencias. Lo que realmente es importante es utilizar una que nos ayude mantener nuestro código debidamente organizado.



Configuración

Añadiremos el plugin android-apt a nuestro classpath en el fichero **build.gradle**. Este se encuentra en la raíz de nuestro proyecto.

```
dependencies{
  classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'
}
```

Dentro del archivo **app/build.gradle**, debemos añadir las dependencias de Retrofit.

```
apply plugin : 'com.neenbedankt.android-apt'

dependencies {

  // retrofit, gson
  compile 'com.google.code.gson:gson:2.6.2'
```

```
  compile 'com.squareup.retrofit2:retrofit:2.0.2'
  compile 'com.squareup.retrofit2:converter-gson:2.0.2'

  // picasso
  compile 'com.squareup.picasso:picasso:2.5.2'
  compile 'jp.wasabeef:picasso-transformations:2.1.0'

  // butterknife
  compile 'com.jakewharton:butterknife:8.4.0'
  apt 'com.jakewharton:butterknife-compiler:8.4.0'
}
```

Seguidamente añadiremos el permiso de red y de comprobación de esta a nuestro archivo **androidmanifest.xml**

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

Creando nuestra clase Pojo

Esta es la clase base sobre la que Gson creará la instancia con los resultados de la llamada rest realizada con Retrofit, colocaremos la clase en el package **model**. Una de las maneras más fáciles de generar nuestras clases es utilizar un generador. Por ejemplo, jsonschema2pojo. Pero debemos tener cuidado, podemos encontrarnos fácilmente con una respuesta Json inmensa de la cual tan solo necesitamos alguna información concreta.

```
package com.adictosalainformatica.fanmanager.model;

import com.google.gson.annotations.Expose;
import com.google.gson.annotations.SerializedName;

public class FanModel {

    @SerializedName("pin")
    @Expose
```

```
private Integer pin;
@SerializedName("status")
@Expose
private Integer status;

/**
 *
 * @return
 * The pin
 */
public Integer getPin() {
    return pin;
}

/**
 *
 * @param pin
 * The pin
 */
public void setPin(Integer pin) {
    this.pin = pin;
}

/**
 *
 * @return
 * The status
 */
public Integer getStatus() {
    return status;
}

/**
 *
 * @param status
 * The status
 */
public void setStatus(Integer status) {
    this.status = status;
}
```

```
}
```

Es importante no acabar con una colección de clases inmensa llena de getters y setters que no vamos a utilizar. Todo ello para mantener nuestro código limpio o no acabar encontrándonos el error «64k method limit in dex». No es raro encontrarnos con él si utilizamos muchas librerías y colecciones de clases generadas a partir del resultado de una respuesta Json desmesurada. Podemos fácilmente solventar el problema revisando nuestras clases Pojo y eliminando todas aquellas variables y clases que no vamos a utilizar e incluso plantearnos si todas las librerías que estamos utilizando son realmente necesarias. Pero si esto no fuera suficiente, podemos solventar el problema con multidex a costa de perder la funcionalidad de instant run.

Creando una instancia de Retrofit

Para enviar solicitudes de red a una API, tenemos que utilizar la clase Retrofit.Builder y especificar la URL base para el servicio. Por lo tanto, crearemos una clase llamada ApiClient.java bajo en el package **rest**.

BASE_URL – es la url base de nuestra API. Vamos a utilizar esta url para todas las solicitudes posteriores.

```
package com.adictosalainformatica.fanmanager.rest;

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


public class ApiClient {

        public static final String BASE_URL =
"http://192.168.1.33/arduino/";
    private static Retrofit retrofit = null;
```

```
    public static Retrofit getClient() {
        if (retrofit==null) {
            retrofit = new Retrofit.Builder()
                    .baseUrl(BASE_URL)
.addConverterFactory(GsonConverterFactory.create())
                    .build();
        }
        return retrofit;
    }
}
```

Preparando end points

Los end points se definen dentro de una interfaz mediante anotaciones especiales de Retrofit para codificar información sobre los parámetros y el tipo de petición. Además, el valor de retorno es siempre una llamada con parámetros <T>, en nuestro caso <FanModel>. Crearemos la interface ApiInterface.java en el package **rest**

```
package com.adictosalainformatica.fanmanager.rest;

import
com.emotionexperience.fragancemanager.model.FraganceModel;

import retrofit2.Call;
import retrofit2.http.GET;
import retrofit2.http.Path;


public interface ApiInterface {
    @GET("digital/{pin}/{value}")
     Call setPin(@Path("pin") int pin, @Path("value") int
value);

    @GET("status/{pin}")
    Call getStatus(@Path("pin") int pin);
}
```

