**Juan Camilo Restrepo Velez 000373886**

Tabla de contenido

[1) Ajuste MariaDB 2](#_Toc40334624)

[2) Ajuste API REST 3](#_Toc40334625)

[2.1) updateData.php 3](#_Toc40334626)

[2.2) postData.php 6](#_Toc40334627)

[3) ConexionApiRest 8](#_Toc40334628)

[3.1) Constructor 8](#_Toc40334629)

[3.2) getData 8](#_Toc40334630)

[3.3) setData 9](#_Toc40334631)

[3.4) updateData 9](#_Toc40334632)

[3.5) downloadData 10](#_Toc40334633)

[3.6) isConnected 10](#_Toc40334634)

[3.7) setHttps 11](#_Toc40334635)

[3.8) tryConnect 11](#_Toc40334636)

[4) EU Estudiante 12](#_Toc40334637)

[4.1) MainActivity 12](#_Toc40334638)

[4.1.1) Iniciar el estado de la actividad 14](#_Toc40334639)

[4.2) ActividadActivity 16](#_Toc40334640)

[4.2.1) Sincronizar SQLite local con nodo MariaDB 19](#_Toc40334641)

[4.2.2) Actualizar el estado de la actividad 20](#_Toc40334642)

[4.3) Modelo de sincronización 26](#_Toc40334643)

[5) EU Profesores Prototipo 28](#_Toc40334644)

[5.1) MainActivity 28](#_Toc40334645)

[5.2) SesionActivity 29](#_Toc40334646)

[5.3) NewSesionActivity 32](#_Toc40334647)

[5.4) ActividadActivity 33](#_Toc40334648)

[Referencias 36](#_Toc40334649)

# 1) Ajuste MariaDB

Se debe adicionar en la base de datos EduApps de MariaDB las siguientes tablas:

USE EduApps;

CREATE TABLE Estados

    (ID INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

     Descripcion VARCHAR(255) NOT NULL);

CREATE TABLE Estudiantes\_Actividades

    (Estudiante\_ID INT NOT NULL,

     Actividad\_ID INT NOT NULL,

     Estado\_ID INT NOT NULL,

     Observaciones VARCHAR(255) NOT NULL,

     PRIMARY KEY (Actividad\_ID, Estudiante\_ID),

     CONSTRAINT FKEstudiante\_Actividad

     FOREIGN KEY (Actividad\_ID) REFERENCES Actividades(ID)

      ON DELETE CASCADE

      ON UPDATE RESTRICT,

     CONSTRAINT FKActividad\_Estudiante

     FOREIGN KEY (Estudiante\_ID) REFERENCES Estudiante(ID)

      ON DELETE CASCADE

      ON UPDATE RESTRICT,

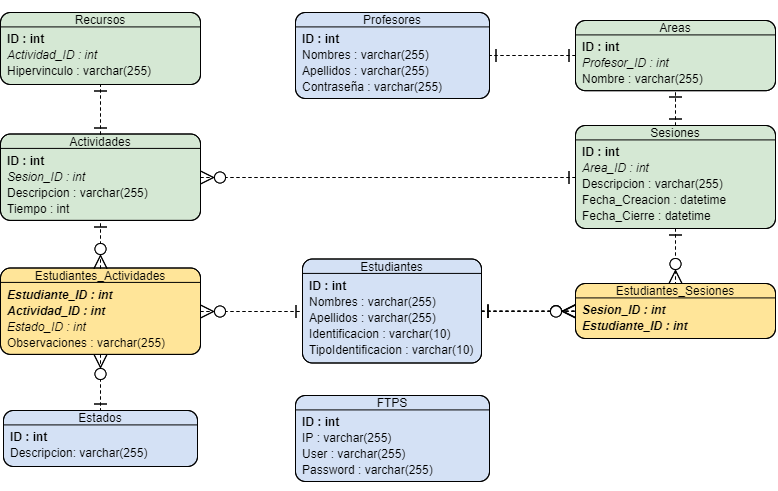
     CONSTRAINT FKActividad\_Estado

     FOREIGN KEY (Estado\_ID) REFERENCES Estados(ID)

      ON DELETE CASCADE

      ON UPDATE RESTRICT);

Dejando el siguiente modelo relacional en el nodo:



Por otro lado, se deben realizar la siguiente inserción:

INSERT INTO Estados VALUES (0,‘Iniciada’),(0,‘Finalizada’),(0,‘Abandonada’);

# 2) Ajuste API REST

Como el API REST que se decidió implementar no está completo (faltan las funciones de update y delete) se nos presenta el impedimento para poder actualizar el estado de la actividad por lo cual debemos implementar al menos la funcionalidad de actualizar los datos, además de mejorar la función de insertar datos ya que no nos devuelve el registro que se acaba de insertar (se requiere para notificar con cuál ID quedó dicho registro).

## 2.1) updateData.php

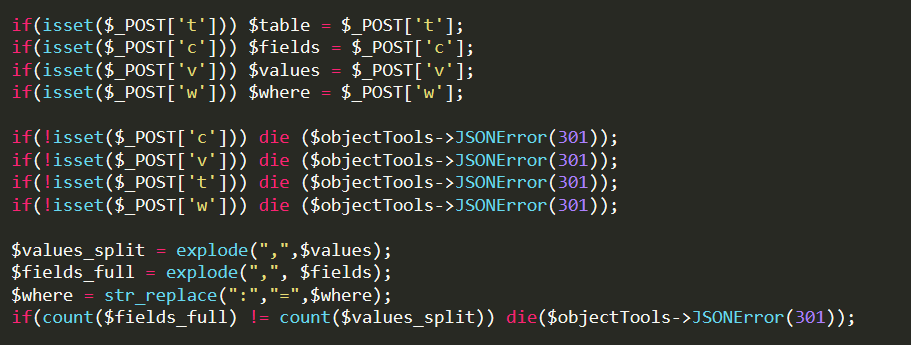
Ahora bien, para poder actualizar datos por medio del API REST se escribe el código necesario en el archivo updateData.php para implementar dicha función por medio de la URL que se deben enviar los siguientes parámetros mediante el método POST:

* Tabla
* Columna
* Valor
* Condición

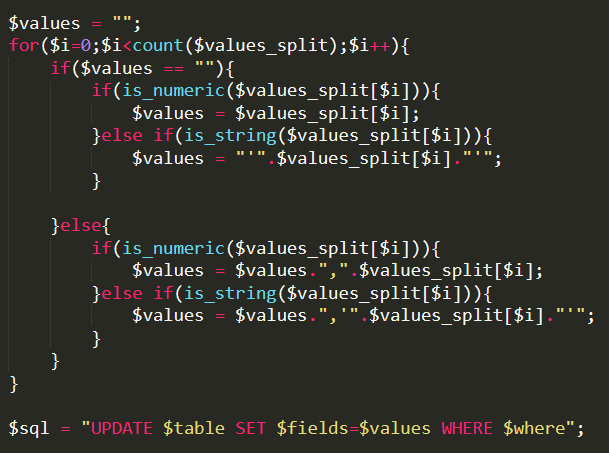
Entonces la URL queda (recordar que da tanto con HTTP como HTTPS por la instalación del certificado del sprint 3):

*https://192.168.0.101/ApiRest/updateData.php?t=table&c=column&v=newValue&w=condicion*

Para el ***código en php*** se reutilizan todas las validaciones que se hacen los demás archivos php por lo que solo se mostrará lo esencial como la obtención de los parámetros:



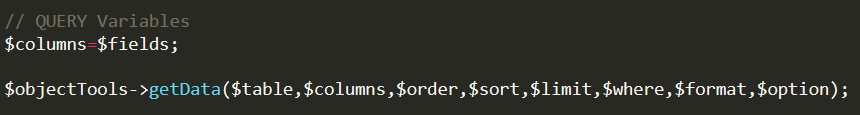
Luego se realizan las validaciones posteriormente mencionadas, para luego realizar la construcción de la instrucción update



Luego se utiliza una clase Tools que se encarga de la conexión con el servidor MariaDB y aplica la sentencia update:

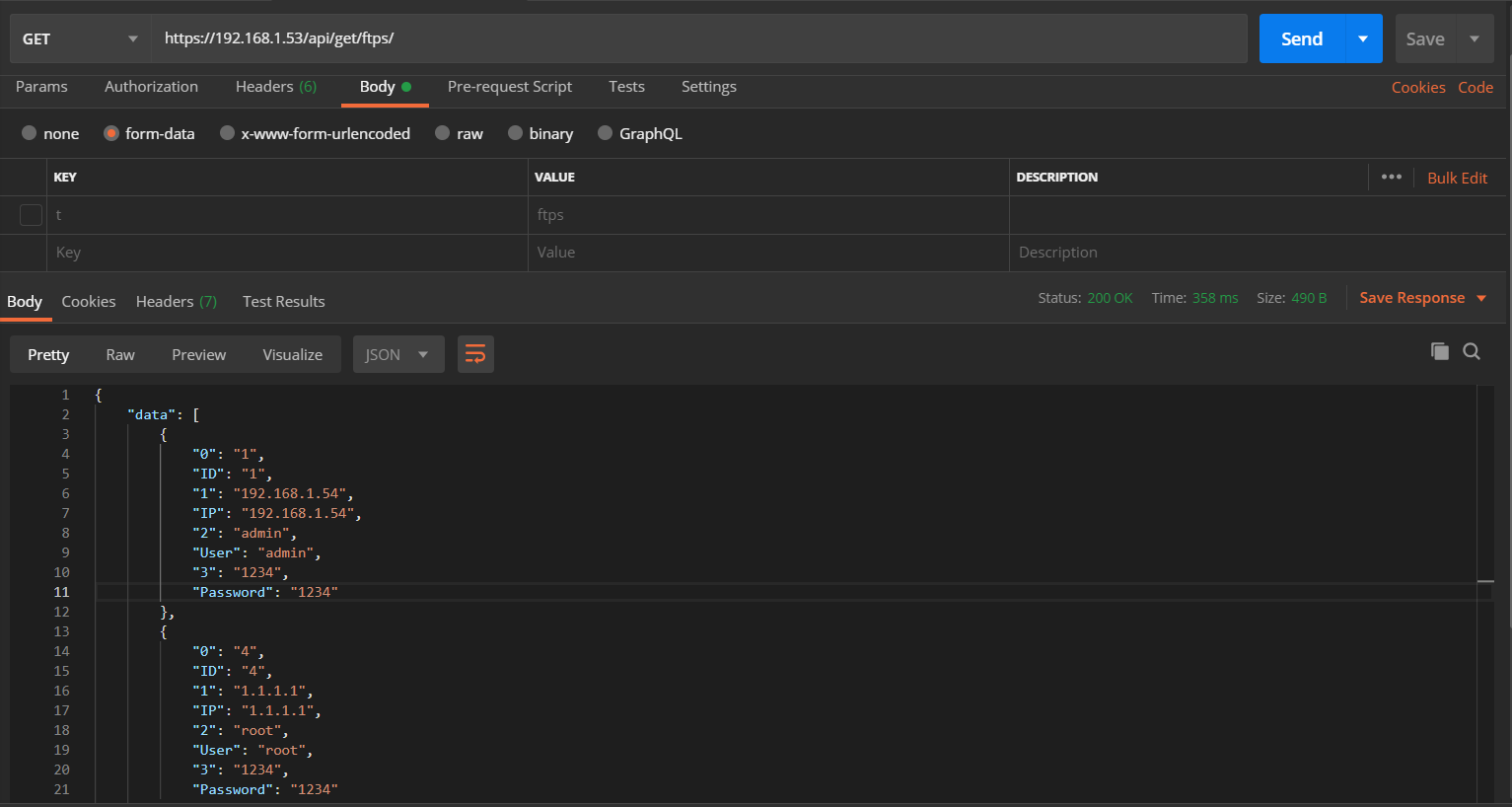


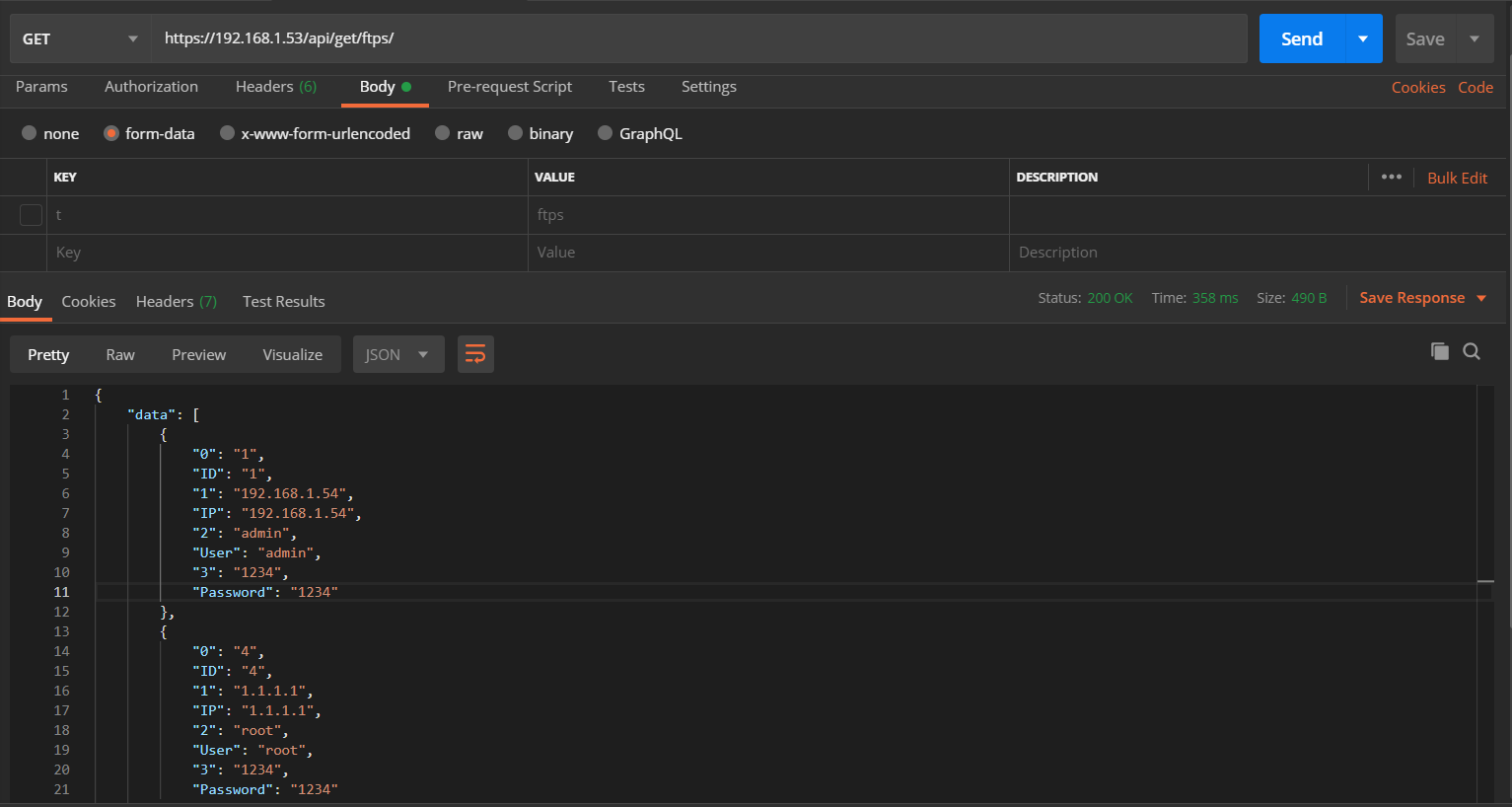
Luego se procede a llamar nuevamente la clase, pero esta vez devolver un JSON con el ID y la columna modificada de la tabla:



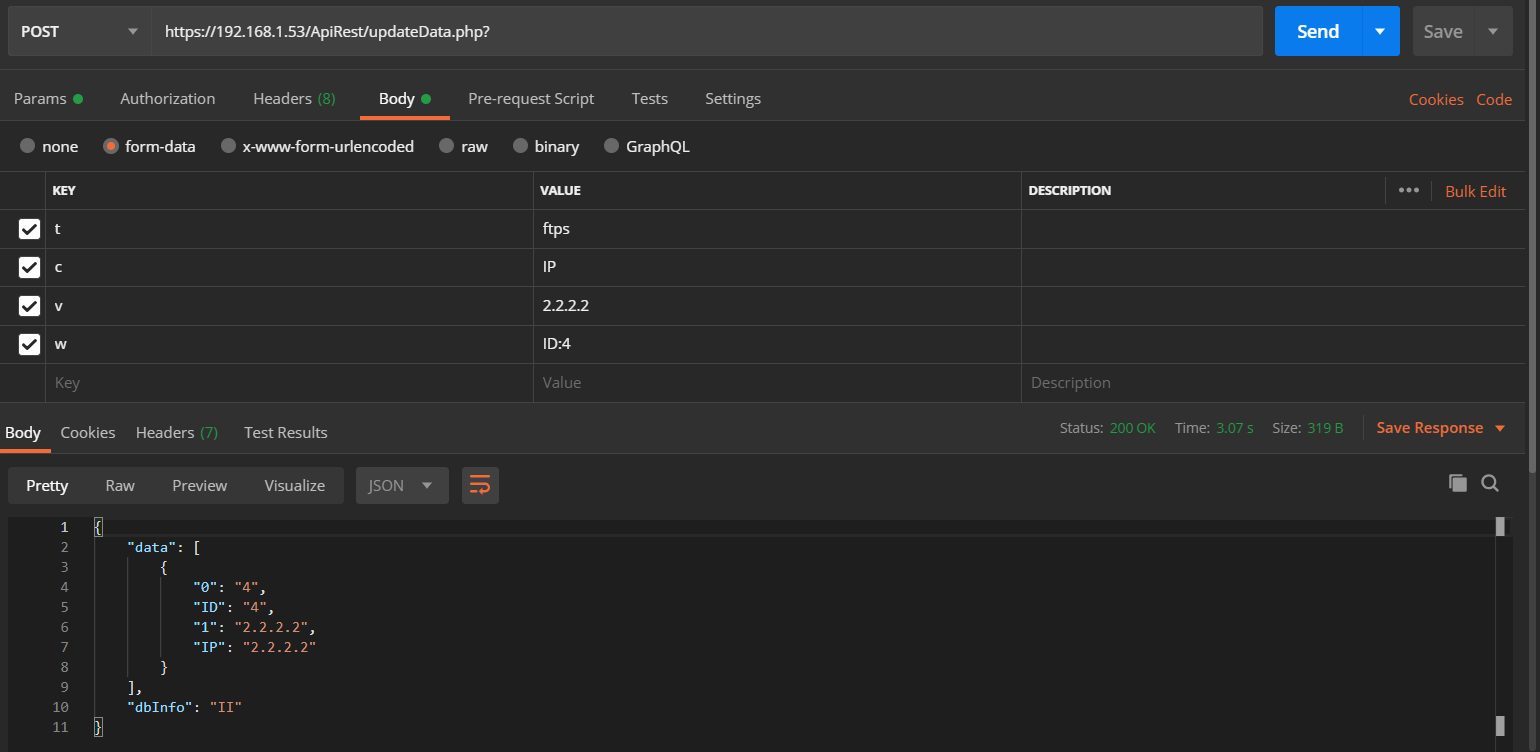
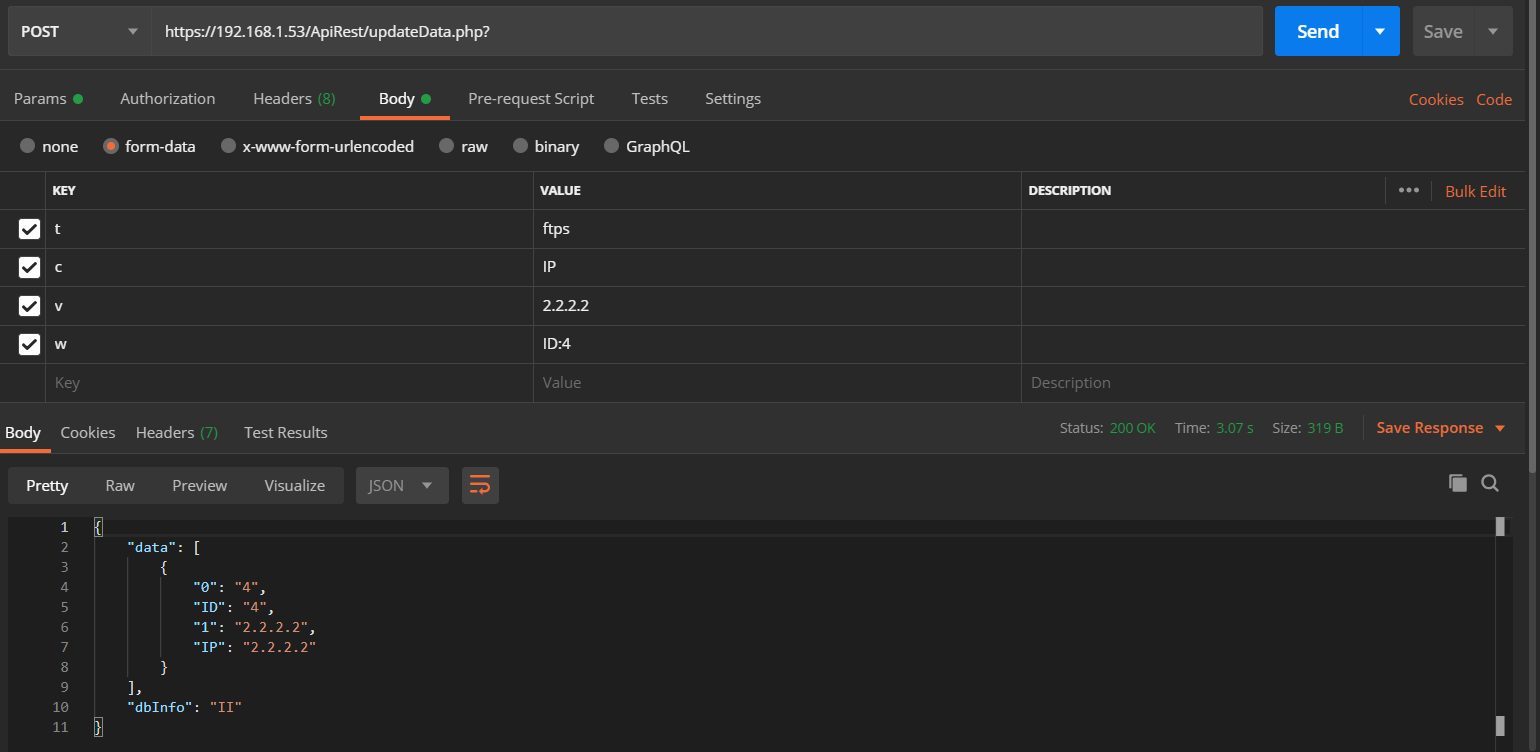
***Prueba desde Postman*** que es un programa que nos permite probar las funcionalidades de nuestra API REST para enviar parámetros por la URL por medio de los diferentes modos (GET, POST, PUT, entre otros).

Primero consultamos los registros que tiene la tabla ftps (Estos son los del servidor de prueba):

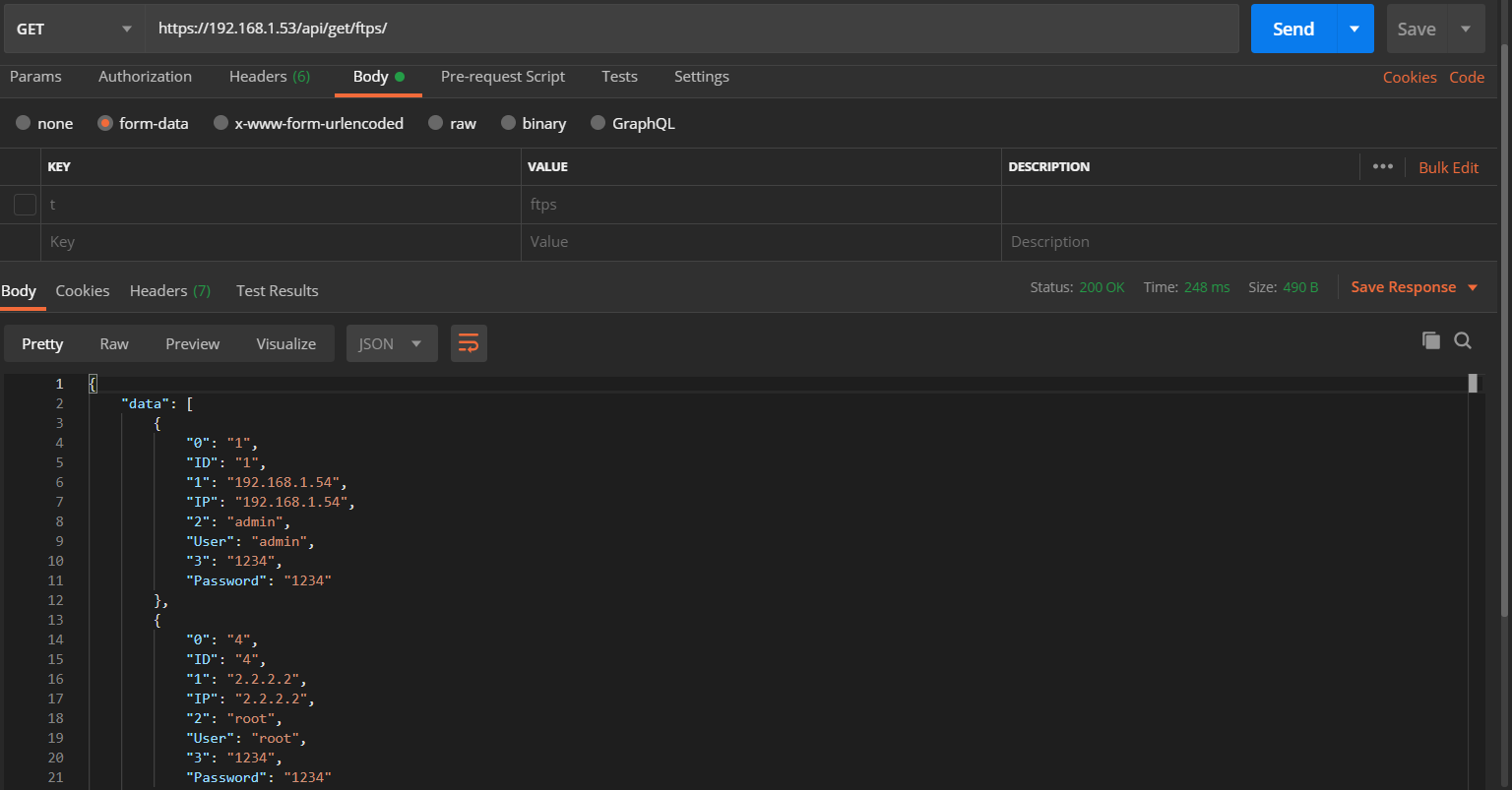
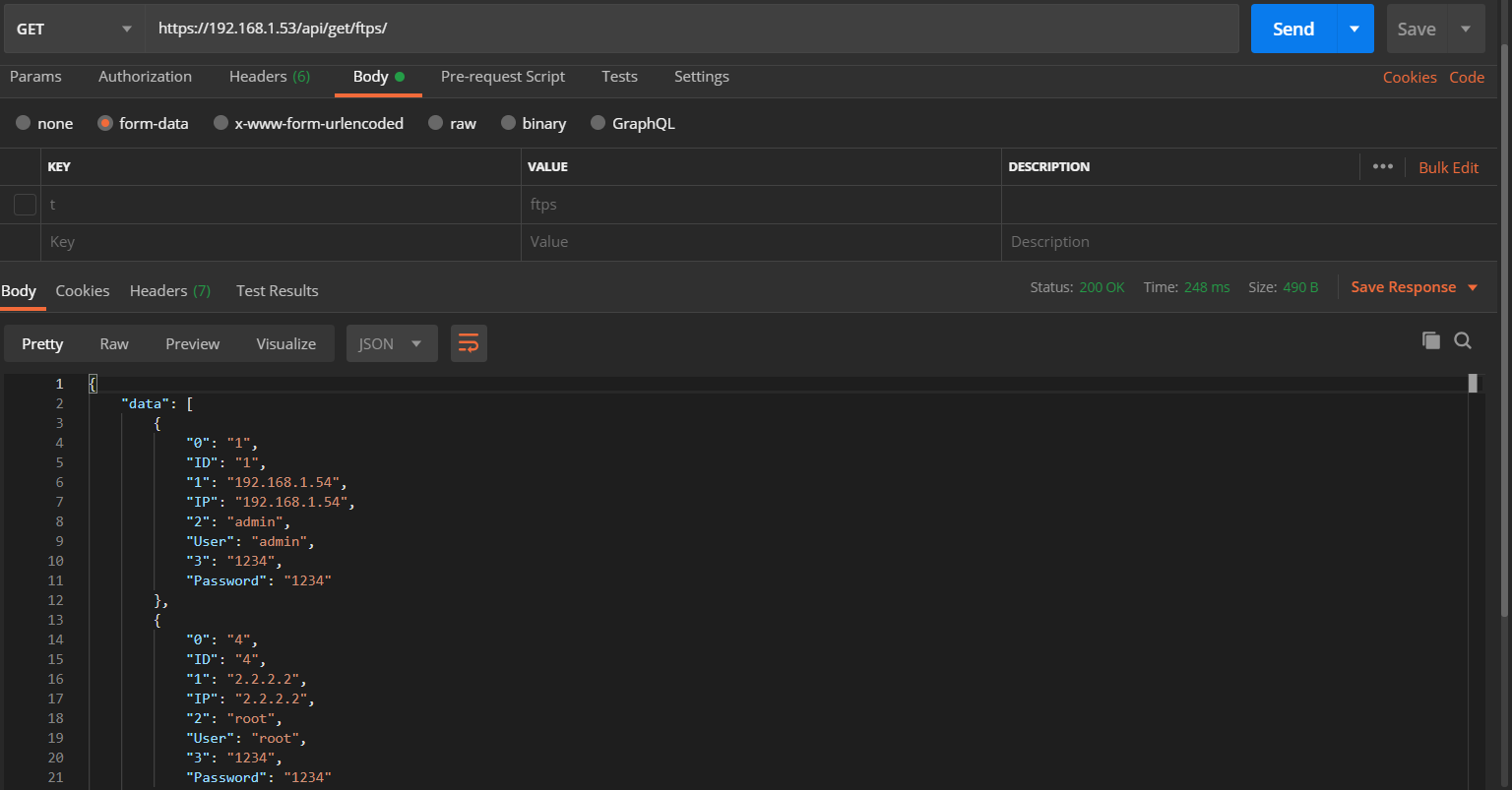




Luego se procede a cambiar la IP del registro con ID igual a 4:

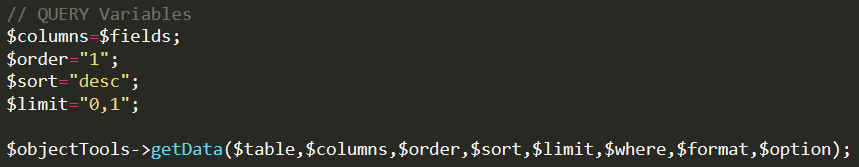


Y ahora volvemos a consultar todos los registros:



## 2.2) postData.php

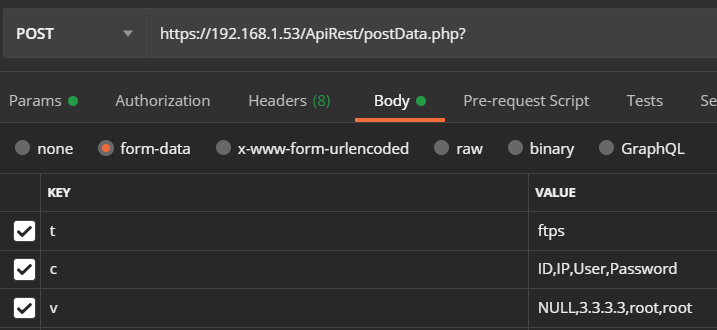
Por otro lado, la opción de insertar datos ya viene implementada, pero esta retorna el ID incorrecto del registro que se acaba de insertar por lo que se modifica la parte después de construir la consulta insert, que se hace lo mismo que con update, se llama un método de la clase Tools para consultar el último registro de la tabla, por ello se especifican todas las columnas, se ordena descendentemente por el ID y se limita solo a un resultado:

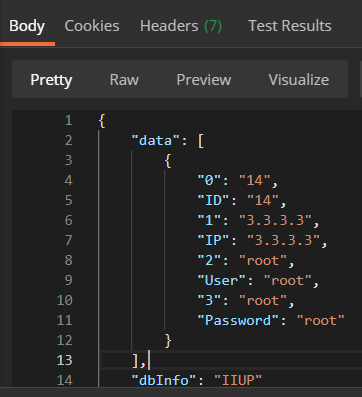


Al igual que update, se envían los parámetros por la URL así con el método POST:

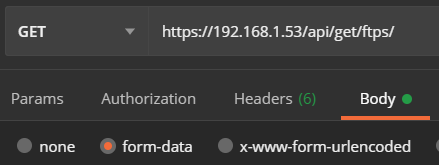
*https://192.168.0.101/ApiRest/postData.php?t=table&c=column1,column2&v=value1,value2*

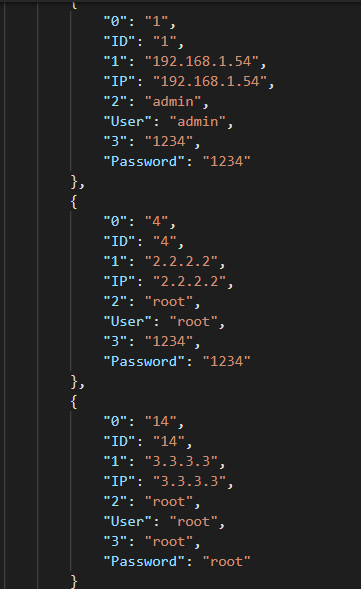
***La prueba en el programa Postman*** para insertar un nuevo registro:





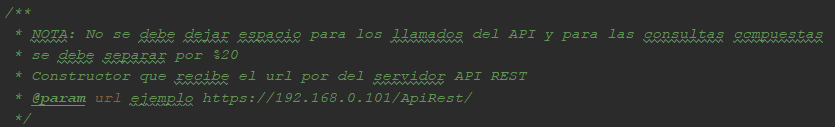
Y la consulta sobre la tabla donde se ve el nuevo registro:





# 3) ConexionApiRest

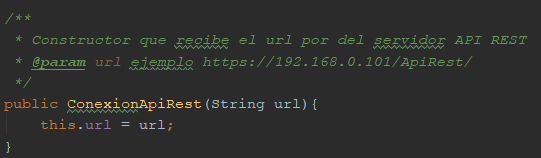
Con un mejor entendimiento del API REST se realiza una mejora de la clase ***ConexionApiRest*** haciendo que sea de más alto nivel para facilitar su uso y comprensión, además todos los métodos llevan el comentario con su descripción de lo que realiza, recibe y retornar, ejemplo:



Y como se ve en el comentario, en todos los llamados de los métodos del API REST no se puede dejar espacio ya que estos se hacen por la url, entonces para las consultas compuestas se debe separar escribiendo *%20*

