



**Índice**

Entorno de desarrollo Android Studio ………………………. 3

Activity ………………………………………………………….. 4

Botón atrás (onBackPressed) ………………………………………… 6

Título de la activity (setTitle) …………………………………………. 6

Menú (ActionBar) ……………………………………………………….. 7

Button …………………………………………………………... 8

Manifest ………………………………………………………… 9

Toast ……………………………………………………………. 10

Values …...……………………………………………………… 11

Strings e idiomas ………………………………………………………. 11

Theme y colors …………………………………………………………. 12

Ícono de la aplicación …………………………………………. 13

Permisos ……………………………………………………….. 14

Splash Screen …………………………………………………. 15

Ejecutar una nueva activity …………………………………… 17

Listener …………………………………………………………. 18

Lista no desplegable (**ListView**) ……………………………. 19

TabView / ViewPager …………………………………………. 22

Lista desplegable (**Spinner**) …………………………………. 25

SharedPreferences ……………………………………………. 28

Navigation view ………………………………………………… 29

Colores ………………………………………………………….. 30

Lista mutable no desplegable (**RecyclerView**) …………….. 33

Fragment ………………………………………………………... 41

Peticiones HTTP ………………………………………………..

**Entorno de Desarrollo Integrado (IDE)**

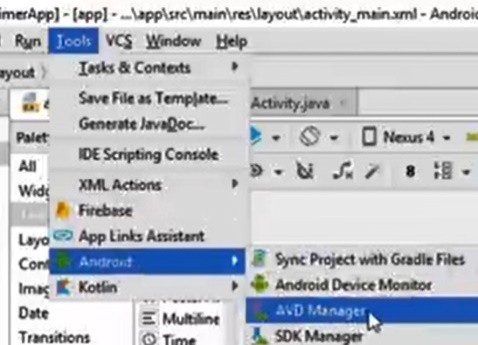
Para empezar a programar con Java para Android nativo (sistema operativo para dispositivos móviles de pantalla táctil), se comienza trabajando con AndroidStudio.

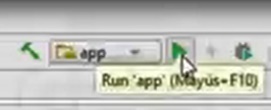
Para Windows, es recomendable tener un sistema operativo de arquitectura de 64 bits, 4 GB de RAM como mínimo (recomendado 8 GB) y un procesador Intel Core i5 o superior para virtualizar dispositivos Android en la computadora.

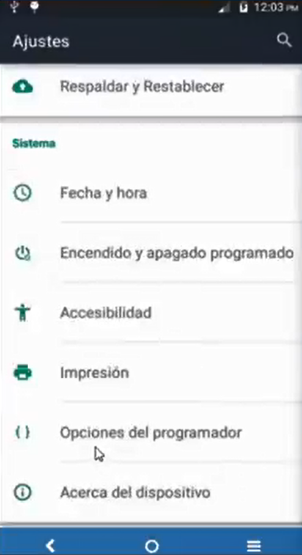
Se debe iniciar descargando el JDK de Java y Android Studio desde su página oficial.

Virtualizar dispositivo

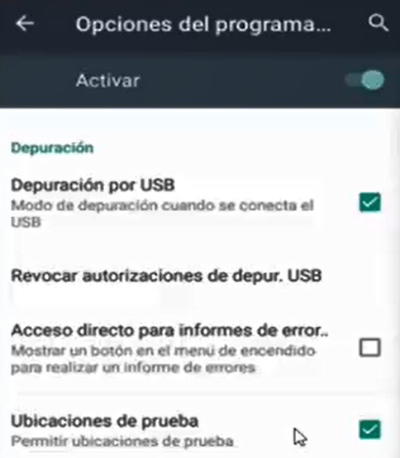
Una vez creado un proyecto en Android Studio, se puede probar su funcionalidad en un dispositivo virtual desde nuestra computadora.

Para ello se utiliza la herramienta “AVD Manager” ubicado en la pestaña “Tools”. Allí, crearemos un dispositivo virtual siguiendo las indicaciones del programa.

Luego, para iniciar el programa, se hace clic sobre el botón “run”.

Ejecutar aplicación en dispositivo físico

Si se desea probar la funcionalidad del proyecto creado en Android Studio en un dispositivo físico, primero se lo debe configurar.

Para ello, se procede a conectarlo a la computadora (la cual debe reconocerlo). Luego, se debe activar el modo programador de Android. Para ello, se debe ir a: Ajustes → Acerca del dispositivo. Una vez allí, buscar “Versión de software” o “Numero de compilación” y presionarla 7 veces. Para verificar que el modo programador esté activo, en el menú de ajustes debe visualizarse un nuevo ítem llamado “Opciones del programador”.

Dentro de “Opciones del programador”, se deben activar las opciones “Depuración por USB” y “Ubicaciones de prueba”.

Luego, el IDE podrá detectar el dispositivo físico.

**Activity**

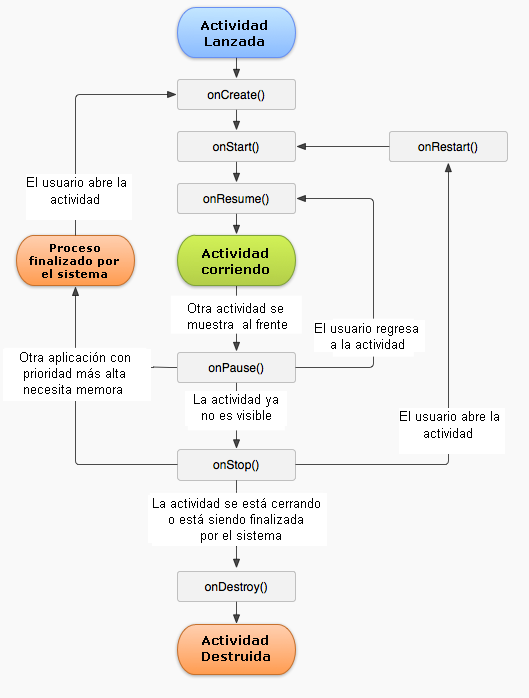
Una activity es cada pantalla de una aplicación y se encuentran conformadas por dos partes:

* **Lógica** archivo .java

Clase (class) que se crea para manipular, interactuar y colocar el código de esa activity.

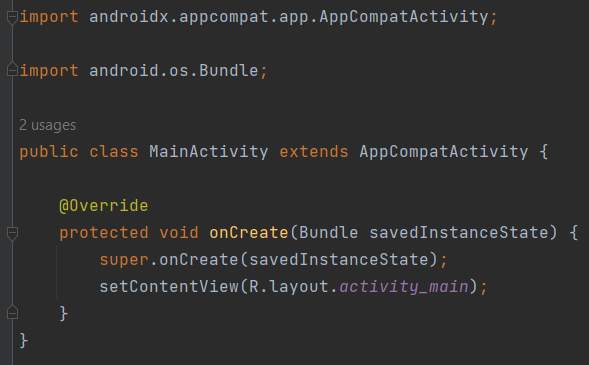
* **Gráfica** archivo XML

Contiene todos los elementos gráficos de la pantalla declaradas en etiquetas similares a las de HTML.

Además, toda activity tiene un ciclo de vida, el cual se explica en el siguiente diagrama:

Main activity

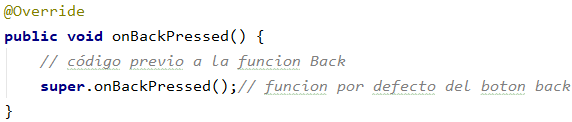
Es la activity principal del proyecto. Donde se inicia la aplicación al abrirse.

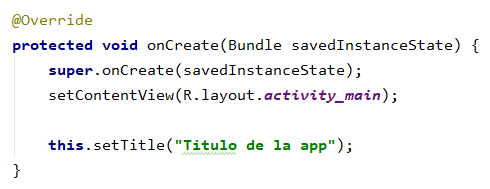


Nombre del Layout donde se encuentra el diseño visual de la activity

Botón atrás

La función **onBackPressed** permite modificar el comportamiento del botón back. Si desea desactivar la función back del botón por defecto, omita línea **super**.



Título de la activity

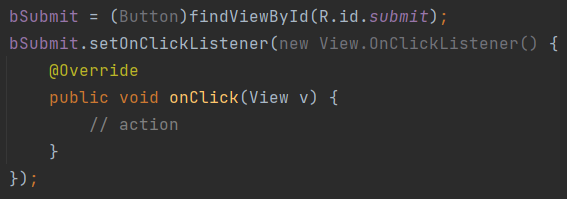
Se puede utilizar **setTitle** para definir el título de una activity. Es recomendable escribirlo en la función **onCreate**.

Menu (actionBar)

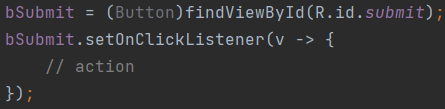
**Button**

Los button se diseñan en el xml, especificándoles un ID que será con el cual se podrá acceder desde la parte programática.





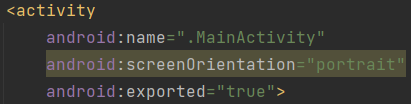
O bien, se puede utilizar la versión **lambda**:



**Manifest**

En el archivo Manifest del proyecto se encuentran especificadas todas las activities existentes. Las cuales tienen diversas configuraciones.