Preparando nuestro activity

A continuación mostramos el código de nuestro activity que se encuentra en el package **ui**. Como se puede observar utilizamos Picasso y Butterknife

```
package com.adictosalainformatica.fanmanager.ui;

import android.app.ProgressDialog;
import android.content.Context;
import android.graphics.Color;
import android.net.ConnectivityManager;
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.util.Log;
import android.widget.ImageButton;
import android.widget.Toast;

import com.adictosalainformatica.fanmanager.R;
import com.adictosalainformatica.fanmanager.model.FanModel;
import com.adictosalainformatica.fanmanager.rest.ApiClient;
import com.adictosalainformatica.fanmanager.rest.ApiInterface;
import com.squareup.picasso.Picasso;

import butterknife.BindView;
import butterknife.ButterKnife;
import butterknife.OnClick;
import
jp.wasabeef.picasso.transformations.ColorFilterTransformation;
import
jp.wasabeef.picasso.transformations.CropCircleTransformation;
import retrofit2.Call;
import retrofit2.Callback;
import retrofit2.Response;

public class MainActivity extends AppCompatActivity {

    @BindView(R.id.main_btn_fan)
    ImageButton btnFan;
```

```
    // Constants
            private static final String TAG =
MainActivity.class.getName();
    private static int PIN = 8;

    private int fanStatus = 0;
    private static ApiInterface apiService;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        ButterKnife.bind(this);

        Picasso
                .with(getApplicationContext())
                .load(R.drawable.fan_image)
                .transform(new CropCircleTransformation())

                .into(btnFan);

                                        apiService     =
ApiClient.getClient().create(ApiInterface.class);

        if(isConetctionEnabled(getApplicationContext())){
            getFanStatus(PIN);
        }else{
             Toast.makeText(getApplicationContext(),"Internet
connection failed",Toast.LENGTH_LONG).show();
        }
    }

    @OnClick(R.id.main_btn_fan)
    public void switchFanStatus() {
        if(isConetctionEnabled(getApplicationContext())){
            if(fanStatus == 0){
                setFanStatus(PIN,1);
            }else{
                setFanStatus(PIN, 0);
            }
```

```
        }else{
             Toast.makeText(getApplicationContext(),"Internet
connection failed",Toast.LENGTH_LONG).show();
        }
    }

    /**
     * Get current Fan status
     * @param pin
     */
    private void getFanStatus(int pin) {
        Call <FanModel> call = apiService.getStatus(pin);
        call.enqueue(new Callback<FanModel>() {
            @Override
              public void onResponse(Call<FanModel> call,
Response<FanModel> response) {
                if (response.isSuccessful()) {
                    fanStatus =  response.body().getStatus();
                    Log.d(TAG, "Fan status: " + fanStatus);

                    if(fanStatus == 0){
                                              int  color  =
Color.parseColor("#33ee092b");
                        setColorFilter(color);

                    }else{
                                              int  color  =
Color.parseColor("#3300ff80");
                        setColorFilter(color);
                    }
                } else {
                    //request not successful (like 400,401,403
etc)
                    Log.e(TAG,response.message());
                }
            }

            @Override
              public void onFailure(Call<FanModel> call,
Throwable t) {
                // Log error here since request failed
```

```
                Log.e(TAG, "Error: " + t.toString());
            }
        });
    }

    /**
     * Set Fan status
     * @param pin
     */
    private void setFanStatus(int pin, int status){
        Call call = apiService.setPin(pin,status);
        call.enqueue(new Callback<FanModel>() {
            @Override
              public void onResponse(Call<FanModel> call,
Response<FanModel> response) {
                if (response.isSuccessful()) {
                    fanStatus =  response.body().getStatus();
                    Log.d(TAG, "Fan status: " + fanStatus);

                    if(fanStatus == 0){
                                              int  color  =
Color.parseColor("#33ee092b");
                        setColorFilter(color);

                    }else{
                                              int  color  =
Color.parseColor("#3300ff80");
                        setColorFilter(color);
                    }
                } else {
                    //request not successful (like 400,401,403
etc);
                    Log.e(TAG,response.message());
                }
            }

            @Override
               public void onFailure(Call<FanModel> call,
Throwable t) {
                // Log error here since request failed
                Log.e(TAG, "Error: " + t.toString());
```

```
            }
        });
    }

    /**
     * Set image filter color
     * @param color
     */
    private void setColorFilter(int color){
        Picasso
                .with(getApplicationContext())
                .load(R.drawable.fan_image)
                                            .transform(new
ColorFilterTransformation(color))
                .transform(new CropCircleTransformation())
                .into(btnFan);
    }

    /**
     * Tests if there's connection
     * @param cx context application
     * @return true or false
     */
    public static boolean isConetctionEnabled(Context cx){
                        ConnectivityManager   conMgr   =
(ConnectivityManager)cx.getSystemService(Context.CONNECTIVITY_
SERVICE);

        if (conMgr.getActiveNetworkInfo() != null
                && conMgr.getActiveNetworkInfo().isAvailable()
                                                        &&
conMgr.getActiveNetworkInfo().isConnected()) {
            return true;
        } else {
            return false;
        }
    }
}
```

I finalmente, nuestro layout

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

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
tools:context="com.adictosalainformatica.fanmanager.ui.MainAct
ivity">


    <ImageButton
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:srcCompat="@drawable/fan_image"
        android:layout_centerVertical="true"
        android:layout_centerHorizontal="true"
        android:id="@+id/main_btn_fan"
        android:background="@null"
        android:padding="10dp"/>
</RelativeLayout>
```

Desde este enlace os podéis descargar fan_image

Observaciones

Después de mucho tiempo lidiando con AsyncTask, rotaciones, memory leaks… Retrofit nos permite abstraernos de todo esto y además es muy fácil de utilizar. Por todo ello, Retrofit se convierte en una librería casi indispensable. Finalmente, dejo el enlace a la web de Retrofit y un ejemplo en Github. El cual, nos permite apagar y encender un ventilador siempre y cuando tengamos nuestro Seeduino Cloud configurado y preparado para trabajar con un relayshield.

- Retrofit
- Seeeduino Cloud – Parte 1 (configuración Seeduino Cloud)

- Seeeduino Cloud – Parte 2 (Montaje relayshield)
- FanManager