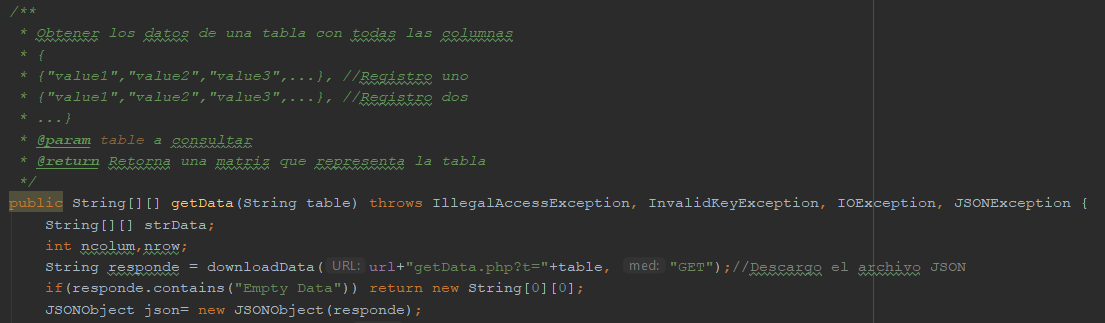
## 3.1) Constructor

Recibe la URL del servidor: *https://192.168.0.101/ApiRest/*

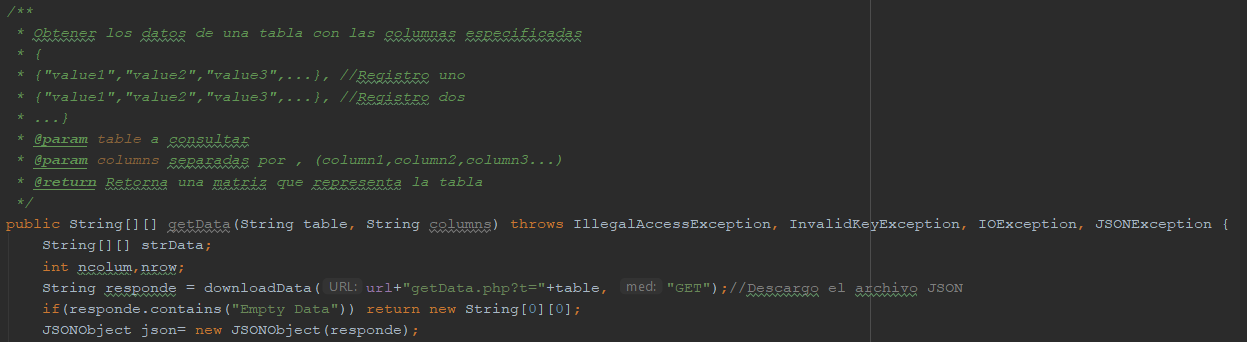


## 3.2) getData

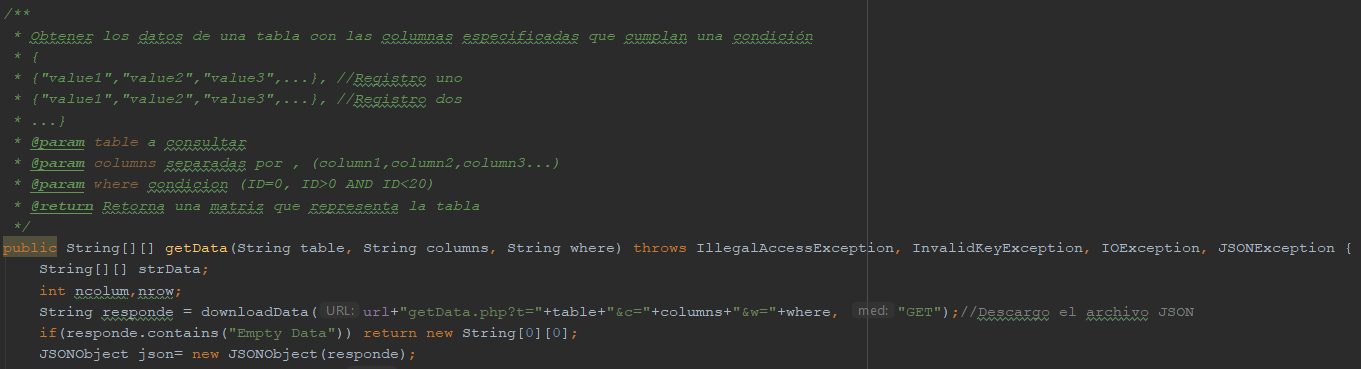
En la decodificación del JSON se mantiene, cambia la forma en que como se llama y en los parámetros que recibe. En este sentido puede obtener el contenido de toda la tabla:



Obtener solo algunas columnas de la tabla:

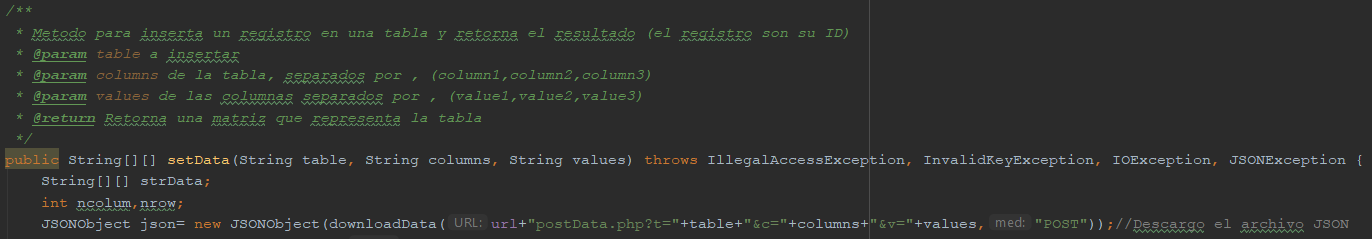


Obtener algunas columnas filtradas por un where:



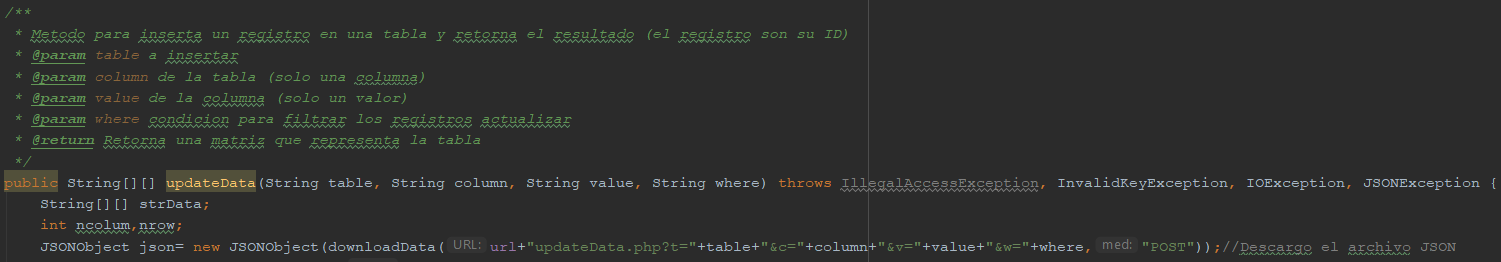
## 3.3) setData

El método enviar por la URL los parámetros para realizar la inserción por el método POST de HTTP y como también devuelve un JSON, tiene la misma decodificación que el método getData, por lo que cambia el URL a enviar a download y se especifica el método POST:



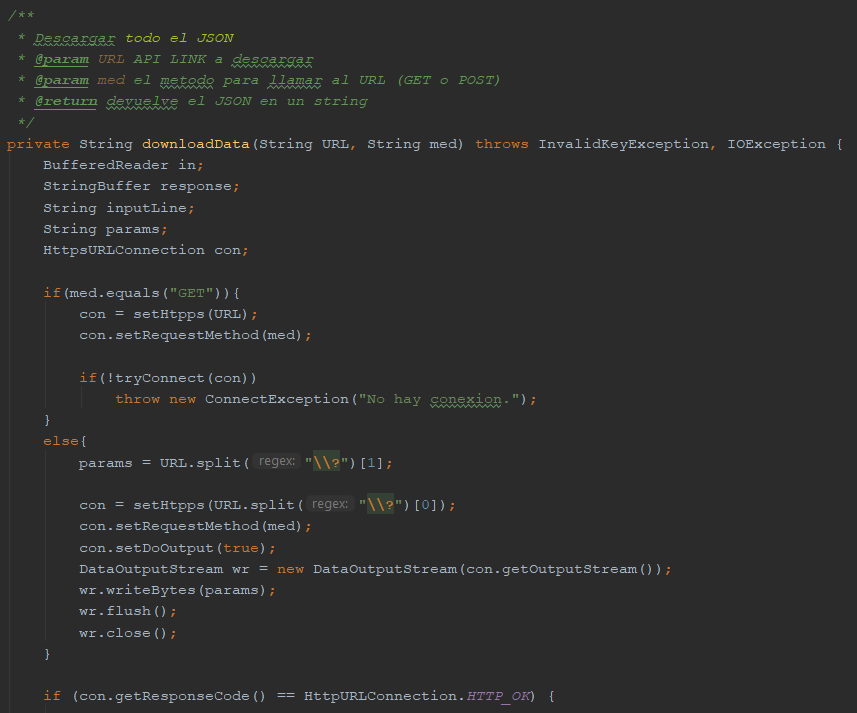
## 3.4) updateData

Al igual que setData, solo cambia el URL a enviar a download, ya que también se devuelve un JSON con el ID y la columna cambiada:



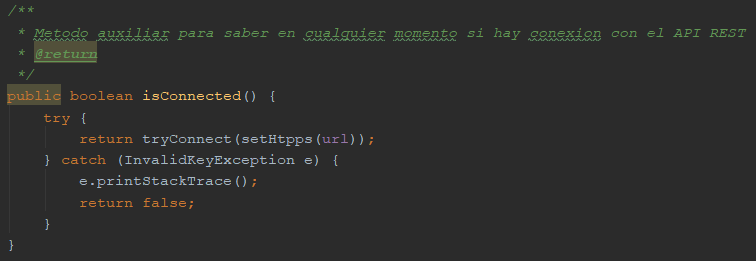
## 3.5) downloadData

Método privado que se encarga de descargar los JSON que retornan los API LINK, además de llamar un método para comprobar la conexión con la página cuando se van a consultar datos:



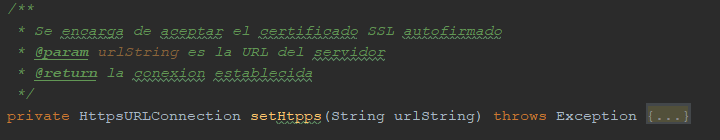
## 3.6) isConnected

Método para saber si el dispositivo tiene conexión al API REST en cualquier momento, este será llamado siempre que se vayan a realizar las actualizaciones o sincronizaciones en el nodo para que la app no se pared y deje de responder:



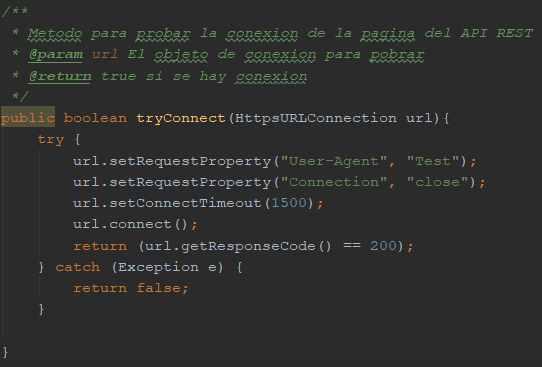
## 3.7) setHttps

Método privado que se encarga de gestionar el permiso para aceptar la conexión con servidor ya que este tiene un certificado SSL auto-firmado:



## 3.8) tryConnect

Método para probar la conexión de una página (API LNK) de forma rápida y evitando que se cuelgue la app cuando no se pueda acceder al nodo:



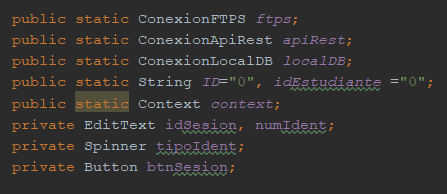
# 4) EU Estudiante

Se hará una breve recopilación de los llamados a las clases auxiliares para manejar la conexión con el servidor MariaDB y con el servidor FTP, así como las clases necesarias para el manejo del front y las nuevas implementaciones que abarca el manejo de la base de datos SQLite interna y todo el código para iniciar, actualizar y sincronizar los estados de las actividades.

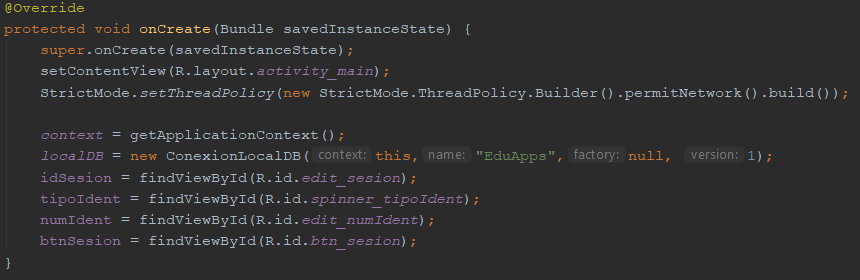
## 4.1) MainActivity

Esta actividad se encarga de iniciar el estado de la actividad, si es primera vez que el estudiante inicia sesión para dichas actividades, insertando los datos del nodo a la base de datos local o actualizando las tablas locales con la base de datos que está en el nodo (sin modificar la tabla que tiene el estado de las actividades), esta última opción si ya se ha iniciado sesión en el dispositivo con anterioridad. Además, si el dispositivo ya tiene información de la sesión que se desea iniciar y no hay conexión con el nodo, se permite entrar en estado *offline* para que el estudiante pueda visualizar las actividades y poder cambiar el estado de cada una de ser necesario.

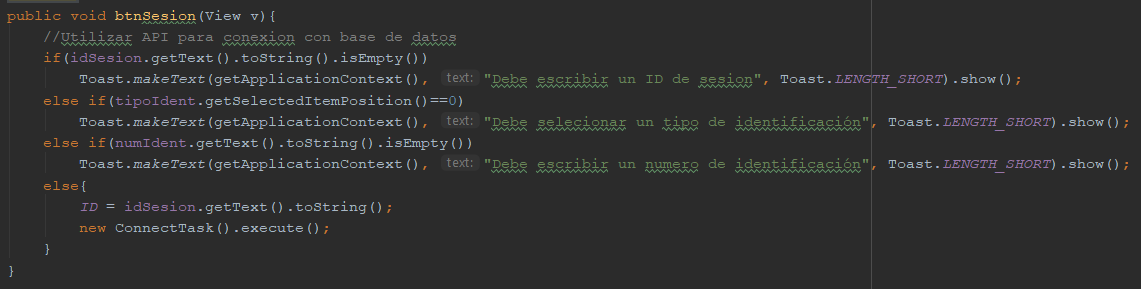
Primero, se establecen las variables para los objetos de las vistas y las instancias de conexión global como estáticas para no estar creando múltiples objetos con el mismo fin:



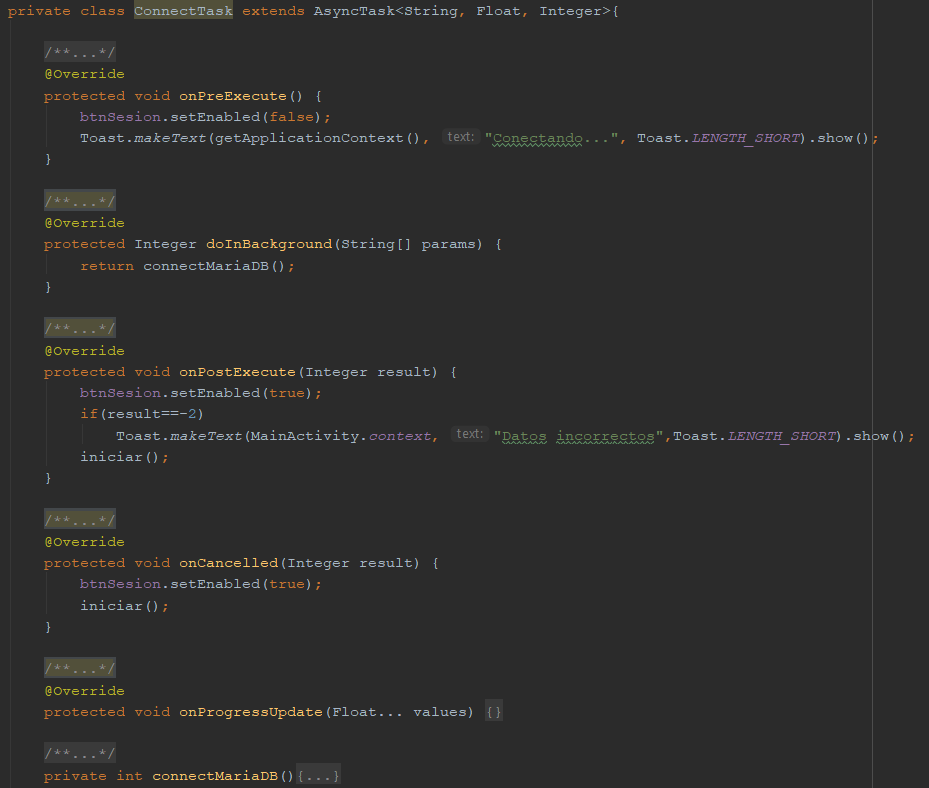
Luego, en el onCreate se hacen las asignaciones necesarias y se habilita la regla de descargar la información desde el hilo principal con la línea de *StrictMode*:



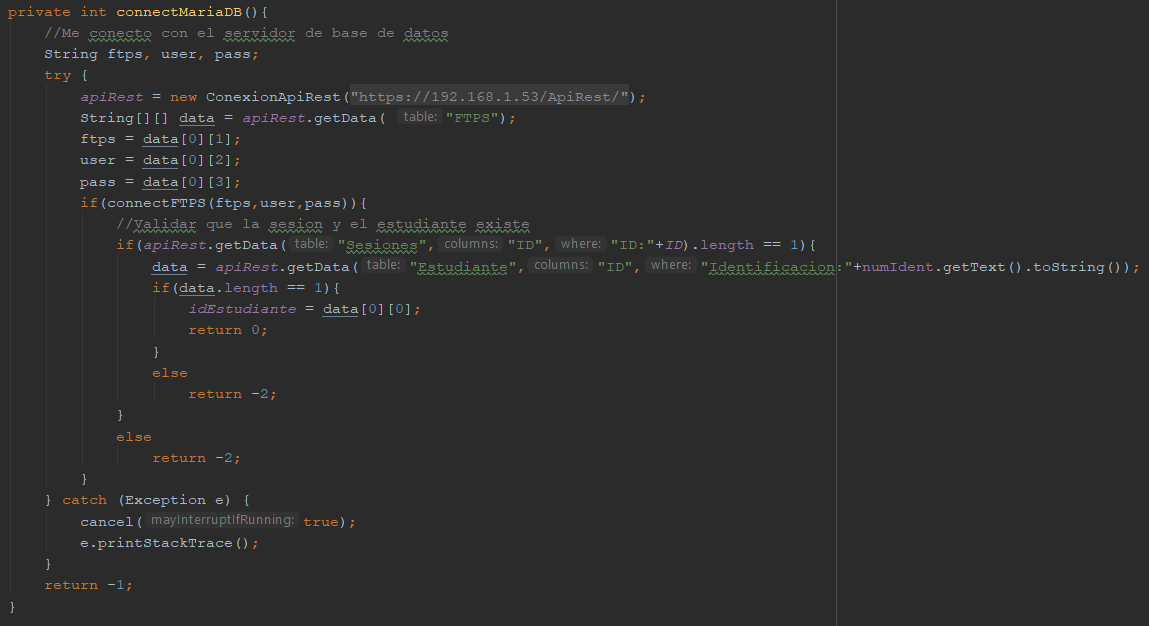
Cuando se dé clic en el botón *Iniciar sesión* se obtiene la información de los objetos haciendo una primera validación de que estos no estén nulos:



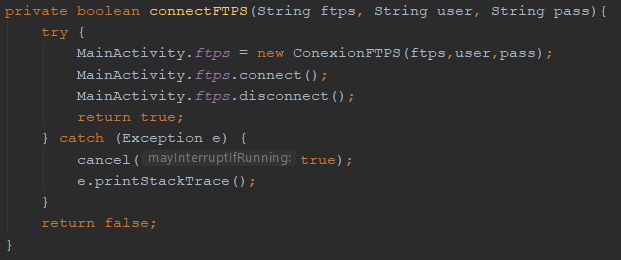
Luego se ejecuta un hilo para probar la conexión con el API REST, que puede retornar 3 valores; 0 (conexión exitosa), -1 (conexión fallida), -2 (Datos erróneos). Además, antes de que se ejecute el hilo se deshabilita el botón de iniciar para evitar que el usuario haga múltiples peticiones:



En el método *connectMariaDB* se realiza la prueba con el API REST, que si es exitosa se establece la variable *online* en *true* y se obtienen los datos del servidor FTP, la sesión y el estudiante para validar que los datos ingresados sean correctos:

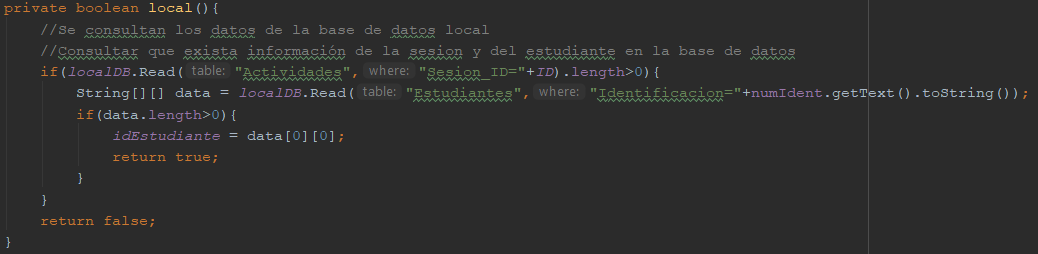


Luego, en el método *connectFTPS*, se valida la conexión con el servidor FTPS según los datos del nodo:



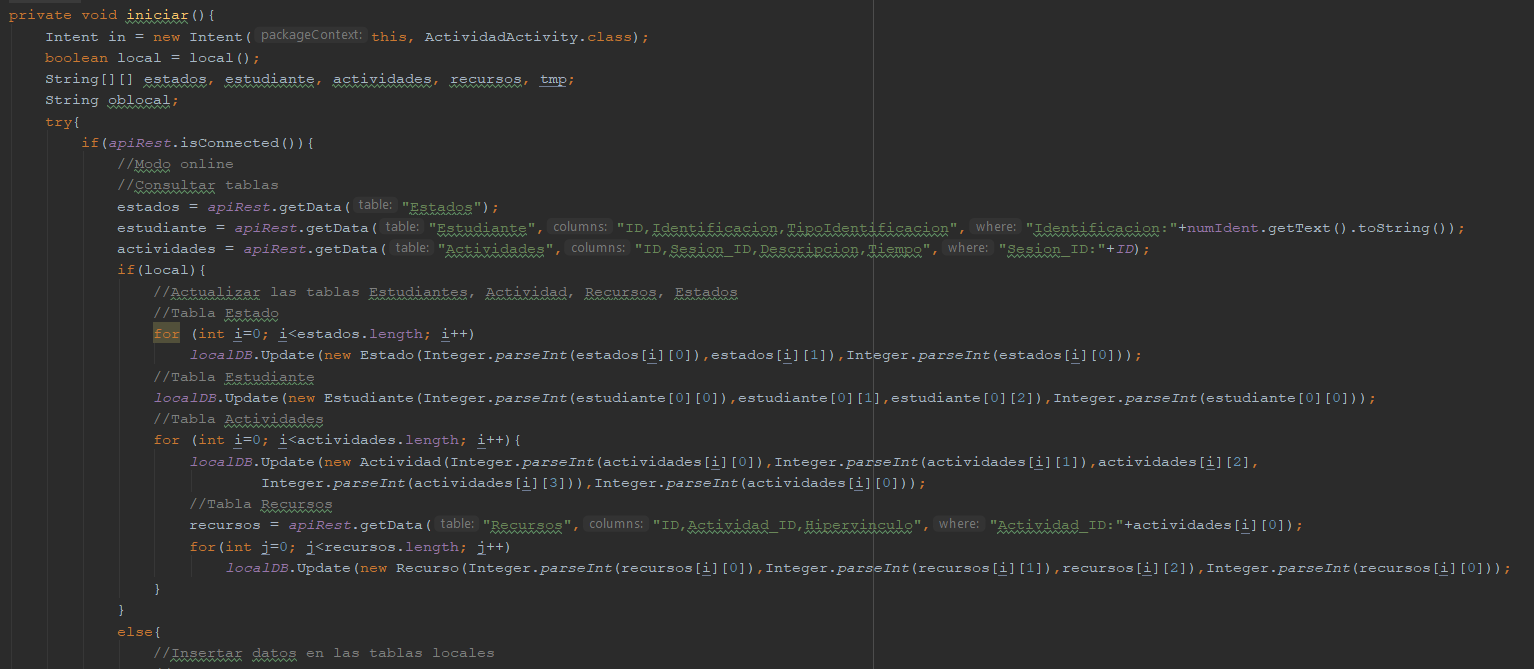
### 4.1.1) Iniciar el estado de la actividad

Posterior a la ejecución del hilo, se ejecuta el método *iniciar* que llama al método *local* para saber que hay información de la sesión en la base de datos local:

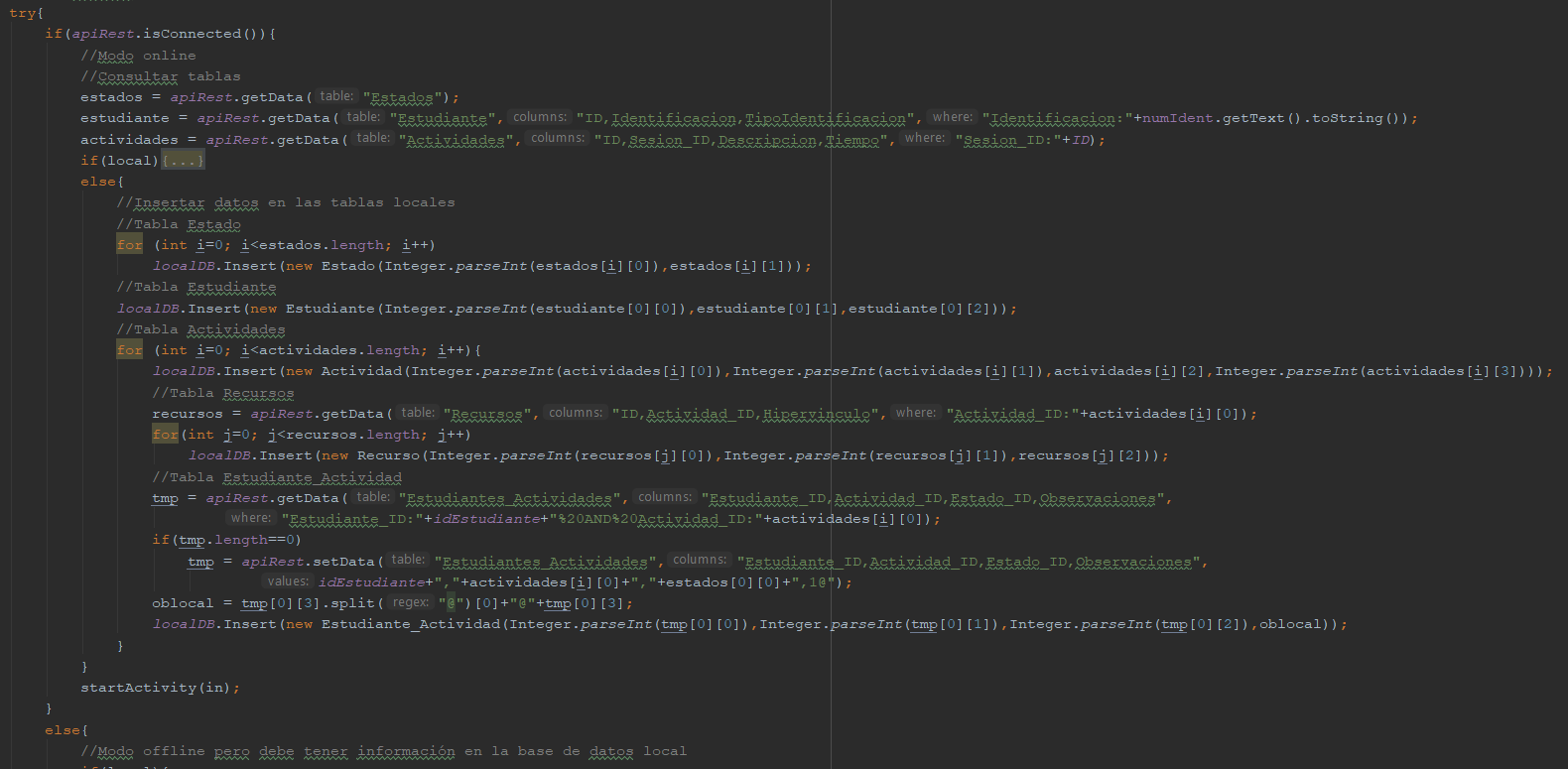


Luego de esto puede pasar lo siguiente:

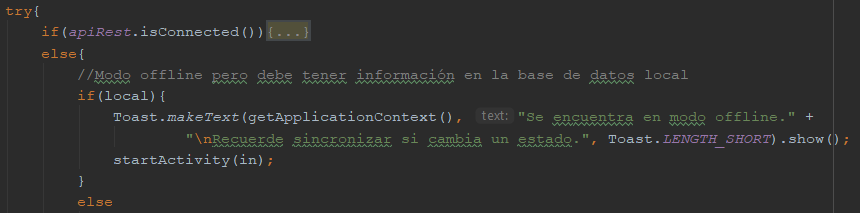
* Si se está en modo *online* y hay información en la base de datos local se actualizan las tablas locales excepto la tabla que contiene los estados de las actividades ya que posteriormente (en la *ActividadActivity* se deben sincronizar), y luego se inicia la actividad siguiente:



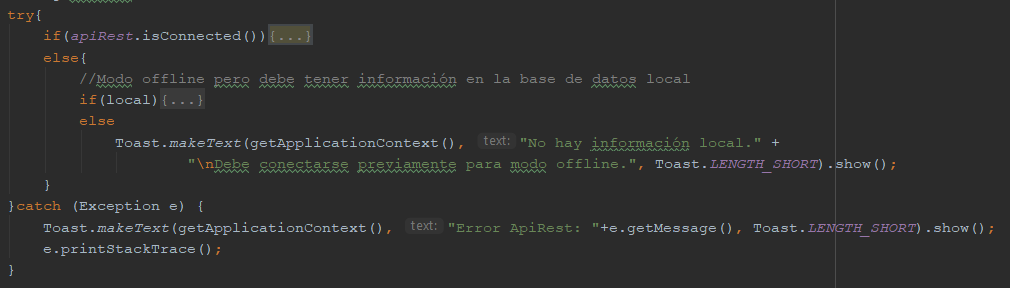
* Si se está en modo *online* pero sin datos en la base de datos local, se insertan en todas las tablas de la base de datos SQLite local y se consulta si es la primera vez que se va a ingresar a la sesión consultando los datos de los estados, **si no existe información se insertan los datos y se ponen las actividades en estado *Iniciada***, y luego se inicia la *ActividadActivity*:



* Si se está en modo *offline* y con datos locales se inicia la *ActividadActivity* que se encarga de cargar la información para visualizar las actividades:

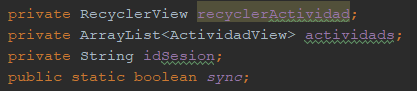


* Si se está en modo *offline* pero sin datos en la base de datos local, no se puede iniciar debido a que debe haber un ingreso previó *online* para poder tener información en la base de datos local:

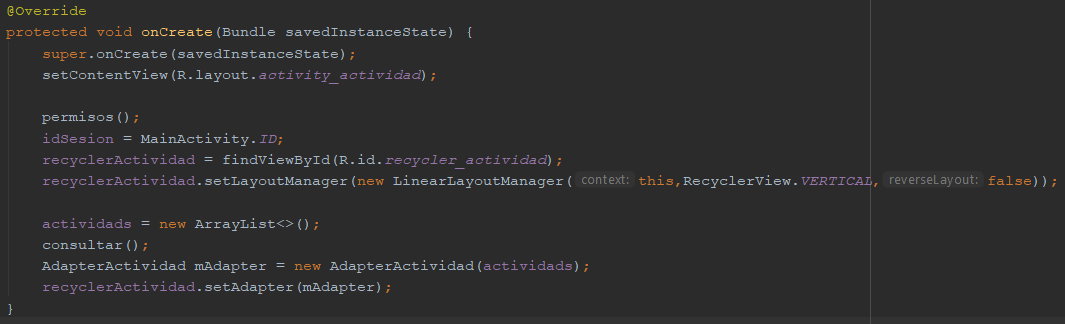


## 4.2) ActividadActivity

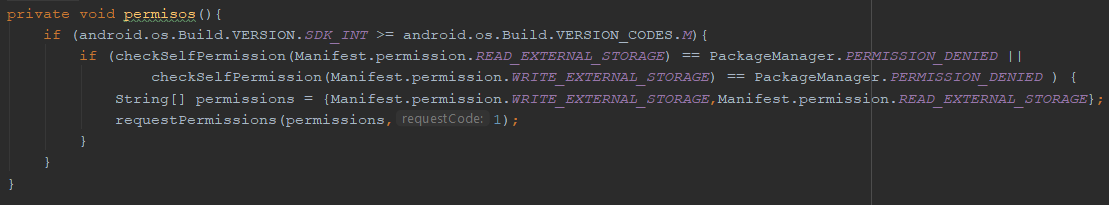
Se establecen las variables que se utilizaran, así como una lista de *ActividadView* que es una clase que nos ayuda al manejo del Front junto con la clase *AdapterActividad*, para establecer todas las actividades de una sesión, además de una variable boolean para saber si aún no se ha sincronizado el dispositivo con el nodo:



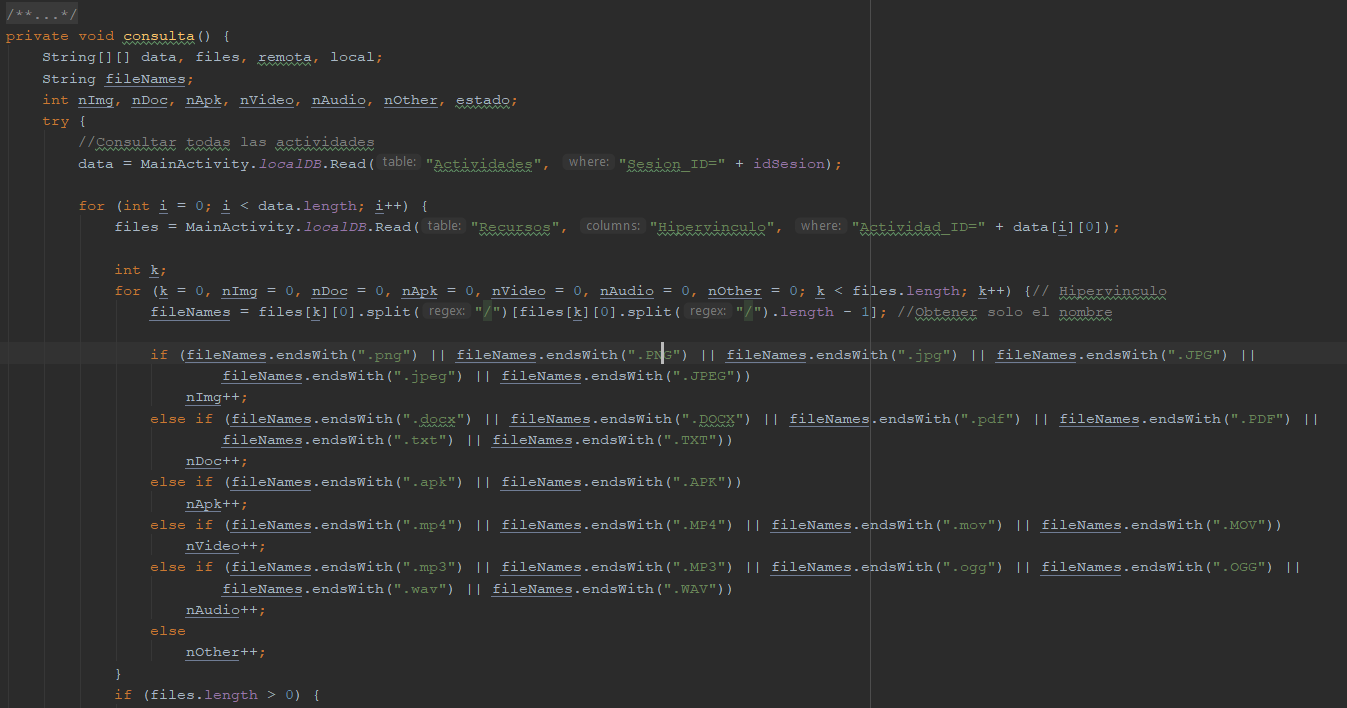
Luego en el onCreate, se piden los permisos de almacenamiento, así como obtener y establecer el objeto de la vista (RecyclerView) y luego consultar las actividades de la sesión para ponerlas en la lista:



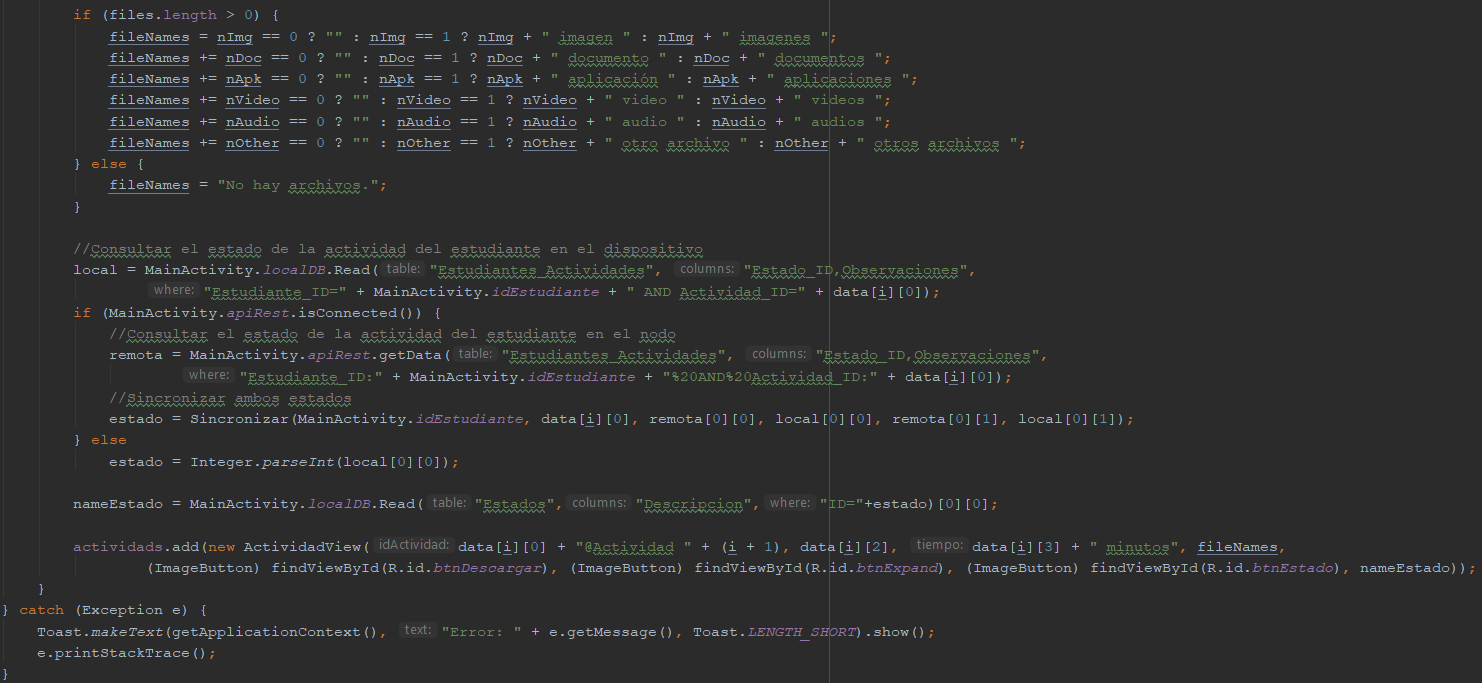
El método de los permisos es el mismo que se ha utilizado en los sprint pasados:



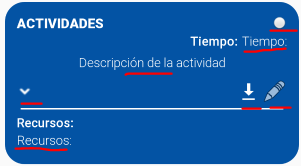
Y el de consulta cambia realizando todas las consultas sobre la base de datos local, ya que al iniciar sesión se valida que se cuente con ella para poder estar en modo *offline*, además con esto ahorramos recursos y procesamiento al no tener que descargar la información en una página porque ya se tiene localmente lo cual permite un mejor rendimiento de la app:



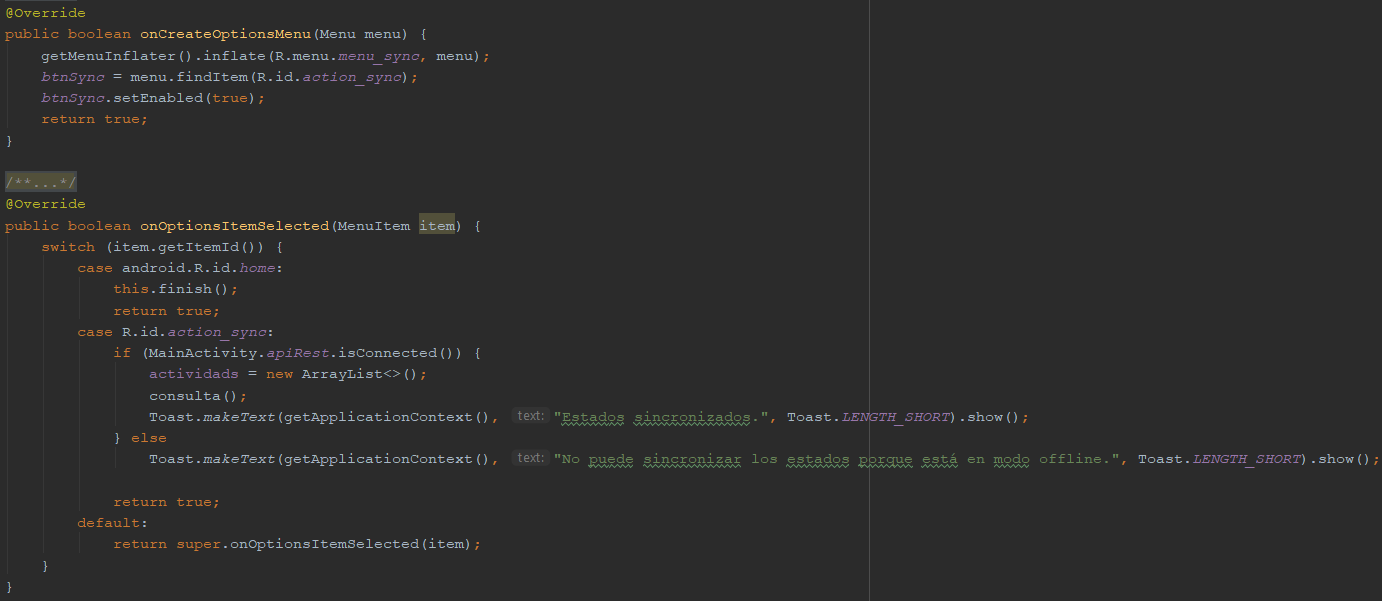
Se analizan los archivos para mostrar de una forma más agradable los recursos de las actividades y luego si se está *online* se realiza la sincronización de forma automática, para esto se consulta el estado local y remoto de las actividades, así como las observaciones que contienen los metadatos que permitirán aplicar el modelo de sincronización (***ver 4.3 Modelo de sincronización***) que se realiza en la función *sincronizar*:



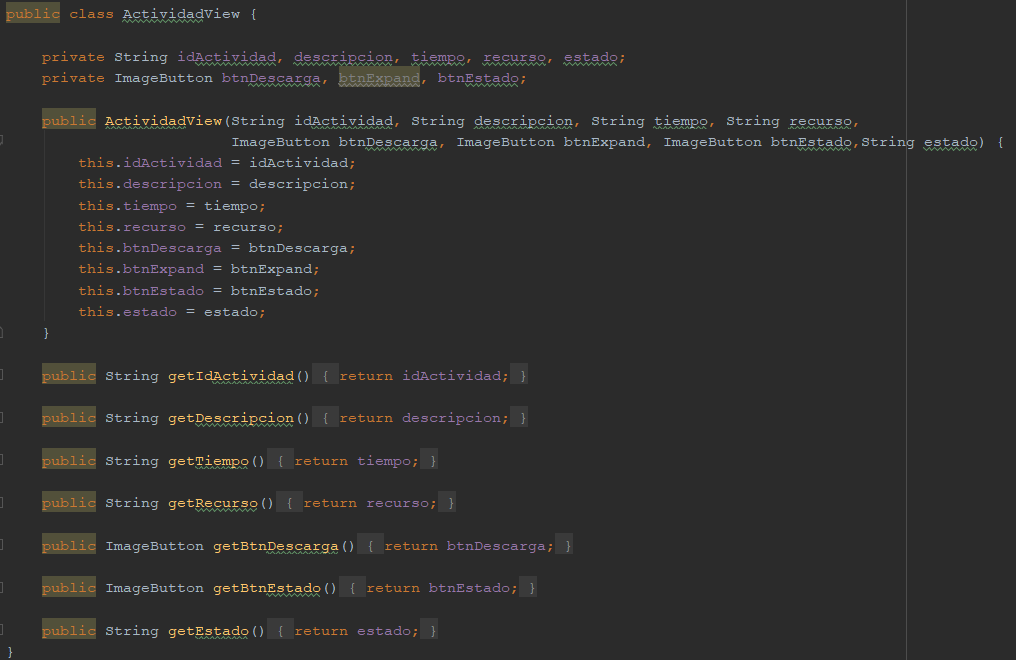
Cabe aclarar, que aumentan los parámetros del constructor de *ActividadView* porque deben estar todos los elementos (líneas rojas) que se cambian o interactúan del *CardView*:



También en la *ActividadActivity* se asigna un menú para poder sincronizar el estado local de las actividades con el estado en el nodo por si se estuvo trabajo offline no perder el progreso del estudiante, al presionar el botón vuelve a consultar la información que a su vez llama al método sincronizar:



La clase *ActividadView* contiene los atributos para cada uno de esos, el *estado* es el que me permite identificar el icono que se debe mostrar para la actividad:

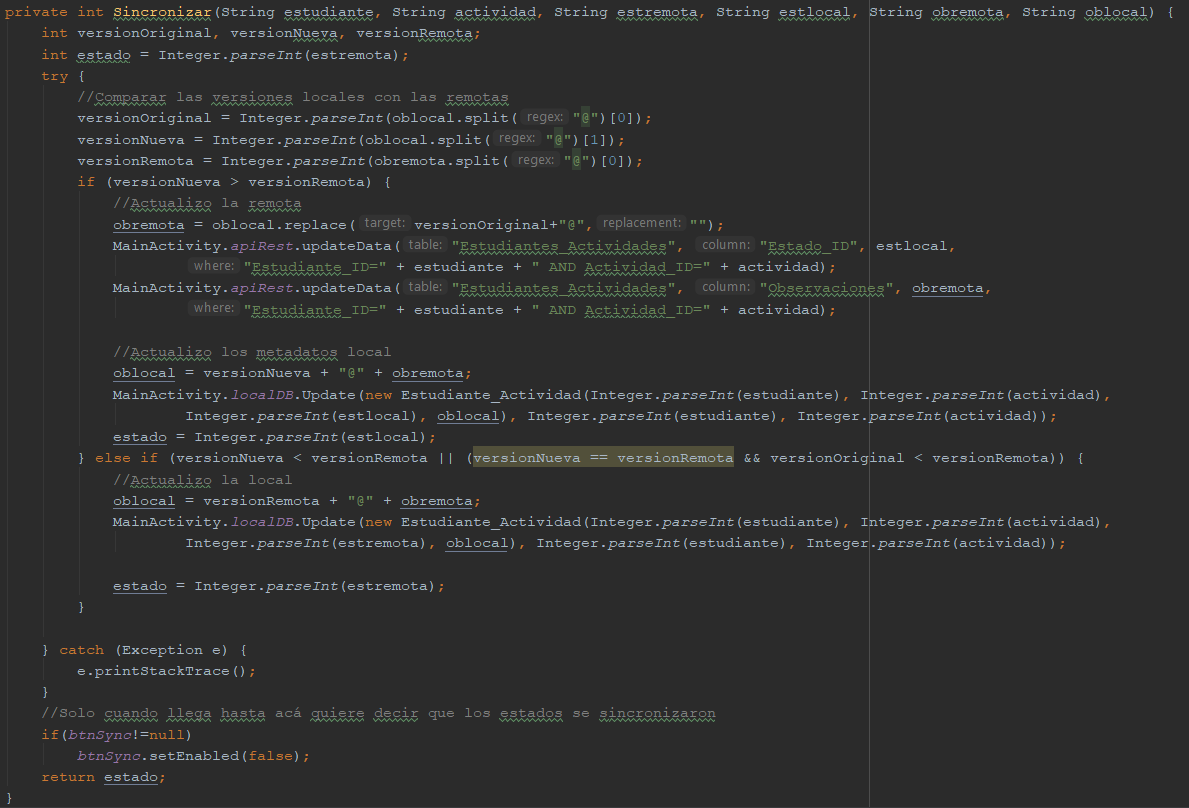


### 4.2.1) Sincronizar SQLite local con nodo MariaDB

Gracias al método *isConnected* la aplicación siempre que tenga haya una conexión los estados se sincronizará automáticamente, de la siguiente forma:

* Si se inició en modo *online* pero se perdió la conexión, al cambiar el estado aparecerá un mensaje diciendo que está en modo offline, y si vuelve la conexión la cambiar un estado, este automáticamente se sincronizarán los estados.
* Si se inició en modo *offline* y se hicieron cambios aparece un mensaje recordando que luego se debe sincronizar, sin embargo, si vuelve la conexión al cambiar un estado se sincronizará este automáticamente.

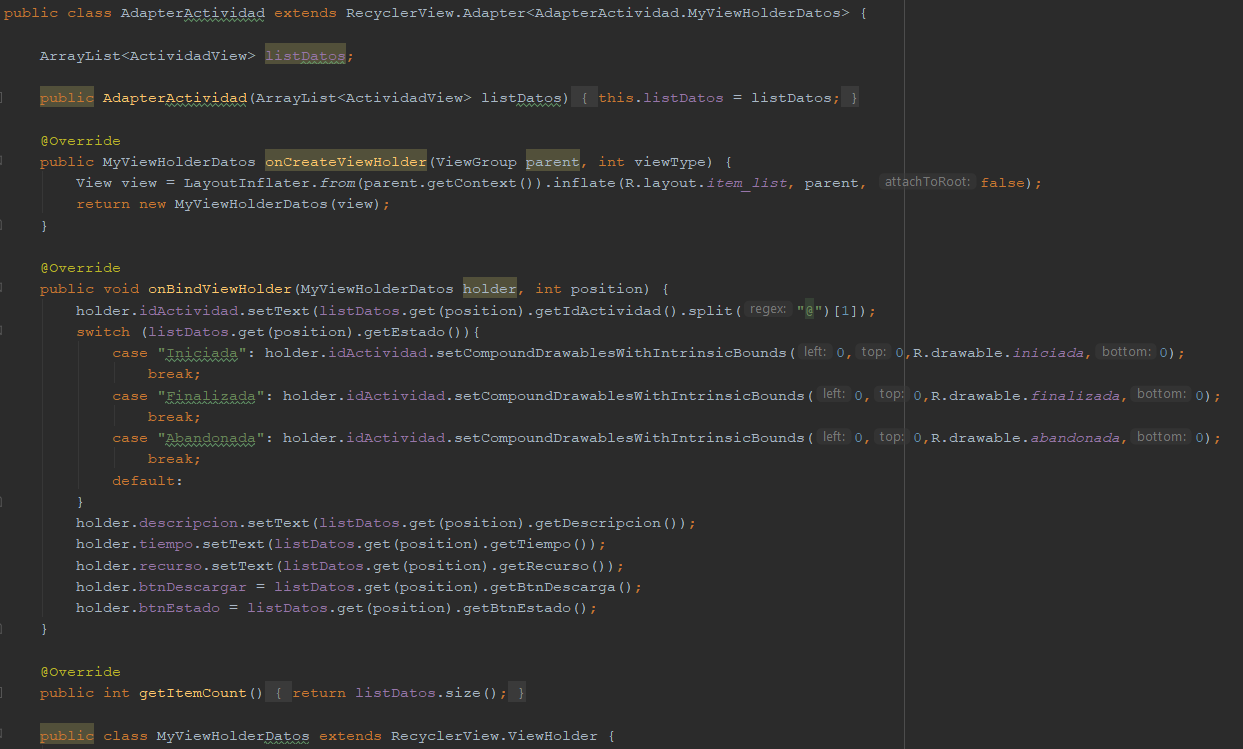
No obstante, estas opciones sincronizarían solo el estado que se acaba de cambiar en ese instante por ello, si se hicieron cambios en modo *offline* y se vuelve a iniciar sesión con conexión se sincronizarán todos los estados automáticamente o si estando en modo *offline* vuelve la conexión se puede dar clic en el botón *sincronizar* (ubicado en la esquina superior derecha) para sincronizar todos los estados que vuelve a realizar la consulta que llama al método sincronizar que me devolverá el estado sincronizado para mostrar en la lista:



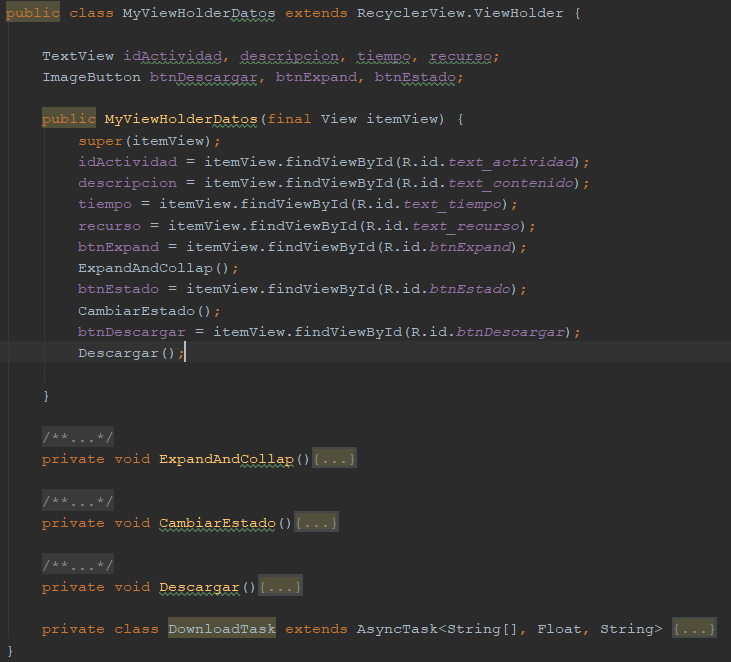
### 4.2.2) Actualizar el estado de la actividad

Ahora bien, la clase *AdapterActivity* es la que se encarga de varias cosa:

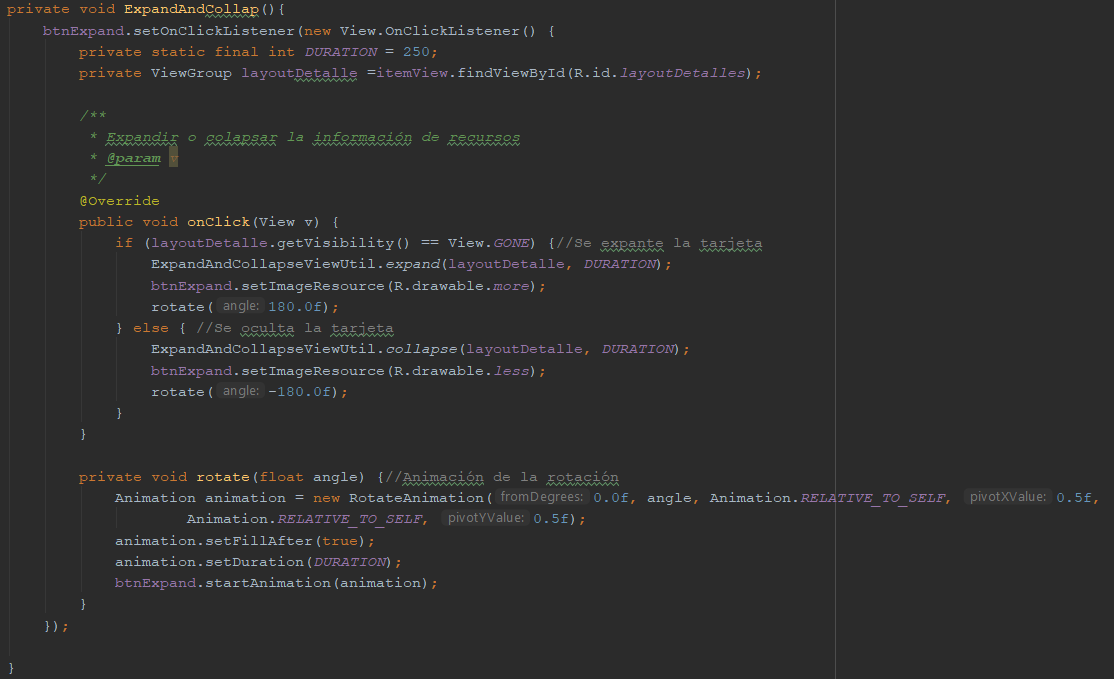
* Replicar la tarjeta tantas veces como actividades hayan, ademas es allí donde se codifica toda la logica cuando se hace clic en un botón de la *CardView*:



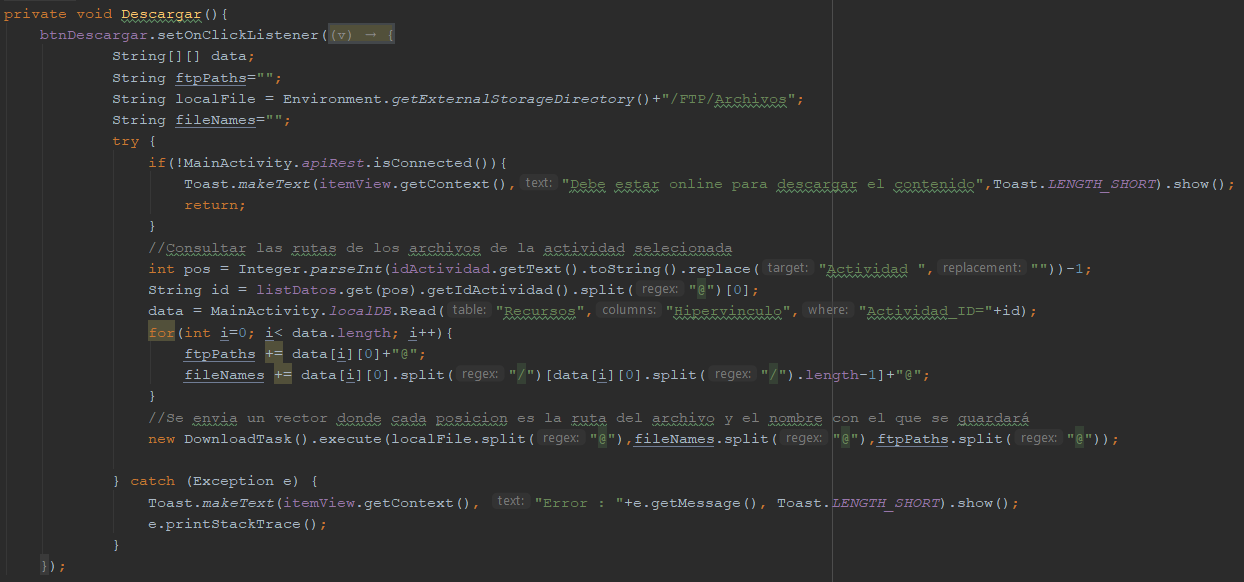
* Para los eventos cuando se hace clic se debe asignar el método *setOnClickListener* en la parte de la clase *MyViewHolderDatos*, primero se establecen las variables y para que el código sea más legible cada evento para los clics (expandir, cambiar estado y descargar):



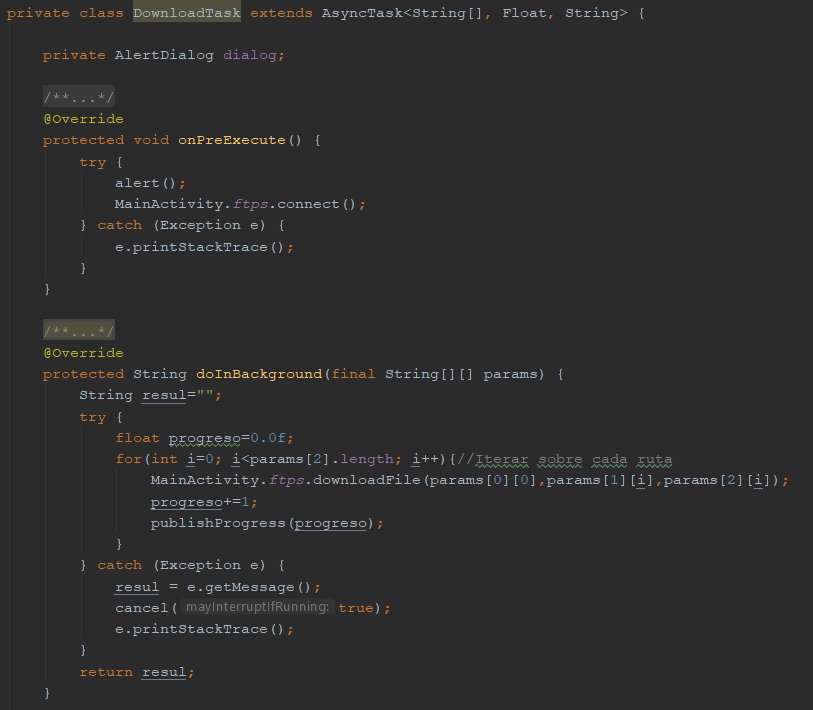
* El método para asignar el evento cuando se desea expandir o colapsar la información de la actividad:



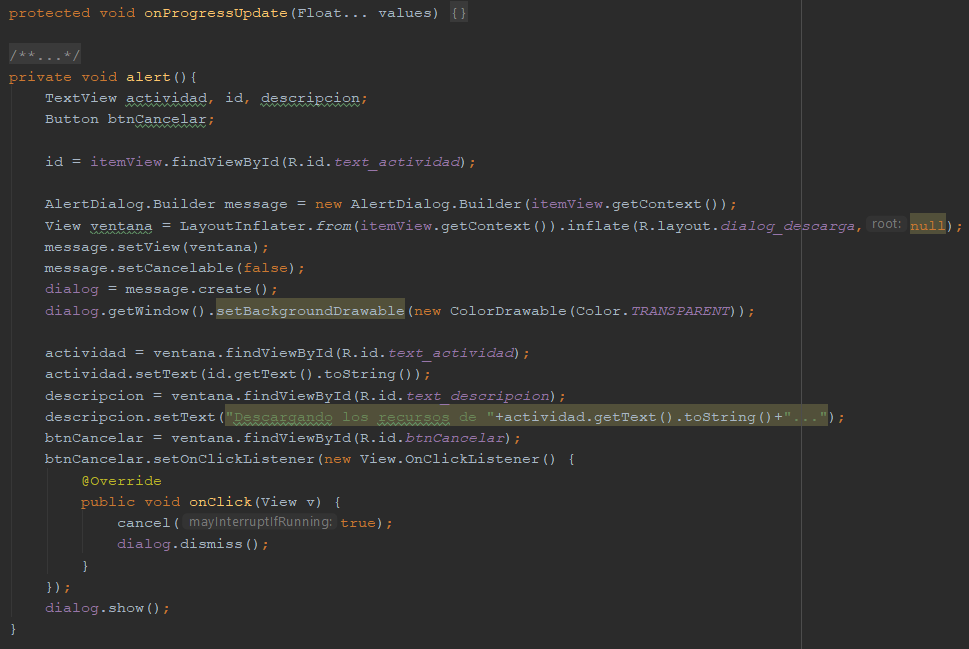
* El método para descarga los recursos de una actividad, que hace uso de un hilo para no bloquear el funcionamiento del hilo principal y tener una mejor experiencia de usuario:



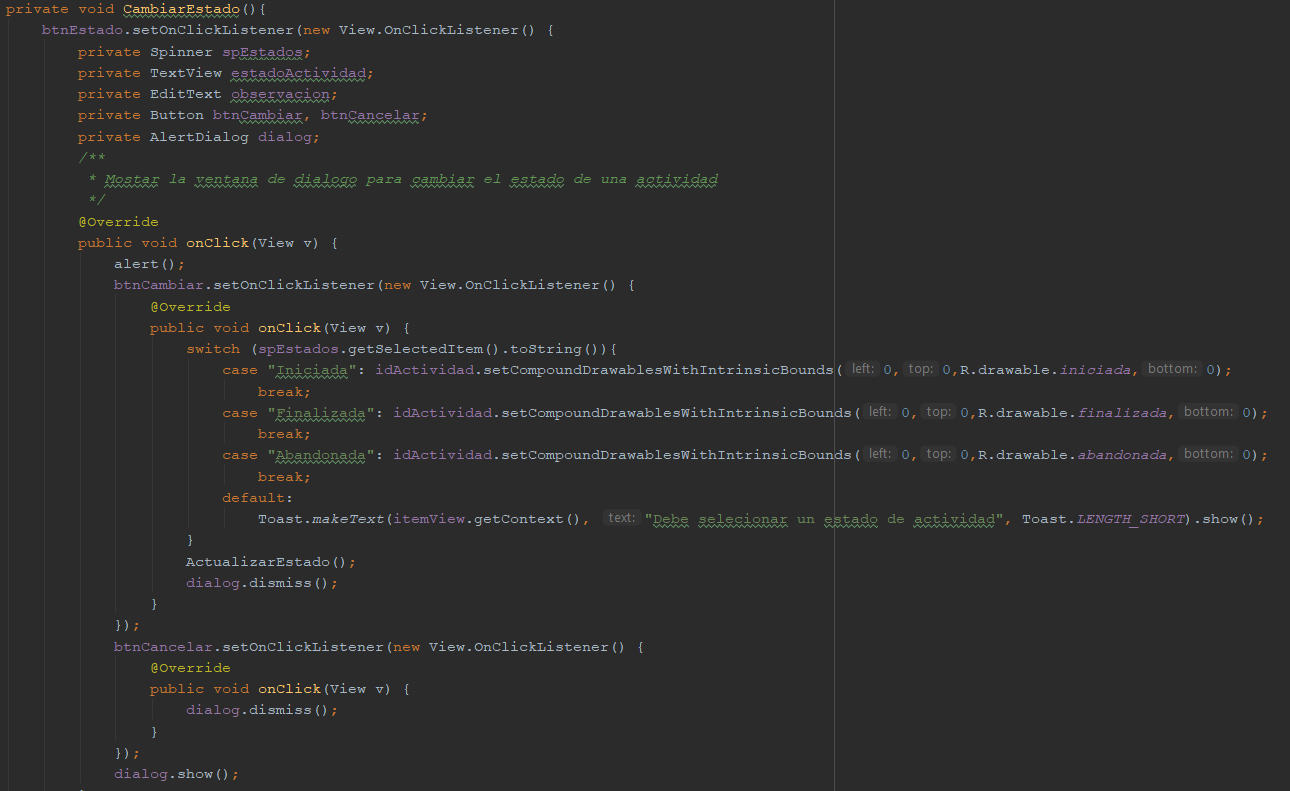
El código de la clase *DownloadTask*, que es una clase para realizar tareas asincrónicas en segundo plano, que establece la ventana de dialogo por el método *alert* y en el método *doInBackground* descarga los recursos:

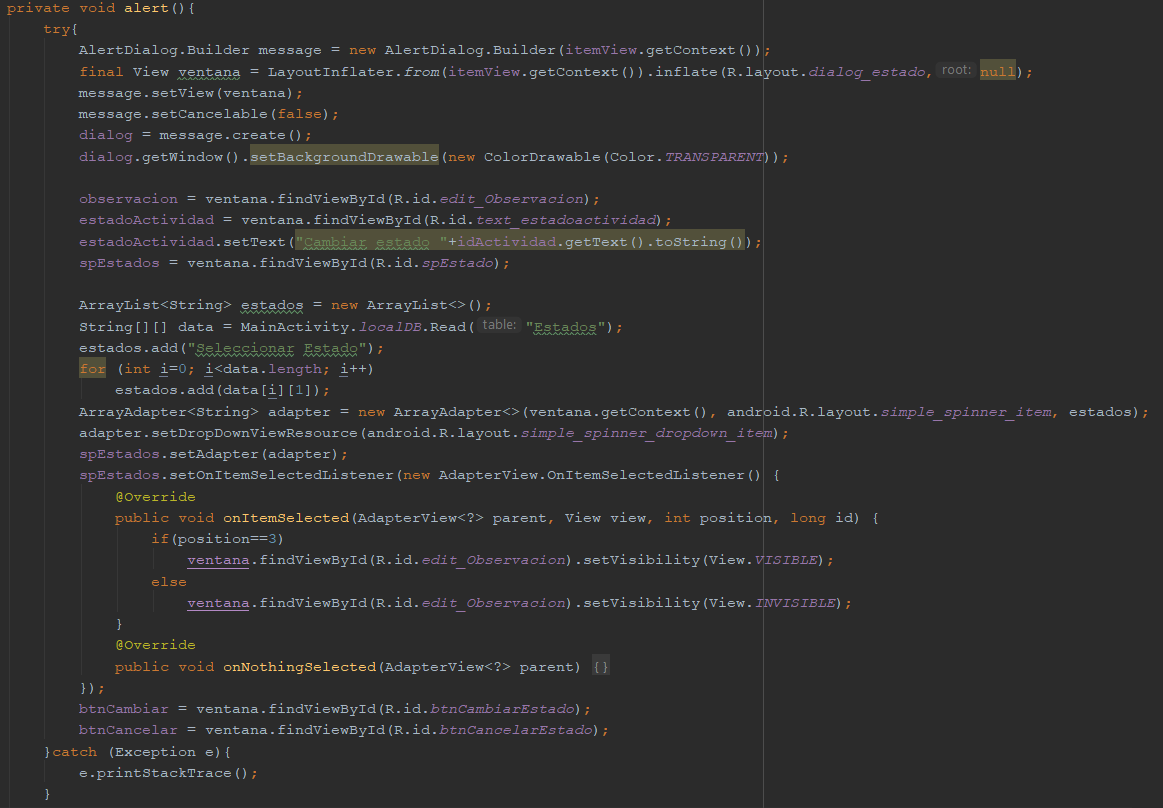


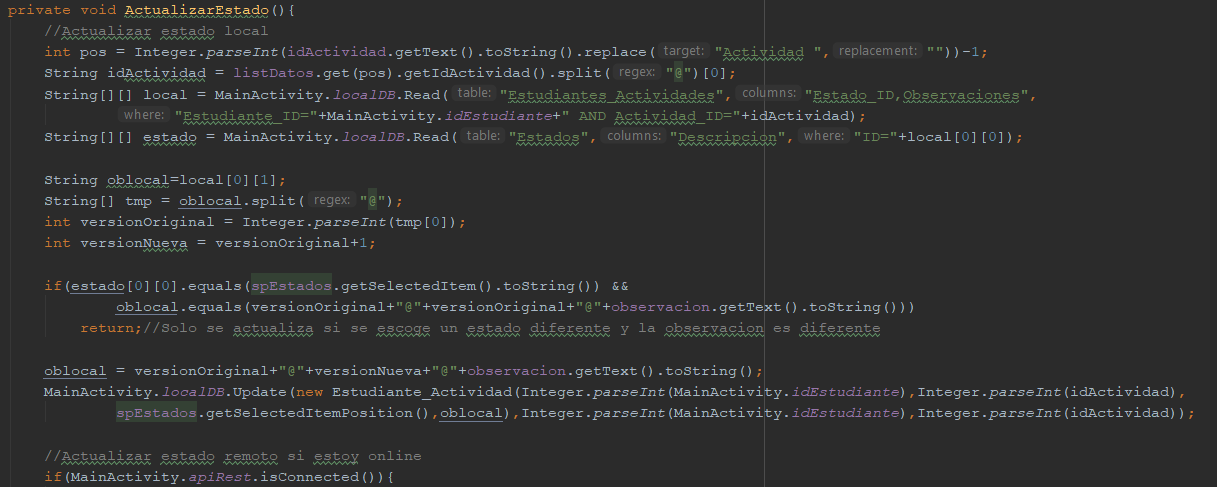


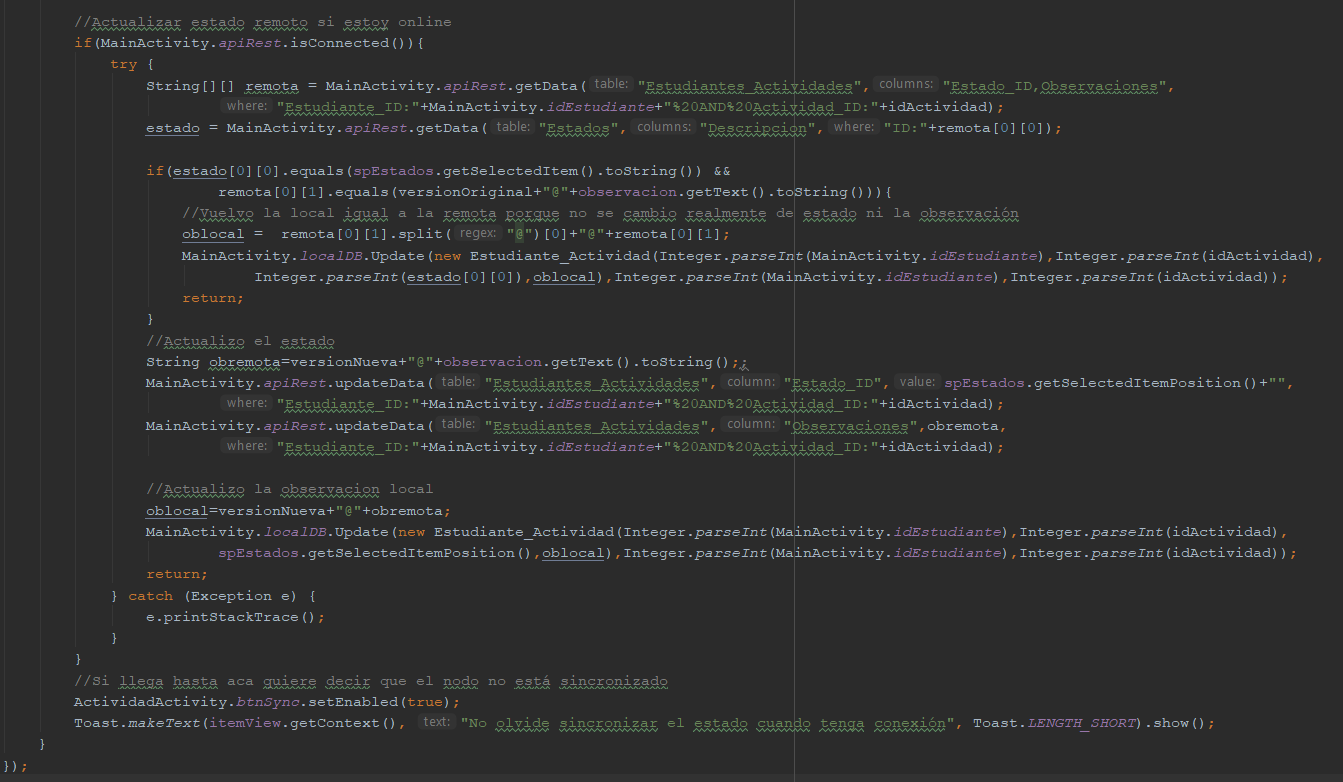


* El método de cambiar el estado se encarga de establecer la ventana de dialogo por medio de la función *alert* (que solo cuando se elige el estado *Abandonada* se muestra el cuadro de texto para ingresar la observación), también actualiza la imagen que se muestra y, lo más importante, de **actualizar el estado en la base de datos local y si esta online, realiza la actualización en el nodo**, siempre y cuando se cumpla: el nuevo estado es diferente al anterior o la observación es diferente, es decir, se actualizará si en verdad se realizó algún nuevo cambio por medio del método *ActualizarEstado*:









## 4.3) Modelo de sincronización

Al iniciar sesión se comparan las versiones locales con la versión remota, y se sincronizará de la siguiente forma:

* Si la versión nueva local es mayor que la versión remota se actualiza la remota con la local y la versión original local pasa a ser la nueva versión remota
* Si la versión nueva local es menor que la versión remota se actualiza la local con la remota, poniendo la versión nueva y original versión local iguales a la remota.
* Si la versión nueva local es igual que la versión remota pasan dos cosas:
  + Si la versión original local es menor que la versión remota se actualiza la local con la remota, debido a que en el nodo está la información real.
  + Si la versión original local es igual que la versión remota no se actualiza

**Nota:** La versión original local nunca es mayor que la versión remota, por lo que así se cubren todos los escenarios posibles.

Además, esto mismo sucede cuando se da clic en el botón Sincronizar, que solo estará activo cuando la nueva versión y la versión original son diferentes que sucede cuando se actualiza un estado y no se logra actualizar en el nodo (modo offline).

* Formato del campo Observaciones en el nodo

|  |  |
| --- | --- |
| <Versión> | <Información> |

* Formato del campo Observaciones en el dispositivo

|  |  |  |
| --- | --- | --- |
| <Versión original> | <Versión nueva> | <Información> |

El flujo es el siguiente:

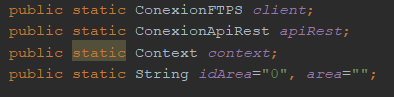
1. Al iniciar sesión se comparan las versiones para sincronizar o para habilitar el botón de sincronizar (offline)
2. Al actualizar el estado de una actividad la *versión nueva* se vuelve la *versión original* + 1, se intenta la sincronización o si no da se habilita el botón de sincronizar (offline)
3. Al dar en el botón sincronizar, se aplica el modelo de sincronización.

# 5) EU Profesores Prototipo

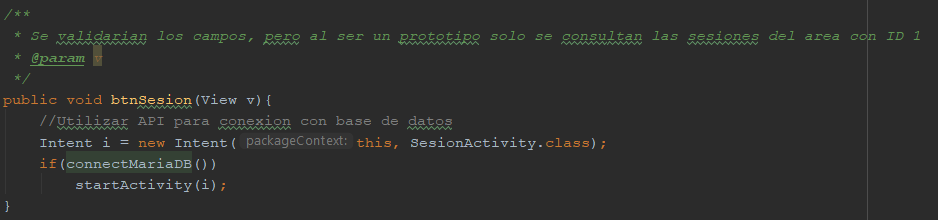
Esta aplicación es un prototipo de la app Profesores en la cual se centrará en crear sesiones sobre una misma área y siempre se crearán tres actividades, cada una con tres recursos (una imagen, un documento, un apk) en código estático (“quemado”).

## 5.1) MainActivity

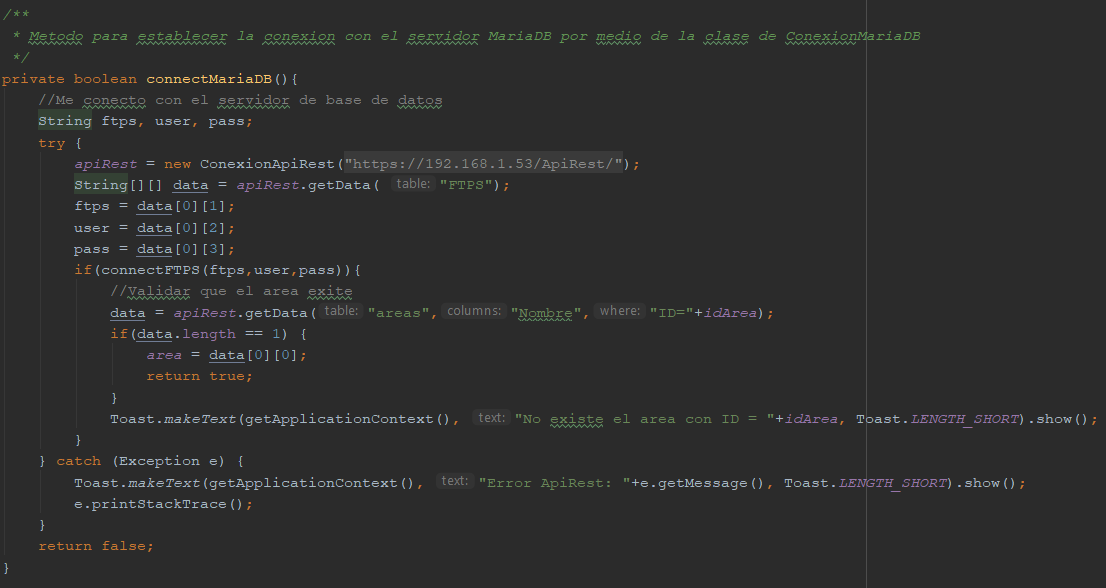
Es la ventana principal donde se debería validar el LogIn del profesor, pero al ser un prototipo simplemente se entrará con el filtro de que el ID del Area es igual a 1, por lo que en esta clase se definen los atributos que se requerirán en las demás clases:



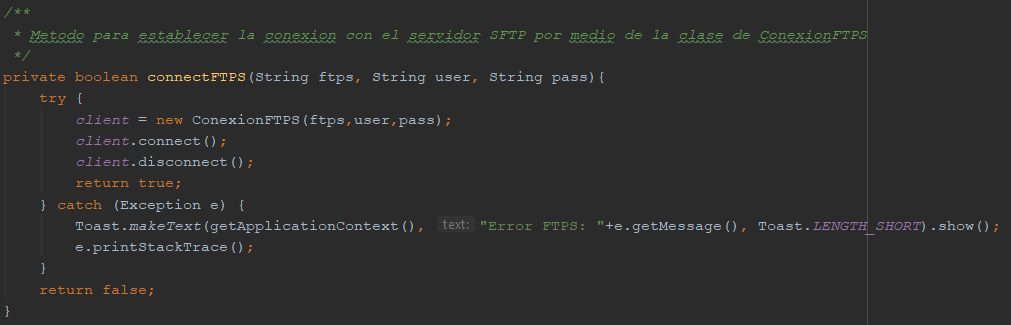
En el onCreate se inicializar el IdArea en 1 y el context. Luego está el botón para iniciar sesión:



Probará la conexión con MariaDB por medio del API REST:

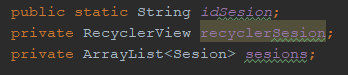


Y con los datos del servidor FTP obtenidos probará la conexión con el servidor FTP:

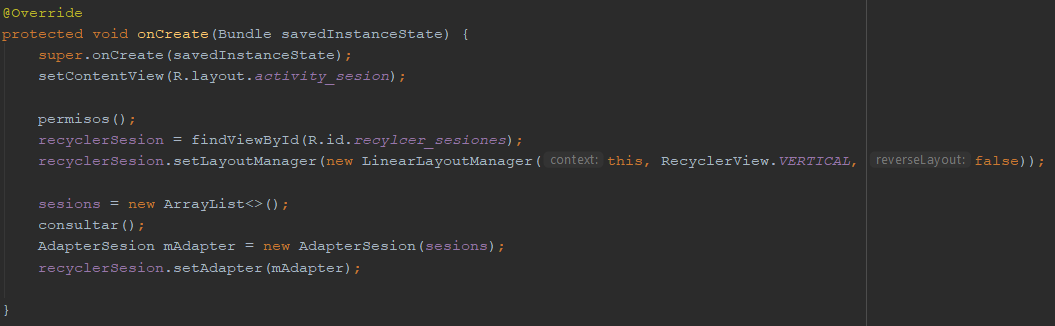


## 5.2) SesionActivity

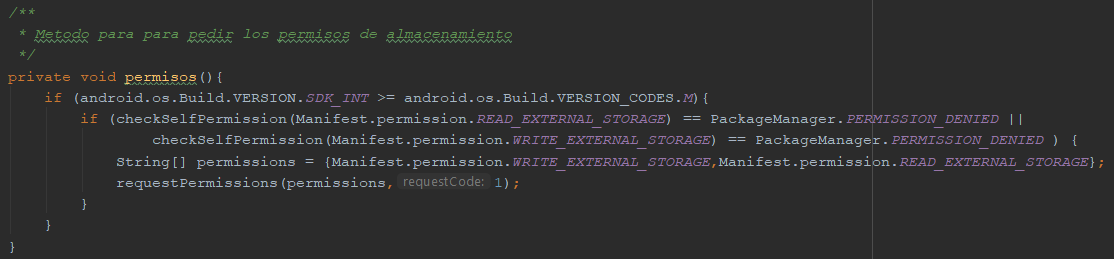
Esta clase estará encargada del manejo lógico para visualizar las sesiones que están creadas en el área cuyo ID es igual a 1, además de dar la opción de crear una sesión y de ver las actividades de una sesión especifica. Para ello primero se crean los atributos necesarios como es el RecyclerView para mostrar la lista de sesiones:



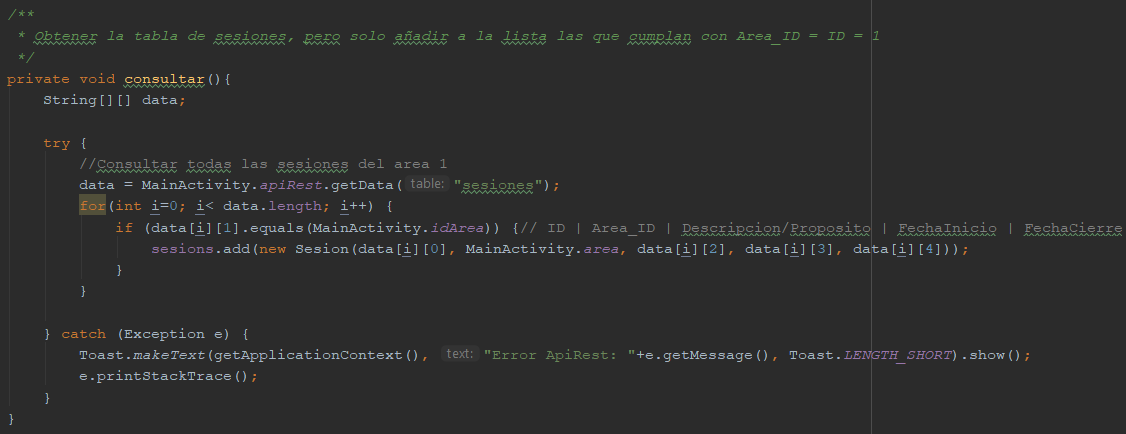
Luego en el onCreate se inicializan las variables y se piden los permisos de almacenamiento, además de consultar todas las sesiones que pertenecen al área con ID 1:



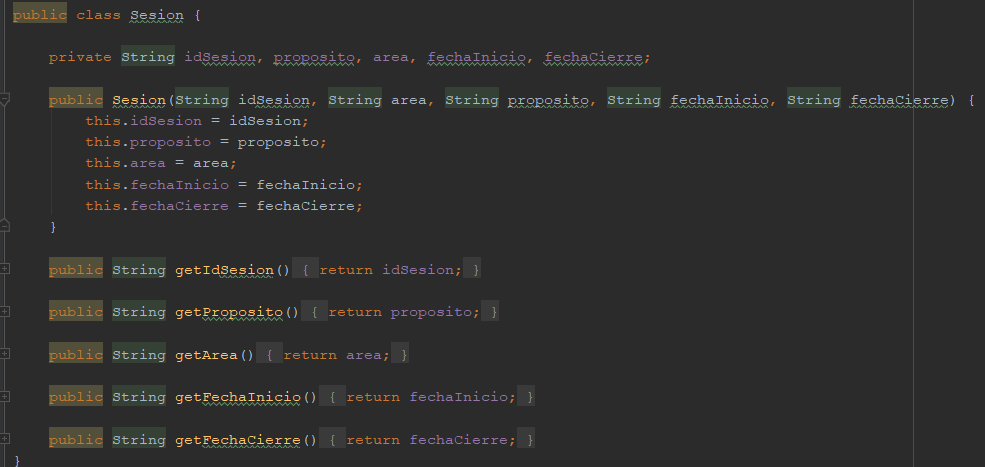
El código de los permisos es exactamente el mismo:



Y para consultar, se debe obtener toda la tabla de sesiones, pero solo añadir a la lista las que tengan la FK (la llave foránea) Area\_ID igual al ID del MainActivity, es decir, que sean igual a 1:



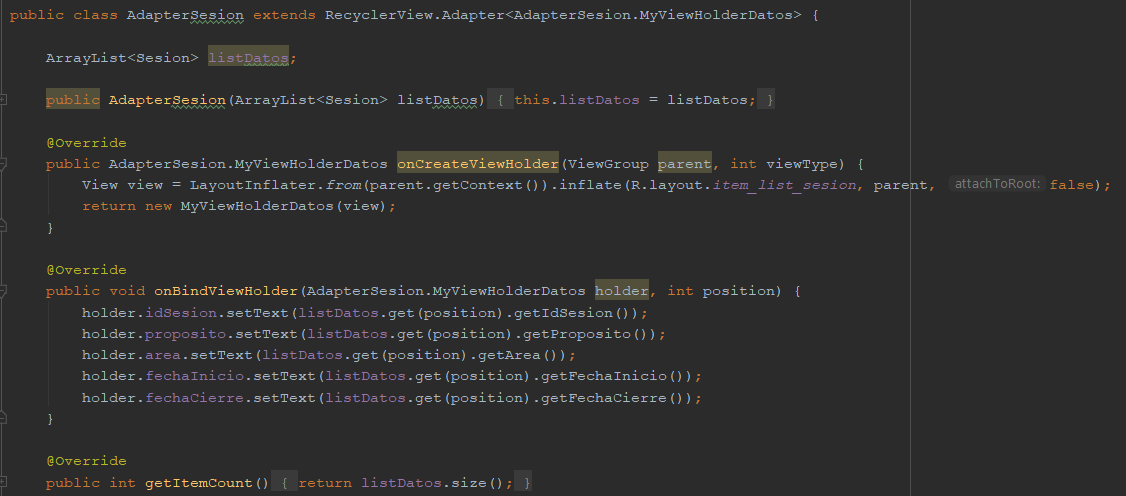
Cabe resaltar que, para esto, la lista donde se agregan los resultados es de tipo Sesion que tiene un atributo para cada columna:



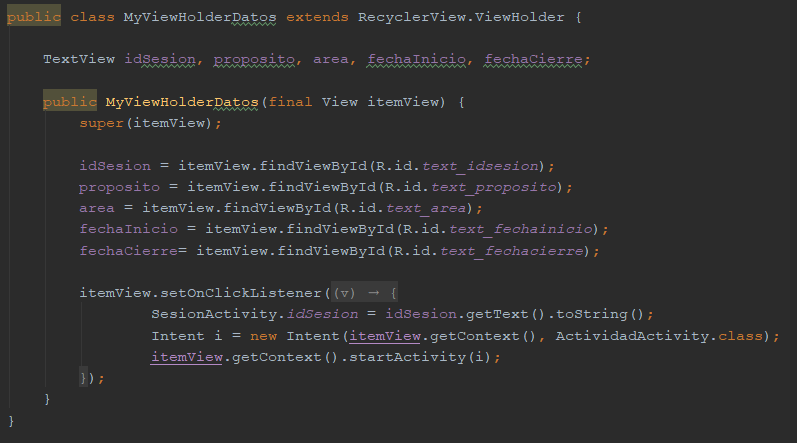
Que son los mismo que se mostrarán en el CardView ya que no se implementará ningún tipo de botón en la tarjeta:



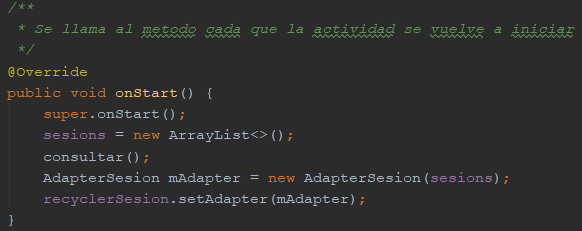
Que para poder mostrar tantas veces las tarjetas como sesiones existentes se crea la clase *AdapterSesion*que hará esto posible:



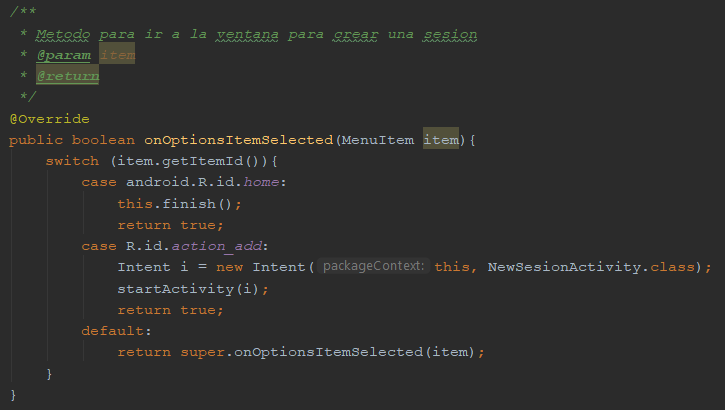
Y es en la clase *MyViewHolderDatos* donde se debe programar el código para cuando se dé clic sobre una tarjeta ir a la ventana con todas las actividades que tiene la sesión seleccionada:



Además, está el método onStart() que se llama cuando la actividad vuelve a iniciar por lo que vuelve a consultar las sesiones por si hubo algún cambio.