* Prohibir rotación de pantalla

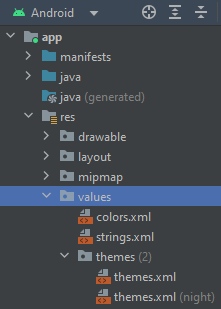


* Prohibir la ejecución de la app en una ventana flotante



**Toast**

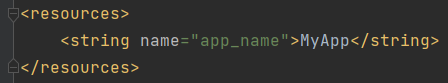
Notificación emergente para el usuario sin interrumpir las funciones de la aplicación. Puede mostrar texto y/o imágenes. Dicha notificación no es interactiva.

**Values**

Strings

En Android Studio es importante que todas las cadenas de texto predefinidas en el proyecto estén definidas dentro de la carpeta “strings.xml”.

Dentro de la etiqueta <resources> establecemos todas las cadenas de texto con su respectivo nombre.



Luego, cuando se necesite usar dicha cadena de texto dentro del proyecto, simplemente se la llama de la siguiente manera:

* En un archivo XML: @string/app\_name
* En un archivo Java: getString(R.string.app\_name);

Así mismo, se pueden usar estas cadenas de texto en diferentes idiomas.

Colors

Existen colores que se aplican por defecto en la aplicación.

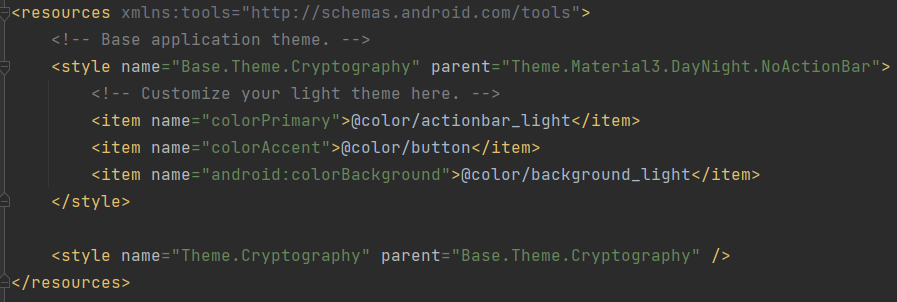
**colorPrimary** color de los botones

**colorPrimaryDark** color del ActionBar

**colorAccent**

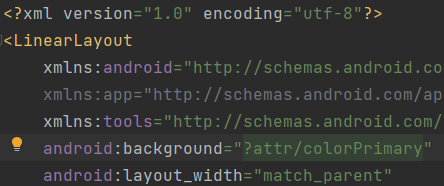
Themes

Son las temáticas de la aplicación. En general, oscuro y claro.



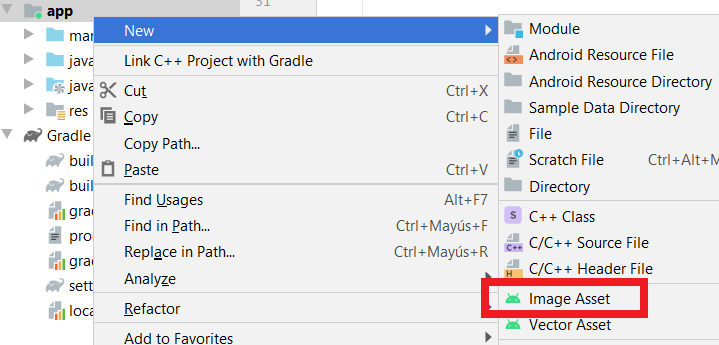
Se pueden utilizar distintos colores según el Theme.

Por ejemplo, en un fondo se puede acceder a un color específico de la siguiente forma:



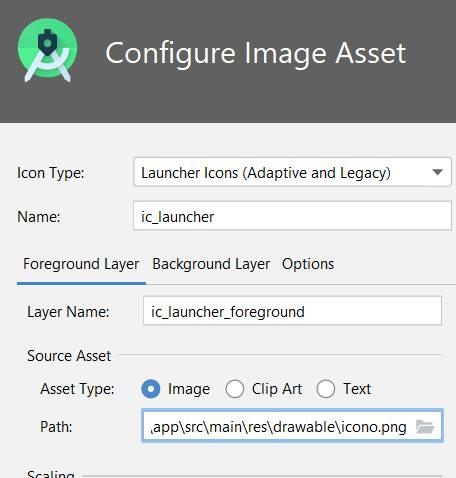
**Icono de la aplicación**

Para agregar un ícono a la aplicación, debes dirigirte a “app” y dar clic derecho. En la pestaña “New”, seleccionar “Image Asset”.



En “Icon type” seleccionar: “Launcher Icons (Adaptative and Legacy)”.

En “Path” seleccionar la ubicación de la imagen a utilizar como ícono.



Luego de configurar a gusto, seleccionar “Next” y “Listo”.

getSupportActionBar().setDisplayHomeAsUpEnabled(true);// muestra el icono de aplicacion en el ActionBar (barra superior)

getSupportActionBar().setIcon(R.mipmap.ic\_launcher);// selecciona el icono que se mostrará en el ActionBar (barra superior)

**Permisos**

Para solicitar un permiso al usuario, como el uso de la cámara o permiso para leer archivos externos de la app, se procede de la siguiente manera.

1. Declaración en el archivo manifests.xml (antes de <**application** ):

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

1. Solicitud de permiso al usuario:

**if**(ContextCompat.*checkSelfPermission*(activity.getApplicationContext(), **manifestPermission**)!= PackageManager.***PERMISSION\_GRANTED***) {  
 ActivityCompat.*requestPermissions*(activity, **new** String[]{manifestPermission}, requestCode);  
}

donde **manifestPermission** es el permiso a solicitar, con el siguiente formato:

Manifest.permission.***WRITE\_EXTERNAL\_STORAGE***

1. Respuesta del programa ante la selección del usuario, ya sea que rechace o no:



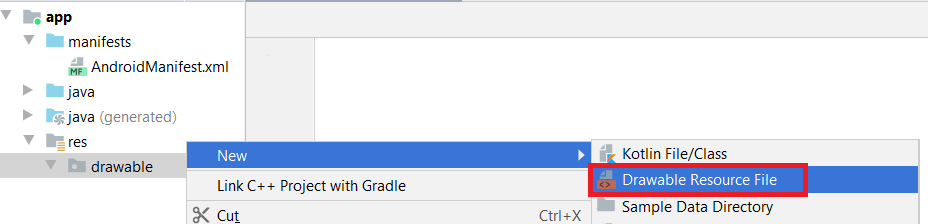
Donde MY\_REQUEST\_CODE es una constante para identificar la solicitud de permiso.

**Splash Screen**

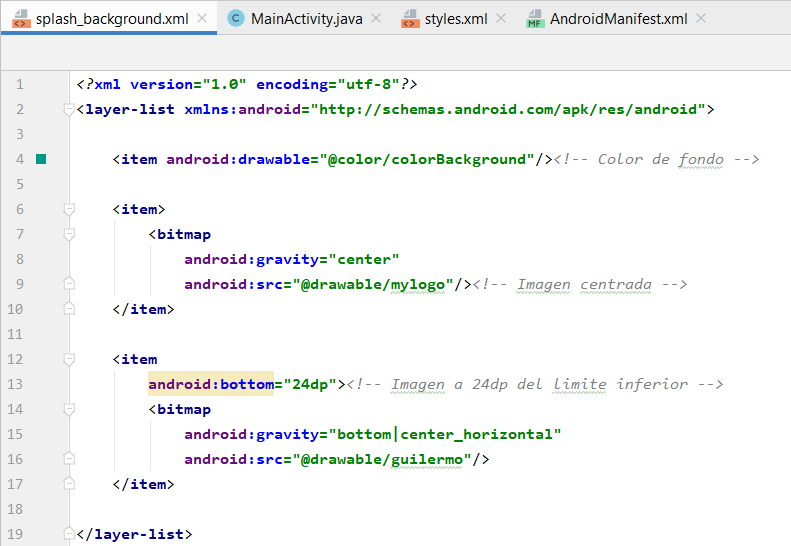
El splash screen es la imagen que se muestra al iniciar una app. Por defecto, es una pantalla completamente en blanco (o negro).

Esta imagen puede cambiarse de la siguiente manera:

1. Creamos un Drawable Resource File (xml) dentro de la carpeta res/drawable con el nombre “splash\_background.xml”.



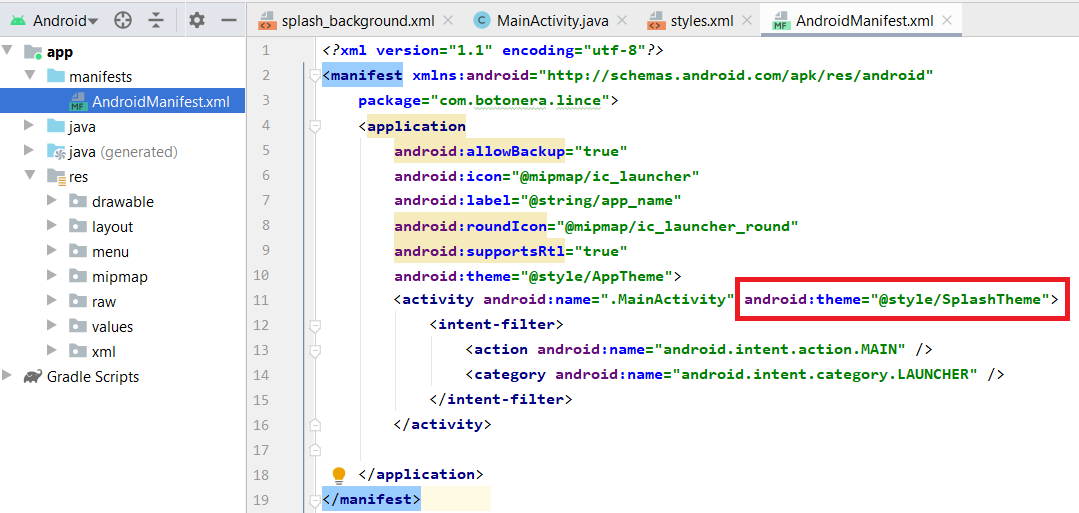
1. Comenzamos a construir nuestra imagen a mostrar en el splash screen con el siguiente formato, con todas las imágenes a utilizar previamente almacenadas dentro de la carpeta drawable:



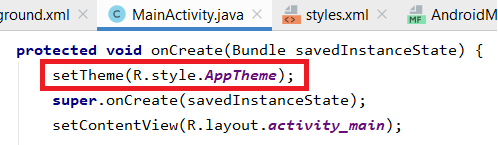
1. Nos dirigimos a la carpeta “styles.xml” ubicada dentro de res/values y agregamos un nuevo style con la imagen que acabamos de crear y el item llamado “windowBackground”:



1. Agregamos la línea “android:theme=”@style/SplashTheme” al MainActivity de nuestra aplicación en el archivo manifests.



1. Finalmente agregamos la línea “setTheme(R.style.AppTheme)” dentro de la función onCreate del archivo java de la MainActivity para que, al iniciarce la activity, inicie con el theme correspondiente.



**Iniciar una nueva Activity**

Para iniciar una nueva activity, se debe utilizar el siguiente bloque de código:



ActivityResultLauncher<Intent> mStartForResult = registerForActivityResult(  
 **new** ActivityResultContracts.StartActivityForResult(),  
 **new** ActivityResultCallback<ActivityResult>() {  
 @Override  
 **public void** onActivityResult(ActivityResult result) {  
  
 **if** (result.getResultCode() == Activity.***RESULT\_OK***) {  
 Intent data = result.getData();  
 **if** (data != **null**) **try** {  
 String text = (String) data.getExtras().get(**"TEXTO"**);  
 **int** number = (**int**) data.getExtras().get(**"NUMERO"**);  
 **boolean** bool = (**boolean**) data.getExtras().get(**"BOOLEANO"**);  
 *// Estos valores serán los recibidos de la otra activity* } **catch** (Exception ignored) {  
 }  
 }  
   
 }  
 }  
);  
  
Intent activityNueva = **new** Intent(**this**, BasicDialogActivity.**class**);  
activityNueva.putExtra(**"TEXTO"**, **"Este texto será enviado a la nueva activity"**);  
activityNueva.putExtra(**"NUMERO"**, 1234);*// numero que será enviado*activityNueva.putExtra(**"BOOLEANO"**, **false**);*// boolean que será enviado*

mStartForResult.launch(activityNueva);

Luego, en la nueva activity, se utilizará el siguiente bloque de código para recibir los datos pasados con la función putExtra

String texto = getIntent().getStringExtra(**"TEXTO"**);*// el nombre debe ser igual***int** numero = getIntent().getIntExtra(**"NUMERO"**, 0);*// el nombre debe ser igual***boolean** bool = getIntent().getBooleanExtra(**"BOOLEANO"**, **false**);*// el nombre debe ser igual*

Finalmente, para poder recibir datos de la nueva activity en la activity original, se utiliza:

Intent returnIntent = **new** Intent();  
 returnIntent.putExtra(**"TEXTO"**, unTexto);  
 returnIntent.putExtra(**"NUMERO"**, unNumero);   
 returnIntent.putExtra(**"BOOLEANO"**, unBooleano);  
 setResult(Activity.***RESULT\_OK***, returnIntent);  
 **this**.finish();*// finaliza la nueva activity*

**Listener**

Un listener es un “escuchador”, es decir, permite obtener información desde otra activity o clase en tiempo real.

Para ello, se utiliza en la clase creada el siguiente bloque de código:



**private** OnListenerResult **onListenerResult**;  
**public interface** OnListenerResult {  
 **void** funcionEnviada1(**int** unParametroEnviado);  
 **void** funcionEnviada2();  
}  
**public void** setOnListenerResult(OnListenerResult listener) {  
 **this**.**onListenerResult** = listener;  
}

Luego, dentro de esta misma clase creada, se debe utilizar la siguiente función para que el listener se active

**onListenerResult**.funcionEnviada1(1234);

Finalmente, en la activity donde se quiere recibir dicha información, se debe utilizar el listener asignado a la clase o activity creada.



**final** MainActivity mainActivity = **this**;  
OtraActivity otraActivity = **new** OtraActivity(**...**);

otraActivity.setOnDialogResult(**new** BasicDialog.OnDialogResult() {  
 @Override  
 **public void** funcionEnviada1(**int** unParametroEnviado) {  
 *// Acciones que realizará esta función cuando sea llamada*  
 *// El parámetro recibido será el mismo que se especificó antes*  
 }  
  
 @Override  
 **public void** funcionEnviada2() {  
 *// Acciones que realizará esta función cuando sea llamada*  
 }  
});

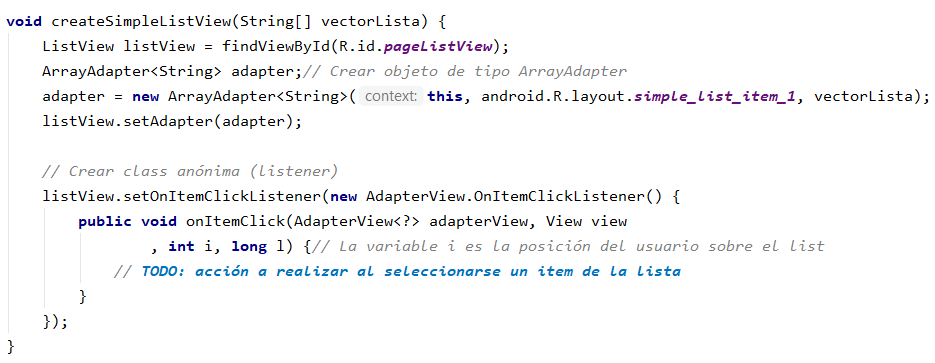
**ListView**

Para crear una lista no desplegable, es necesario iniciar creándola en el layout de la activity:

<**ListView  
 android:id="@+id/list\_view"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:listSelector="@android:color/transparent"  
 android:divider="@null"  
 android:dividerHeight="0dp"**>  
</**ListView**>

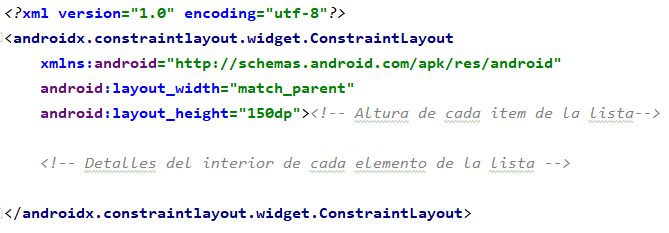
Donde **listSelector** es el color que se muestra al tocar el item de la lista, **divider** es la línea que separa cada ítem de la lista y **dividerHeight** es el tamaño de dicho separador.

A continuación, crear el listView en el archivo java de nuestro activity:



Donde vectorLista corresponden con los elementos de la lista que serán mostrados.

ListView personalizado

Para personalizar un ListView, hay que crear un ***layout resource file*** propio de la lista.

En <!-- Detalles del interior de cada elemento de la lista --> puede ir cualquier cosa, desde imágenes y textos hasta botones interactivos.

Además, es necesario una class java extendida por BaseAdapter para nuestro ListView:

**public class** ListViewPersonalizado **extends** BaseAdapter {  
  
 **private final** Context **context**;  
 **private final** String[] **elementos**;  
 **private static** LayoutInflater *inflater* = **null**;  
  
 **public** ListViewPersonalizado(Context context, String[] elementos, ...) {  
 *inflater* = (LayoutInflater)context.getSystemService(Context.***LAYOUT\_INFLATER\_SERVICE***);  
 **this**.**context** = context;  
 **this**.**elementos** = elementos;  
 }  
   
 @Override  
 **public int** getCount() { **return elementos**.**length**; } *// número de filas* @Override  
 **public** Object getItem(**int** posicion) { **return** posicion; }  
  
 @Override  
 **public long** getItemId(**int** posicion) { **return** posicion; }  
  
 @Override  
 **public** View getView(**final int** fila, View convertView, ViewGroup parent) {  
 @SuppressLint(**"ViewHolder"**) **final** View view =  
 *inflater*.inflate(R.layout.***listviewpersonalizado***, parent, **false**);  
  
 *//* ***TODO: personalización de cada fila del listView (ej: con un listener)* return** view;  
 }  
}

Donde ***listviewpersonalizado*** es el nombre del ***layout file resource*** de nuestro *listView*.