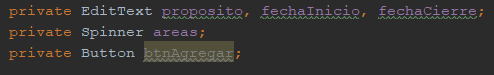


Por último, cuando se dé clic en el boton “+” de la venta de sesión se abrirá una ventana para poner los datos necesarios para crear una sesión y sus actividades con sus recursos por debajo:

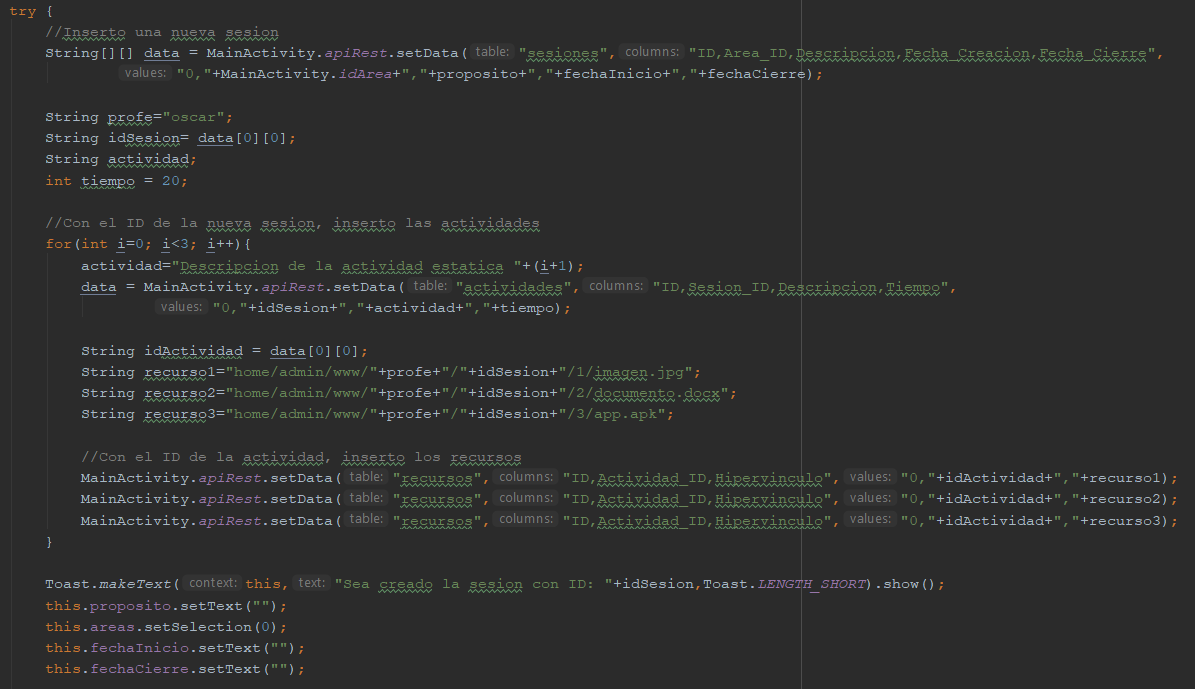


## 5.3) NewSesionActivity

En esta clase se corresponde a crear una nueva sesión en el Area\_ID 1 y que esta contenga 3 actividades que su vez, cada una contiene 3 recursos, entonces primero se identifican los campos de la sesión:



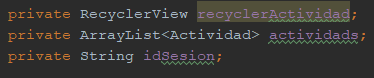
Y cuando se hace clic en el botón agregar se validan los campos y crea la sesión, se obtiene el ID con la cual quedo agregado para crear las tres actividades, que se guarda el ID de la actividad para crearle los tres recursos:



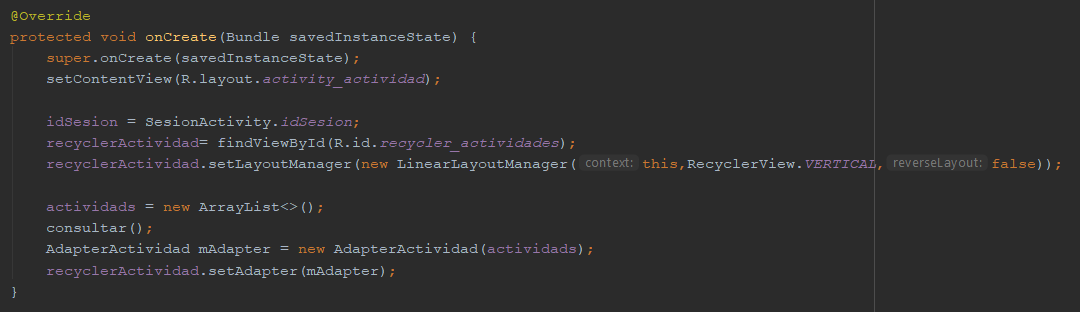
[Subir archivos al nodo FTPS]

## 5.4) ActividadActivity

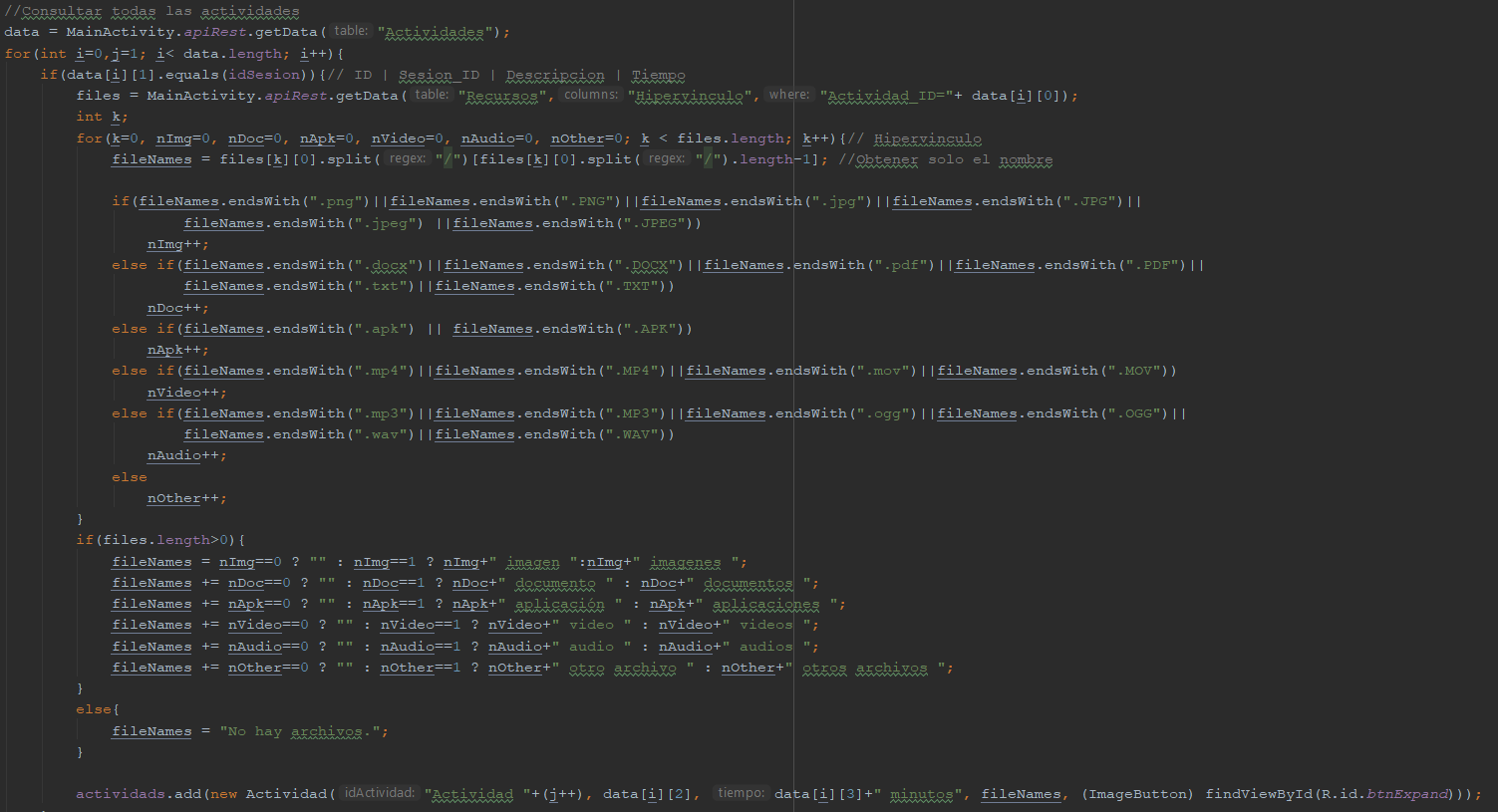
Se encarga de mostrar la lista de actividades que tiene una sesión en específico, solo se muestra la información sin botones, entonces en primer lugar se crean las variables necesarias como las listas de actividades y el RecyclerView y el idSesion que se utilizará:



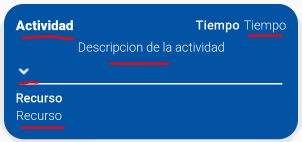
Luego en el onCreate() se hace las asignaciones e inicializaciones:



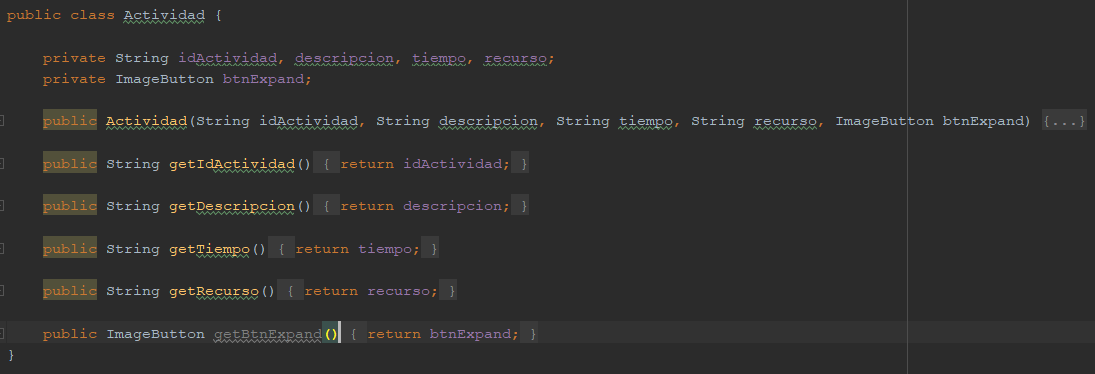
Como ya se ha hecho anteriormente, el método consultar se encargará de llenar la lista con los registros de la base de datos que cumplan con la condición de que el ID de sesión sea igual al idSesion:



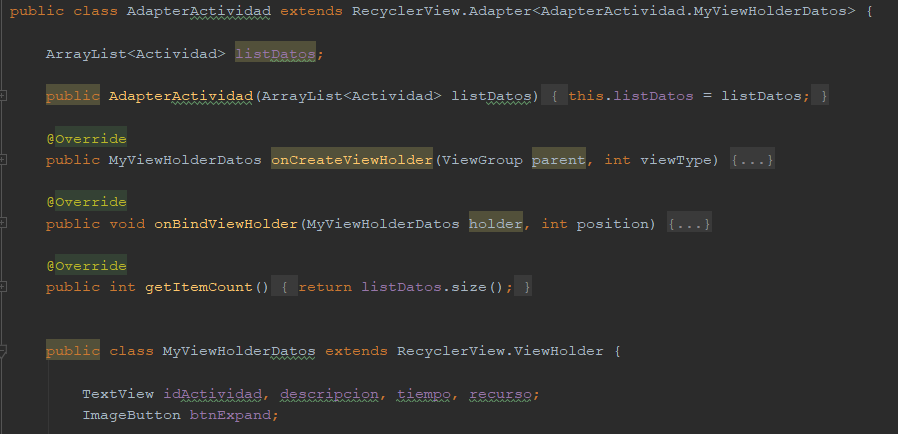
Solo se tiene un objeto demás en la lista que es el icono para expandir o colapsar la tarjeta para ver u ocultar, respectivamente, los recursos de la actividad, por lo que cumple que están todos los objetos de la CardView:



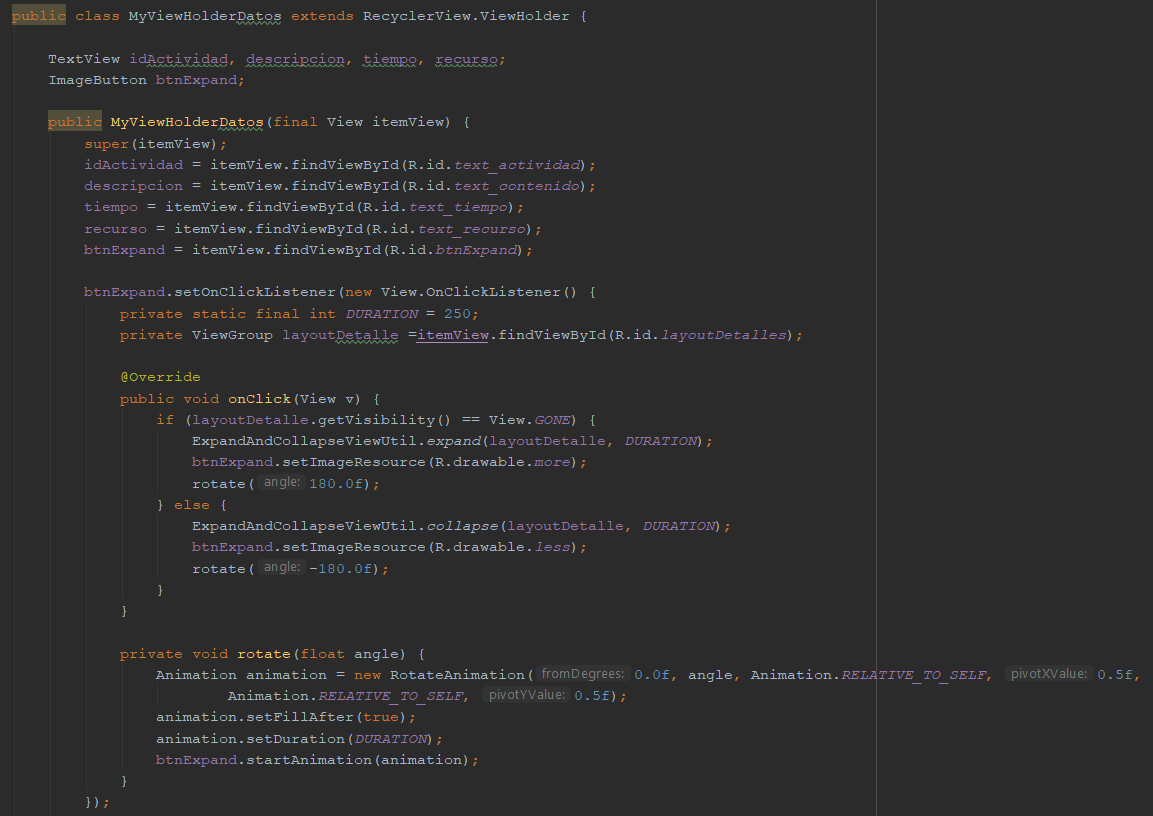
Y la clase que nos ayuda para crear una lista con todos estos objetos:



Por último, el *AdapterActividad* que nos replica la tarjeta en el RecyclerView:



Recordar que en la clase interna *MyViewHolderDatos* es donde se programa el código cuando se hace clic en la tarjeta o en algún objeto de esta:



# Referencias

Rodriguez. (2020). Automatic API REST Droidcon. Slideshare.net. Retrieved 5 May 2020, from <https://www.slideshare.net/AlejandroEsquiva/automatic-apirestdroidcon>

Diseño Android: Tarjetas con CardView. danielme.com. (2020). Retrieved 5 May 2020, from <https://danielme.com/2015/08/12/diseno-android-tarjetas-con-cardview/>

Optimizando la interfaz Android - Compound Drawables. Optimizando la interfaz Android - Compound Drawables. (2020). Retrieved 5 May 2020, from <https://elbauldelprogramador.com/optimizando-la-interfaz-android-compound-drawables/>

Diseño Android: Spinner. danielme.com. (2020). Retrieved 5 May 2020, from <https://danielme.com/2013/04/25/diseno-android-spinner/>

Cómo interpretar el ciclo de vida de una actividad. developer. (2020). Retrieved 6 May 2020, from <https://developer.android.com/guide/components/activities/activity-lifecycle?hl=es>

¿Cuál es la mejor forma de iterar un Cursor de Android?. Stackoverrun.com. (2020). Retrieved 14 May 2020, from <https://stackoverrun.com/es/q/2841081>

como habilito e inhabilito las opciones de un menú. Stack Overflow en español. (2020). Retrieved 14 May 2020, from <https://es.stackoverflow.com/questions/44790/como-habilito-e-inhabilito-las-opciones-de-un-men%C3%BA>