Es de mucha utilidad agregar un listener en la class:

@Override  
**public** View getView(**final int** fila, View convertView, ViewGroup parent) {  
 @SuppressLint(**"ViewHolder"**) **final** View view =  
 *inflater*.inflate(R.layout.***listviewpersonalizado***, parent, **false**);  
  
 ImageView imageView = view.findViewById(R.id.***boton\_interactivo***);  
 imageView.setOnClickListener(**new** View.OnClickListener() {  
 @Override  
 **public void** onClick(View view) {  
 **onItemClickListener**.bottonClicked(idElemento);  
 }  
 });  
  
 **return** view;  
}  
  
*// Listener***private** OnElementClickListener **onItemClickListener**;  
**public interface** OnElementClickListener {  
 **void** bottonClicked(**int** idElemento);  
}  
**public void** setOnBottonCLick(OnElementClickListener listener) {  
 **this**.**onItemClickListener** = listener;  
}

Luego, añadir el listener a la activity donde se mostrará el listView:

**private void** createPersonalizatedListView(String[] vectorLista) {  
 ListView listView = findViewById(R.id.***list\_view***);  
 ListViewPersonalizado lvp = **new** ListViewPersonalizado(**this**, vectorLista, ...);  
  
 *// Listener del ListView personalizado* lvp.setOnBottonCLick(**new** ListViewPersonalizado.OnElementClickListener() {  
 @Override  
 **public void** bottonClicked(**int** idElemento) {  
 *//* ***TODO: acciones*** }  
 });  
  
 listView.setAdapter(lvp);*// Adapter al listView de nuestro activity*}

**TabView / ViewPager**

Para crear una vista dividida por columnas, se debe crear un TabView junto con ViewPager, empezando por el layout de la activity:

<**LinearLayout   
 xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:tools="http://schemas.android.com/tools"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent" android:orientation="vertical"**>  
  
 <**com.google.android.material.tabs.TabLayout  
 android:id="@+id/tabLayout"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"**/>  
  
 <**androidx.viewpager2.widget.ViewPager2  
 android:id="@+id/viewpager"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"**/>  
  
</**LinearLayout**>

Es necesario crear una class PageAdapter para adaptar las Tabs a cada vista.

**public class** PageAdapter **extends** FragmentStateAdapter {  
 ArrayList<Fragment> **arrayList** = **new** ArrayList<>();  
   
 **public** PageAdapter(@NonNull FragmentManager fragmentManager,  
 @NonNull Lifecycle lifecycle) {  
 **super**(fragmentManager, lifecycle);  
 }  
   
 @NonNull  
 @Override  
 **public** Fragment createFragment(**int** position) {  
 **return arrayList**.get(position);  
 }  
   
 @Override  
 **public int** getItemCount() {  
 **return arrayList**.size();  
 }  
   
 **public void** addFragment(Fragment fragment) {  
 **arrayList**.add(fragment);  
 }  
}

A continuación, crear los fragments que contendrán cada Tab:

**public class** Page **extends** Fragment {  
   
 **private** Context **context**;  
   
 **public** Page(Context context, ...) {  
 **this**.**context** = context;  
 }  
   
 @Override  
 **public void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 }  
   
 @Override  
 **public void** onViewCreated(@NonNull View view,

@Nullable Bundle savedInstanceState) {  
 **super**.onViewCreated(view, savedInstanceState);  
 *// Inicializar elementos del layout* }  
   
 @Override  
 **public** View onCreateView(LayoutInflater inflater,   
 ViewGroup container,  
 Bundle savedInstanceState) {  
 **return** inflater.inflate(R.layout.**fragment\_page**, container, **false**);  
 }  
}

Será necesario crear un layout para cada fragment con el contenido que será mostrado en cada page.

Por último, añadir todo a la activity:

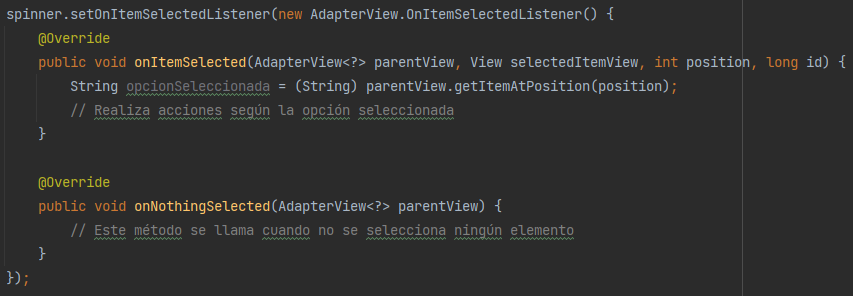
*// Atributos globales***private** ArrayList<Fragment> **page** = **new** ArrayList<>();  
**private** ViewPager2 **viewPager**;  
  
*// Método para mostrar pages***private void** createPageView() {*// in onCreate  
  
 // PageAdapter para ViewPager* PageAdapter pageAdapter = **new** PageAdapter(getSupportFragmentManager(), getLifecycle());  
  
 *// Fragments del PageAdapter del ViewPager* **page**.add(**new** Page(**this**, ...)); pageAdapter.addFragment(**page**.get(**page**.size() - 1)); *// #0* **page**.add(**new** Page(**this**, ...)); pageAdapter.addFragment(**page**.get(**page**.size() - 1)); *// #1* **page**.add(**new** Page(**this**, ...)); pageAdapter.addFragment(**page**.get(**page**.size() - 1)); *// #2  
  
 // ViewPager2* **viewPager** = findViewById(R.id.***viewpager***);  
 **viewPager**.setAdapter(pageAdapter);  
  
 *// TabLayout* TabLayout tabLayout = findViewById(R.id.***tabLayout***);  
  
 **new** TabLayoutMediator(  
 tabLayout, **viewPager**, **new** TabLayoutMediator.TabConfigurationStrategy() {  
 @Override  
 **public void** onConfigureTab(@NonNull TabLayout.Tab tab, **int** position) {  
 **switch**(position) {*// Crear ícono: File -> New -> ImageAsset (Action Bar and Tab Icons)* **case** 0: { tab.setIcon(R.drawable.***ic\_user0***); tab.setText(**"User0"**); **break**; } *// #0* **case** 1: { tab.setIcon(R.drawable.***ic\_user2***); tab.setText(**"User1"**); **break**; } *// #1* **case** 2: { tab.setIcon(R.drawable.***ic\_user3***); tab.setText(**"User2"**); **break**; } *// #2* }  
 }  
 }).attach();  
}

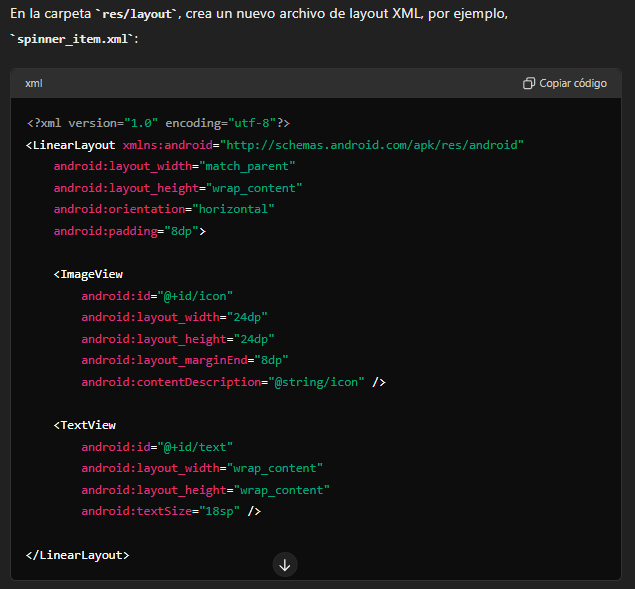
**Lista desplegable (Spinner)**

Agregar el Spinner en el Layout.xml (por ejemplo, en MainActivity.xml). Luego, se procede a asignar los elementos de la lista con un ArrayAdapter:

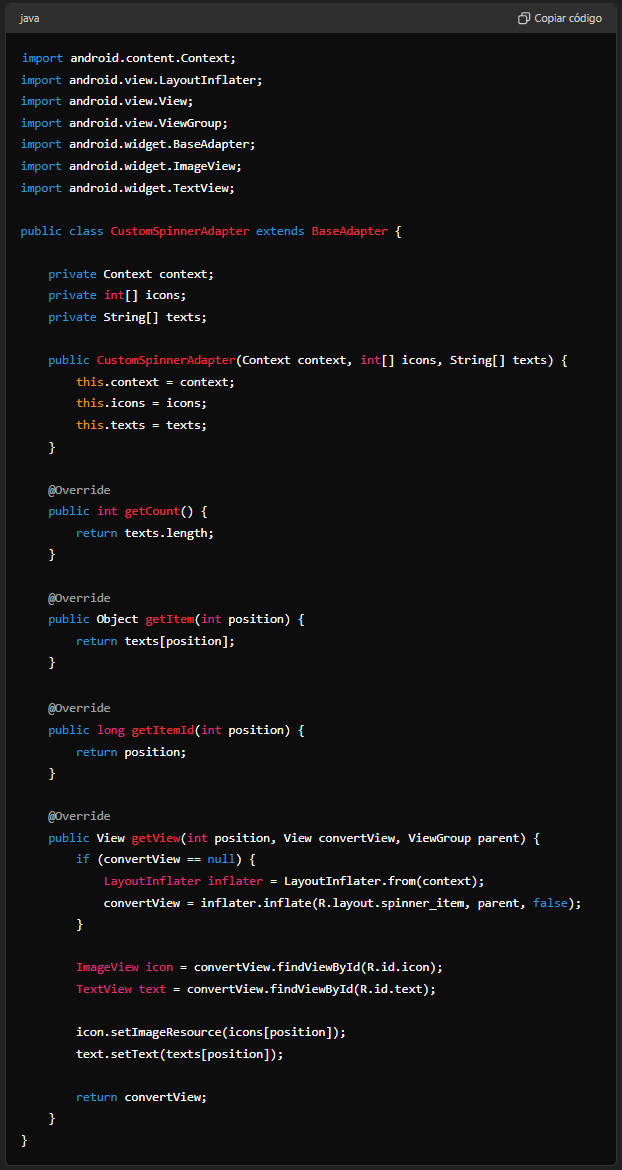
Spinner spinner = findViewById(R.id.*spinner*);  
String[] opciones = {"Opción 1", "Opción 2", "Opción 3"};  
ArrayAdapter<String> adapter = new ArrayAdapter<>(this, android.R.layout.*simple\_spinner\_item*, opciones);  
adapter.setDropDownViewResource(android.R.layout.*simple\_spinner\_dropdown\_item*);  
spinner.setAdapter(adapter);

Y se pueden añadir listeners que detecten cuándo el spinner fue seleccionado:

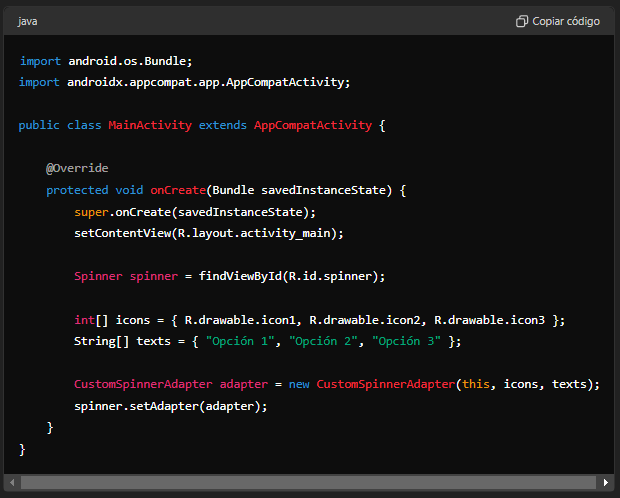


Si se desea que los ítems del spinner tengan un ícono (o cualquier otra personalización), se debe crear un Layout que represente cada ítem del spinner.

Luego, crear una class que herede de BaseAdapter

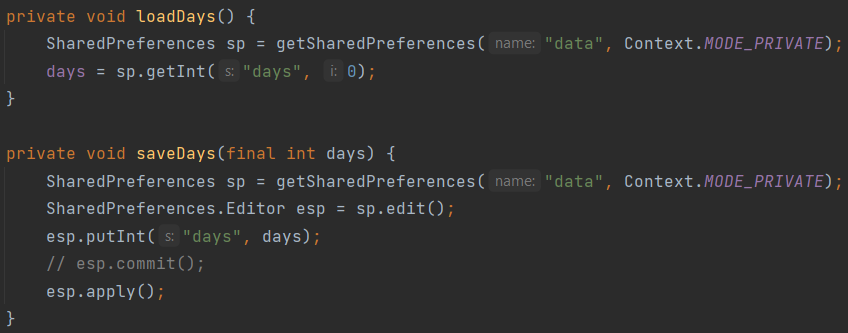


Finalmente, en la class donde se quiera usar el Spinner (por ejemplo, MainActivity) se procede a agregarlo.



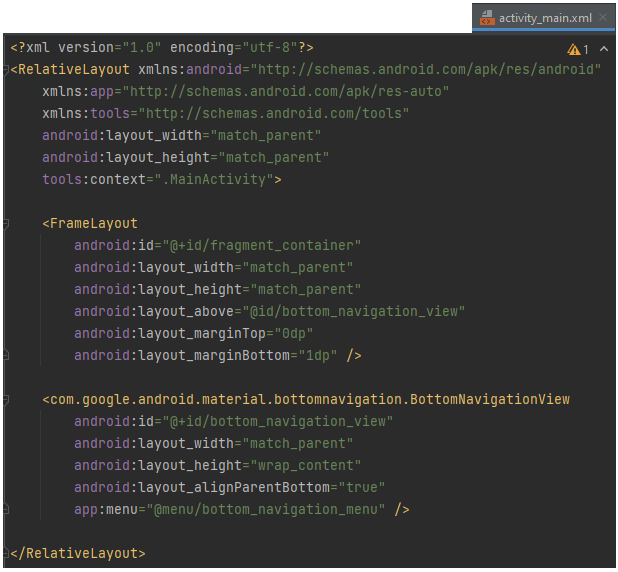
**SharedPreferences**

Pequeña cantidad de información que se guarda a modo de configuración.



**Navigation view**

Permite la creación de views para mostrarse sin necesidad de cambiar de Activity (similar a ViewPager).



**Colores**

**Colores Primarios**

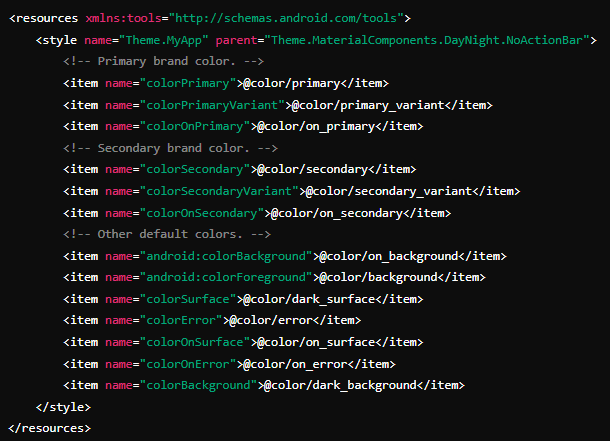
1. **colorPrimary**:
   * Este es el color principal de tu aplicación. Se utiliza para elementos prominentes como la barra de herramientas (toolbar) y el color de fondo de algunos componentes.
2. **colorPrimaryVariant**:
   * Este es una variante más oscura o más clara del color principal. Se usa para proporcionar un poco de contraste con el color principal. Se puede ver en la barra de estado y otros componentes que necesitan una ligera variación del color principal.
3. **colorOnPrimary**:
   * Este color se utiliza para el contenido que aparece encima del color primario. Por ejemplo, el texto en una barra de herramientas que tiene el color colorPrimary como fondo debe usar colorOnPrimary para garantizar una buena legibilidad.

**Colores Secundarios**

1. **colorSecondary**:
   * Este es el color secundario de tu aplicación. Se utiliza para elementos de acción como botones, íconos y componentes interactivos.
2. **colorSecondaryVariant**:
   * Similar al colorPrimaryVariant, este es una variante del color secundario que proporciona contraste adicional. Se puede usar en elementos donde se necesite una variación del color secundario.
3. **colorOnSecondary**:
   * Este color se utiliza para el contenido que aparece encima del color secundario. Por ejemplo, el texto en un botón que tiene el color colorSecondary como fondo debe usar colorOnSecondary.

**Otros Colores Predeterminados**

1. **android:colorBackground**:
   * Este es el color de fondo principal de tu aplicación. Se usa para la mayoría de las superficies de la aplicación, como el fondo de las actividades y fragmentos.
2. **android:colorForeground**:
   * Este color se utiliza para el contenido de primer plano, como texto y gráficos. Normalmente, se establece en contraste con el color de fondo para garantizar una buena legibilidad.
3. **colorSurface**:
   * Este es el color de las superficies de los componentes, como tarjetas (cards), hojas (sheets), y menús. Se usa para las superficies elevadas que están encima del fondo principal.
4. **colorError**:
   * Este color se utiliza para indicar errores. Se usa en componentes como mensajes de error, íconos de advertencia y borde de campos de texto que tienen un error.
5. **colorOnSurface**:
   * Este color se utiliza para el contenido que aparece encima de las superficies. Por ejemplo, el texto en una tarjeta (card) debe usar colorOnSurface para garantizar la legibilidad.
6. **colorOnError**:
   * Este color se utiliza para el contenido que aparece encima del color de error. Por ejemplo, el texto en un mensaje de error debe usar colorOnError para garantizar la legibilidad.
7. **colorBackground**:
   * Similar a android:colorBackground, este es el color de fondo principal de la aplicación. En el tema claro, puede ser un color claro, y en el tema oscuro, puede ser un color oscuro.



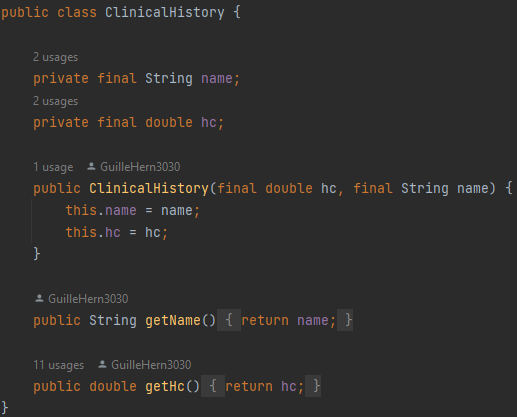


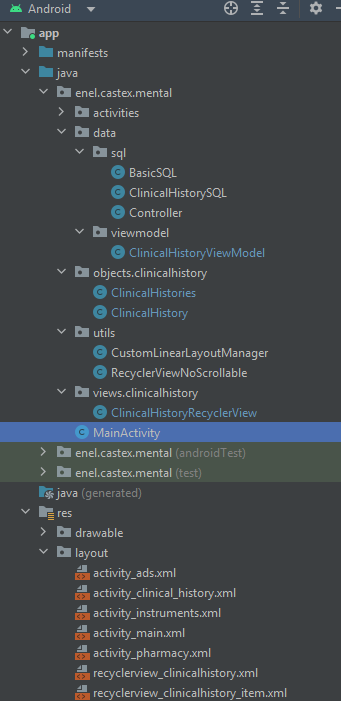
**RecyclerView**

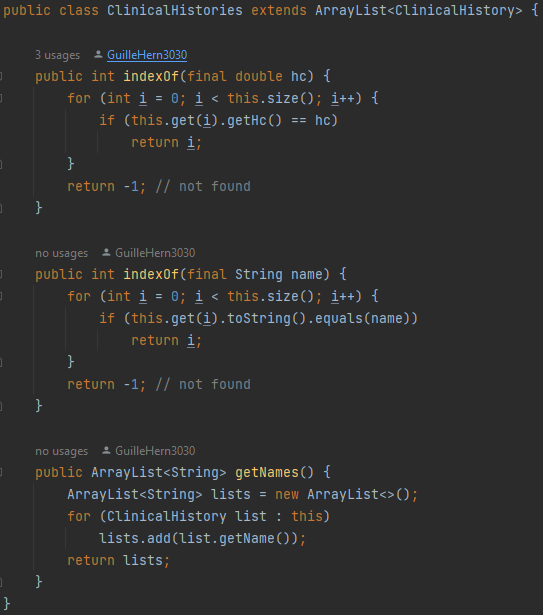
Se pueden crear listas no desplegables mutables, es decir, que se puedan modificar en tiempo real (LiveData) sin detener la ejecución de la aplicación.

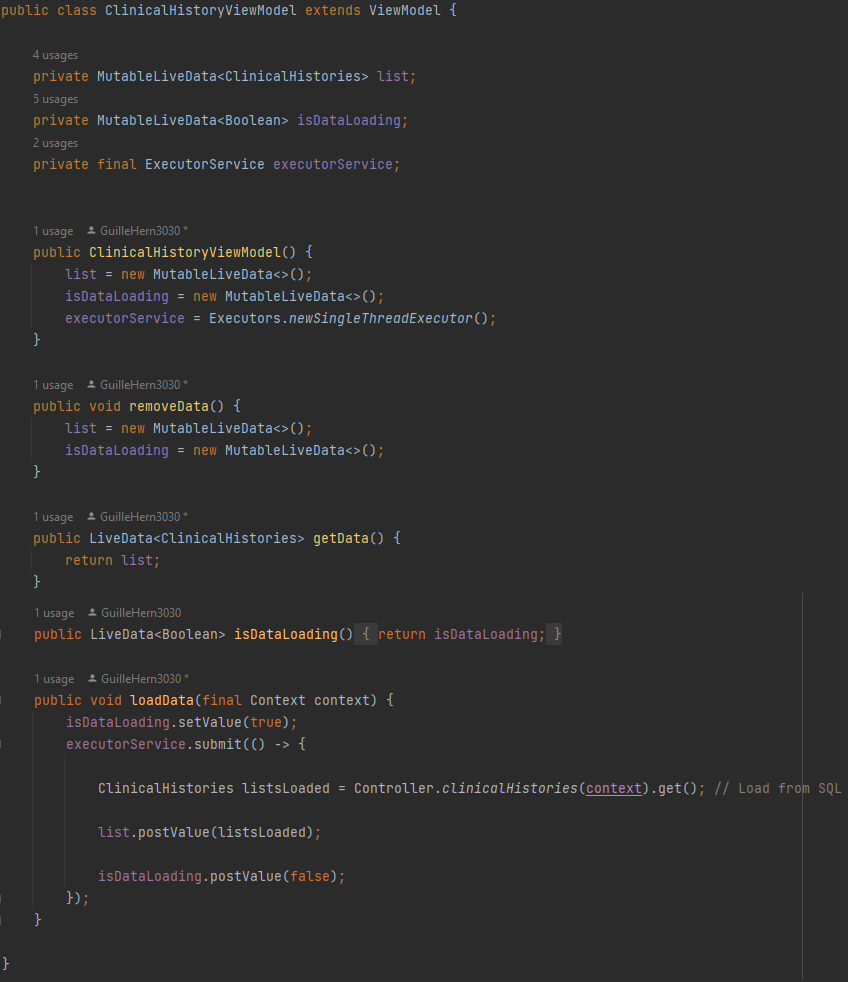
Para este ejemplo, usaremos una lista donde se muestren historias clínicas, que se puedan eliminar y agregar en tiempo real con una Base de Datos (SQL).

* + 1. Crear el objeto “Historias clínicas” y su Array

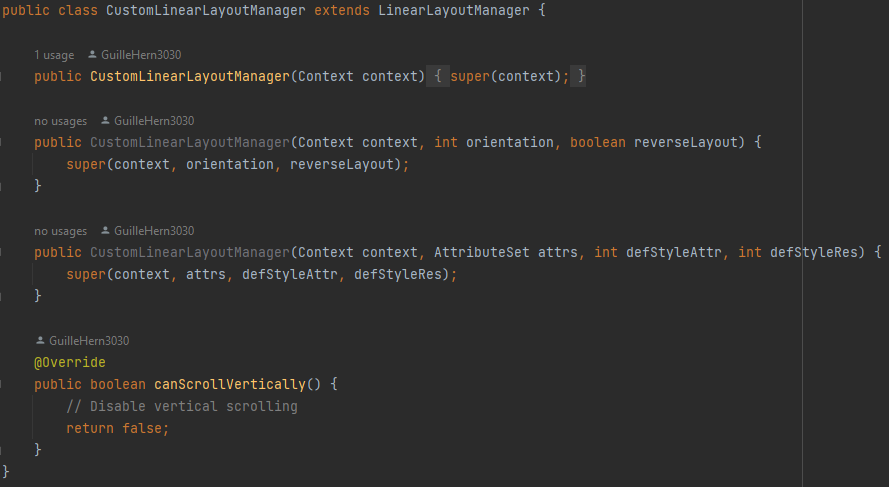


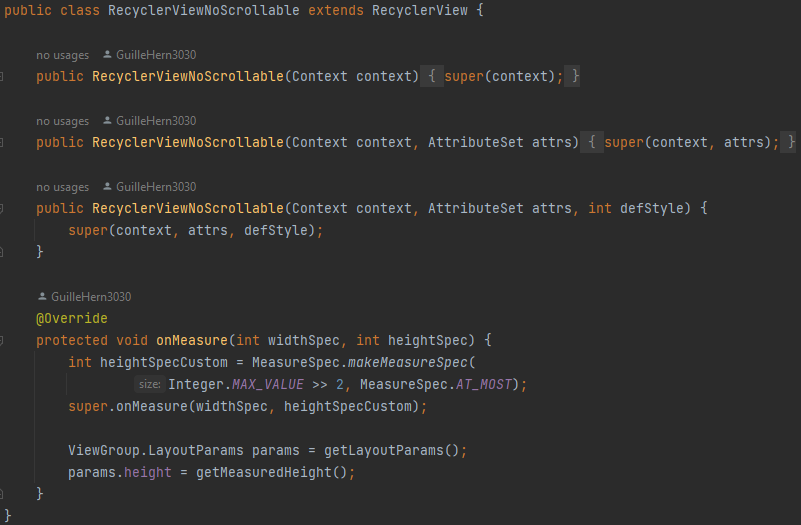


* + 1. Crear el ArrayList que maneje ese objeto
    2. Crear el controlador de la Base de Datos (en este caso se usó BasicSQL)
    3. Crear el ViewModel que permitirá cargar los datos en segundo plano

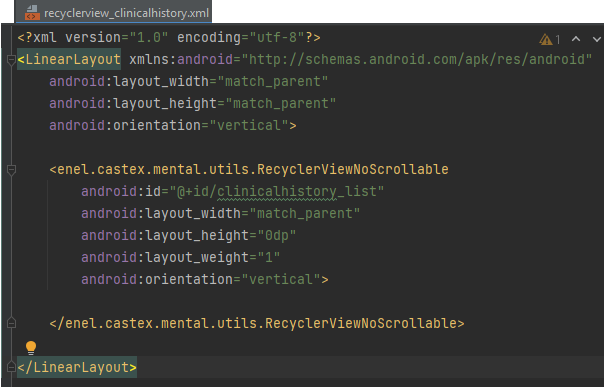


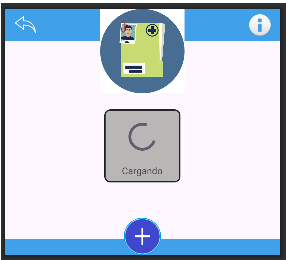
* + 1. Crear los elementos útiles que permitirán gestionar mejor los archivos





* + 1. Crear los Layout





* + 1. Crear la class del RecyclerView que permitirá gestionar la lista mutable

public class ClinicalHistoryRecyclerView extends RecyclerView.Adapter<ClinicalHistoryRecyclerView.ViewHolder> {  
  
 private final Activity context;  
 private final FrameLayout parent;  
 private final RecyclerViewNoScrollable recyclerView;  
  
 private ClinicalHistories listArray;  
  
 public ClinicalHistoryRecyclerView(final Activity context, final FrameLayout parent) {  
 this.context = context;  
 this.parent = parent;  
  
 final View view = LayoutInflater.*from*(context)  
 .inflate(R.layout.*recyclerview\_clinicalhistory*, parent, false);  
 parent.addView(view);  
  
 recyclerView = view.findViewById(R.id.*clinicalhistory\_list*);  
  
 // Listeners  
 this.setOnItemClickListener(hc -> onItemClickListener.onItemClicked(hc));  
 this.setOnItemLongClickListener(hc -> onItemLongClickListener.onItemLongClicked(hc));  
 }  
  
 */\*\*  
 \* Initializes the mutable list with an ArrayList  
 \** ***@param*** *listArray ArrayList to show  
 \*/* public void initialize(final ClinicalHistories listArray) {  
 if (recyclerView != null) {  
 this.listArray = listArray != null ? listArray : new ClinicalHistories();  
 recyclerView.removeAllViews();  
 recyclerView.setLayoutManager(new CustomLinearLayoutManager(context));  
 recyclerView.setAdapter(this);  
 } else Log.*e*("InitializeError", "RecyclerView is null");  
 }  
  
 // Ítems de la lista (recyclerview\_clinicalhistory\_item.xml)  
 public static class ViewHolder extends RecyclerView.ViewHolder {  
 TextView tvName, tvClinicalHistoryNumber;  
 LinearLayout frame;  
  
 public ViewHolder(View view) {  
 super(view);  
 tvName = view.findViewById(R.id.*hc\_name*);  
 tvClinicalHistoryNumber = view.findViewById(R.id.*hc\_description*);  
 frame = view.findViewById(R.id.*hc\_frame*);  
 }  
 }  
  
 @NonNull  
 @Override  
 public ViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {  
 View view = LayoutInflater.*from*(parent.getContext())  
 .inflate(R.layout.*recyclerview\_clinicalhistory\_item*, parent, false);  
 return new ViewHolder(view);  
 }  
  
 @Override  
 public int getItemCount() {  
 return listArray.size();  
 }  
  
 @Override  
 public void onBindViewHolder(@NonNull ViewHolder holder, int row) {  
 final ClinicalHistory item = listArray.get(row);  
 holder.frame.setOnClickListener(v -> onItemClickListener.onItemClicked(item)); // listener  
 holder.frame.setOnLongClickListener(v -> {  
 onItemLongClickListener.onItemLongClicked(item);  
 return true;  
 });  
 holder.tvClinicalHistoryNumber.setText(item.hc());  
 holder.tvName.setText(item.getName());  
  
 }  
  
 // Editors  
 public void add(ClinicalHistory item) {  
 if (item != null) {  
 final int index = listArray.indexOf(item.getHc());  
 if (index >= 0 && index < listArray.size()) { // hc exists  
 listArray.set(index, item);  
 //Controller.*clinicalHistories*(context).edit(item.getHc(), item);  
 notifyItemChanged(index);  
 } else {  
 listArray.add(item);  
 //Controller.*clinicalHistories*(context).add(item);  
 notifyItemInserted(listArray.size());  
 }  
 }  
 }  
  
 public void set(final ClinicalHistory oldItem, final ClinicalHistory newItem) {  
 if (oldItem != null && newItem != null) {  
 final int index = listArray.indexOf(oldItem.getHc());  
 if (oldItem.getHc() != newItem.getHc()) {  
 if (index >= 0 && index < listArray.size()) {  
 listArray.set(index, newItem);  
 //Controller.*clinicalHistories*(context).edit(oldItem.getHc(), newItem);  
 notifyItemChanged(index);  
 }  
 }  
 }  
 }  
  
 public void remove(ClinicalHistory item) {  
 if (item != null) {  
 final int index = listArray.indexOf(item.getHc());  
 if (index >= 0 && index < listArray.size()) {  
 listArray.remove(index);  
 //Controller.*clinicalHistories*(context).delete(item);  
 notifyItemRemoved(index);  
 }  
 }  
 }  
  
 public ClinicalHistories getAll() {  
 return listArray;  
 }  
  
 public int size() {  
 return listArray.size();  
 }  
  
 public void setVisibility(final int v) {  
 parent.setVisibility(v);  
 }  
  
 // Listener definition  
 private onClinicalHistoryClickListener onItemClickListener;  
 public interface onClinicalHistoryClickListener {  
 void onItemClicked(final ClinicalHistory hc);  
 }  
 public void setOnItemClickListener(onClinicalHistoryClickListener listener) {  
 this.onItemClickListener = listener;  
 }  
  
 // Listener definition  
 private onClinicalHistoryLongClickListener onItemLongClickListener;  
 public interface onClinicalHistoryLongClickListener {  
 void onItemLongClicked(final ClinicalHistory hc);  
 }  
 public void setOnItemLongClickListener(onClinicalHistoryLongClickListener listener) {  
 this.onItemLongClickListener = listener;  
 }  
  
}

* + 1. Crear el Activity o Fragment donde se mostrará

public class ClinicalHistoryActivity extends AppCompatActivity {  
  
 private ClinicalHistoryRecyclerView clinicalHistoriesRecyclerView;  
 private TextView listEmptyView;  
 private View loadingView;  
  
 @Override  
 protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.*activity\_clinical\_history*);   
  
 //requireView(). // in fragments  
 listEmptyView = findViewById(R.id.*tv\_list\_empty*);  
 loadingView = findViewById(R.id.*loading\_view*);  
  
 // context: this = requireActivity() on Fragments  
 clinicalHistoriesRecyclerView = new ClinicalHistoriesRecyclerView(this, findViewById(R.id.*recyclerview\_container*));  
  
 clinicalHistoriesRecyclerView.setOnItemClickListener(hc -> {  
 // TODO: Qué sucederá al hacer click en un item de la lista  
 });  
  
 clinicalHistoriesRecyclerView.setOnItemLongClickListener(hc -> {  
 // TODO: Qué sucederá al hacer click prolongado en un item de la lista

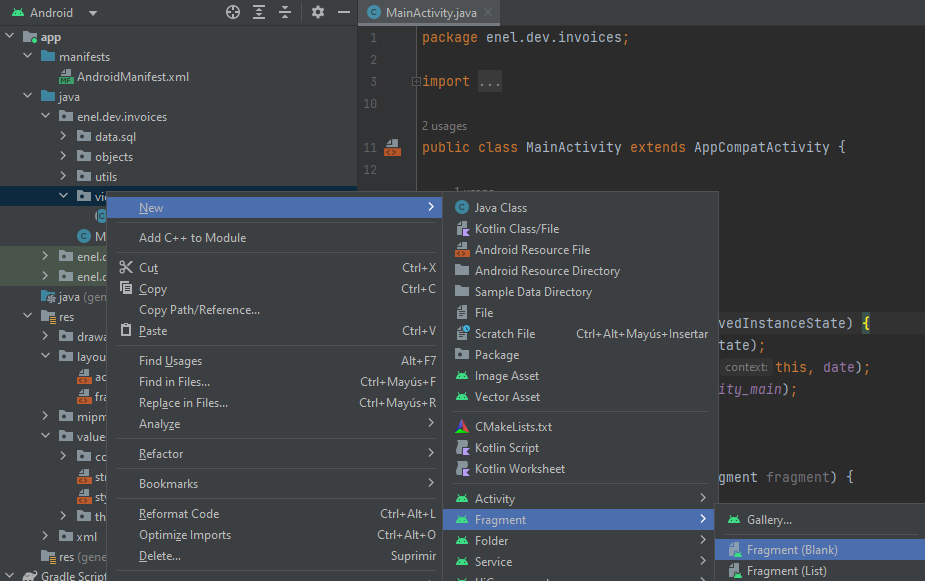
onRemoveClinicalHistory(hc) // Proveniente de un listener

});  
  
 findViewById(R.id.*bBack*).setOnClickListener(v -> finish());

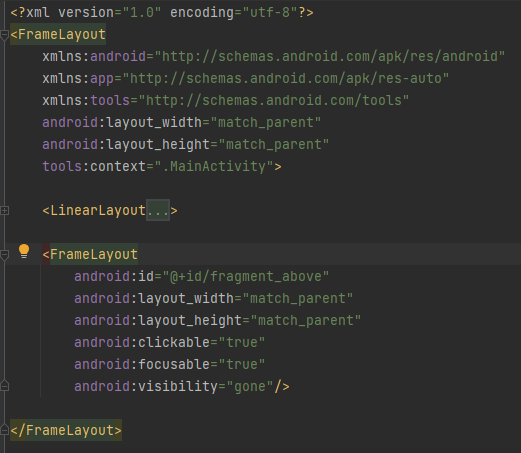
findViewById(R.id.*floatting\_button*).setOnClickListener(v -> {  
 // TODO: Qué sucederá al hacer click en el botón de “crear”  
 onCreateClinicalHistory(new ClinicalHistory(hcNumber, hcName)); // Proveniente de un listener  
 });  
  
 loadData();  
 }  
  
 private void loadData() {  
  
 final ClinicalHistoriesViewModel viewModel = new ClinicalHistoriesViewModel();  
  
 // Observa el estado de carga  
 //viewModel.isDataLoading().observe(getViewLifecycleOwner(), isLoading -> { // in fragments  
 viewModel.isDataLoading().observe(this, isLoading -> { // in activity  
 if (isLoading)  
 showLoadingView();  
 });  
  
 viewModel.getData().observe(this, list -> {  
 viewModel.removeData();  
  
 clinicalHistoriesRecyclerView.initialize(list); // Inicializa la lista en el RecyclerView  
  
 if (list != null && list.size() > 0)  
 showListView();  
 else showListEmptyView();  
 });  
  
 viewModel.loadData(this); // requireActivity() in Fragments  
 }  
  
 private void onCreateClinicalHistory(final ClinicalHistory hc) {  
 if (clinicalHistoriesRecyclerView.getItemCount() == 0)  
 showListView();  
 clinicalHistoriesRecyclerView.add(hc);  
  
 }  
  
 private void onRemoveClinicalHistory(final ClinicalHistory hc) {  
 if (clinicalHistoriesRecyclerView.getItemCount() == 1)  
 showListEmptyView();  
 clinicalHistoriesRecyclerView.remove(hc);  
 }  
  
 private void showListEmptyView() {  
 listEmptyView.setVisibility(View.*VISIBLE*);  
 loadingView.setVisibility(View.*GONE*);  
 clinicalHistoriesRecyclerView.setVisibility(View.*GONE*);  
 }  
  
 private void showListView() {  
 listEmptyView.setVisibility(View.*GONE*);  
 loadingView.setVisibility(View.*GONE*);  
 clinicalHistoriesRecyclerView.setVisibility(View.*VISIBLE*);  
 }  
  
 private void showLoadingView() {  
 listEmptyView.setVisibility(View.*GONE*);  
 loadingView.setVisibility(View.*VISIBLE*);  
 clinicalHistoriesRecyclerView.setVisibility(View.*GONE*);  
 }  
  
 /\* private void showDialog(final Dialog dialog) {  
 findViewById(R.id.*dialog*).setVisibility(View.*VISIBLE*);  
 getSupportFragmentManager()  
 .beginTransaction()  
 .replace(R.id.*dialog*, dialog)  
 .commit();  
 } \*/  
}

**Fragment**

Puedes crear un Fragment mediante la siguiente instrucción

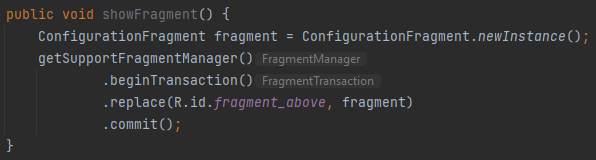


En la Activity donde se mostrará el Fragment, es necesario tener un FrameLayout para que sea su contenedor.

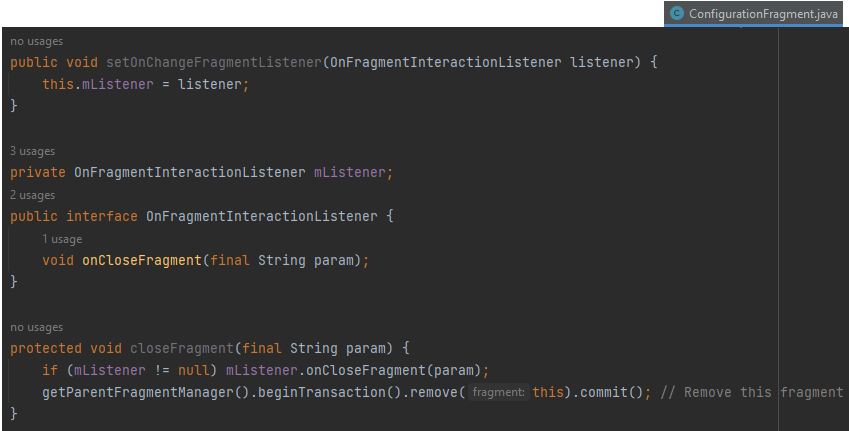


Luego, en la Activity (u otro Fragment) donde se mostrará el nuevo Fragment se ejecuta la siguiente instrucción.

*En este caso, el nuevo Fragment creado se llama ConfigurationFragment.*

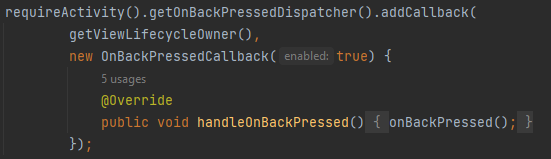


Es posible configurar un Listener en el nuevo Fragment, que puede ser útil para transportar información al Activity.



Luego, en el MainActivity se configura:

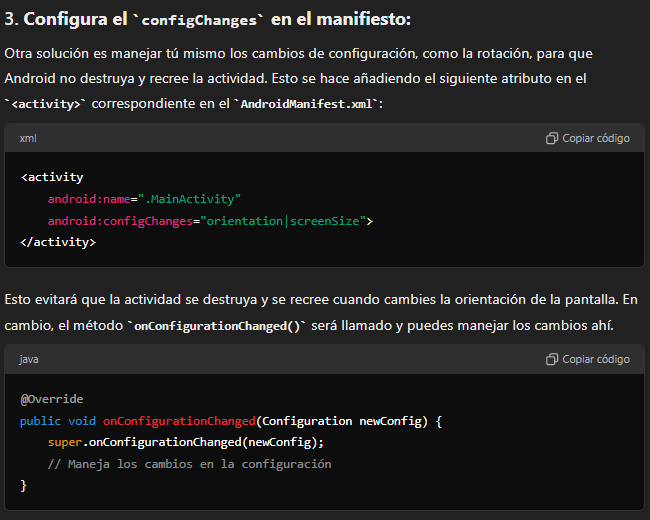
Se puede configurar en el Fragment lo que sucederá si se presiona el botón Atrás de la siguiente manera:



Donde **onBackPressed** será una función.

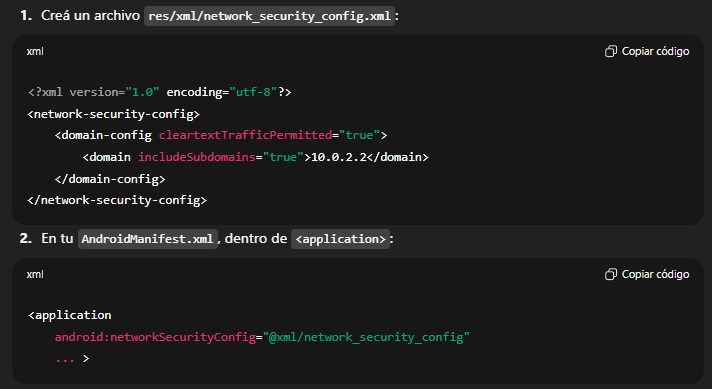
Un problema común de utilizar fragments, es que al rotar la pantalla se destruyen y recrean. Si no se manejan correctamente, crasheará la aplicación.

Este proceso se puede evitar de la siguiente forma:



**Peticiones HTTP**

Desde Android 9 sólo está permitido hacer peticiones HTTPS. Sin embargo, se puede configurar para permitir dichas peticiones.



Por otra parte, es necesario saber que, si se va a realizar una petición al **localhost**, hay que diferenciar la URL que se utiliza en el emulador de Android Studio al que se utiliza en un dispositivo físico.

* Emulador http://10.0.2.2:PUERTO
* Dispositivo físico http://192.168.0.xxx:PUERTO (ip local de la PC)

También será necesario definir el permiso de uso de internet desde el archivo **manifest** de la aplicación.

Además, para realizar peticiones a servidores se deben utilizar Threads para hacerlas en segundo plano y no detener toda la aplicación mientras se espera la respuesta del servidor.

Teniendo todo lo anterior en cuenta, se procede de la siguiente manera:

private void getMessages(final String domain) {  
  
 new Thread(() -> {  
 HttpURLConnection urlConnection = null;  
 try {  
 URL url = new URL(domain);  
 urlConnection = (HttpURLConnection) url.openConnection();  
 urlConnection.setRequestMethod("GET");  
  
 int status = urlConnection.getResponseCode();  
 if (status == 200) {  
 BufferedReader reader = new BufferedReader(  
 new InputStreamReader(urlConnection.getInputStream())  
 );  
  
 StringBuilder result = new StringBuilder();  
 String line;  
 while ((line = reader.readLine()) != null) {  
 result.append(line);  
 }  
  
 String json = result.toString();  
 // Ahora podés parsear el JSON (por ejemplo con org.json)  
 JSONArray array = new JSONArray(json);  
  
 // Message es un objeto creado dentro de la app (una class)  
 List<Message> mensajes = new ArrayList<>();  
 for (int i = 0; i < array.length(); i++) {  
 JSONObject obj = array.getJSONObject(i);  
 Message m = new Message(  
 obj.getInt("id"),  
 obj.getString("message"),  
 obj.getString("name"),  
 obj.getString("email")  
 );  
 mensajes.add(m);  
 }  
  
 // Volver al hilo principal para actualizar la UI  
 new Handler(Looper.*getMainLooper*()).post(() -> {  
 // Aquí usás la lista mensajes (ej: actualizar un TextView o una Lista)  
 for(Message mensaje : mensajes)  
 Log.*i*("Mensajes", mensaje.toString());  
 textView.setText(mensajes.get(0).toString());  
 });  
 }  
  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 if (urlConnection != null) {  
 urlConnection.disconnect();  
 }  
 }  
 }).start();  
}