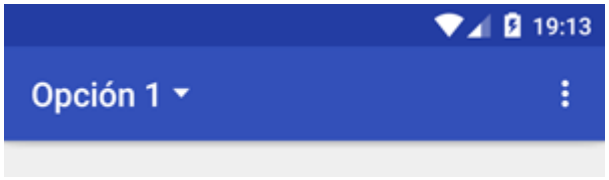
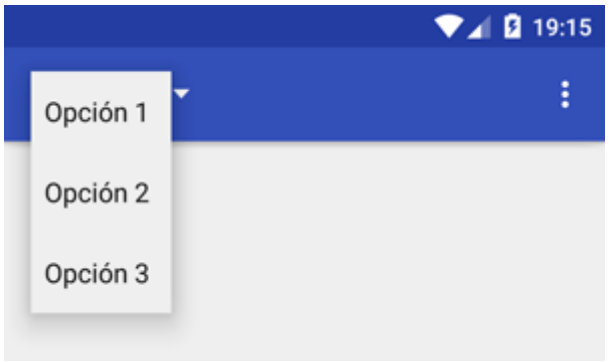
Actionbar / Appbar / Toolbar en Android (III): Filtros y Tabs

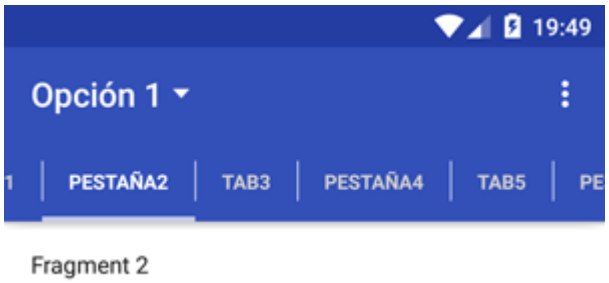
En los dos artículos anteriores aprendimos a hacer uso de la funcionalidad básica de una action bar y utilizar el nuevo componente Toolbar para conseguir el mismo comportamiento e incluso extenderlo a otras partes de la interfaz.

En este tercer artículo sobre el tema vamos a ver dos métodos de navegación aplicables a nuestras aplicaciones y que están íntimamente relacionados con la action bar. El primer de ellos, el más sencillo, será utilizar un *filtro* (*page filter*), o para entendernos mejor, una lista desplegable integrada en la action bar (o como ya dijimos, la *app bar*).





El segundo método, algo más laborioso aunque nos ayudaremos de algunas clases ya existentes, consiste en utilizar pestañas (*tabs*) bajo la action bar. Estas pestañas, a su vez, podrán ser fijas o deslizantes.



Hasta la llegada de Android 5.0, ambos métodos de navegación se conseguían, entre otras muchas cosas, llamando al método `setNavigationMode()` de la action bar con los valores `NAVIGATION_MODE_LIST` y `NAVIGATION_MODE_TABS` respectivamente. Sin embargo, con la versión 5.0 este método se ha marcado como *deprecated*, por lo que no se recomienda su uso. Aquí veremos formas alternativas de implementar ambos patrones de navegación.

Empecemos por la primera de las alternativas: el uso listas desplegables (*spinner*) integradas en la action bar. Esto no debería tener ningún misterio para nosotros con lo que ya llevamos aprendido en artículos anteriores. En primer lugar ya dijimos que el nuevo control `Toolbar` se comporta como cualquier otro contendor, en el sentido de que puede contener a otros controles. Por tanto, podemos empezar por añadir a nuestra aplicación un nuevo `Toolbar` que contenga un control `Spinner`.

```
1
2       <android.support.v7.widget.Toolbar
3           xmlns:android="http://schemas.android.com/apk/res/android"
4           xmlns:app="http://schemas.android.com/apk/res-auto"
5           android:layout_height="?attr/actionBarSize"
6           android:layout_width="match_parent"
7           android:minHeight="?attr/actionBarSize"
8           android:background="?attr/colorPrimary"
9           android:elevation="4dp"
10          android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
11          app:popupTheme="@style/ThemeOverlay.AppCompat.Light" >
12
13          <Spinner android:id="@+id/CmbToolbar"
14              android:layout_width="wrap_content"
15              android:layout_height="wrap_content" />
16
17      </android.support.v7.widget.Toolbar>
18
```

Igual que ya explicamos en el artículo anterior, incluiremos este código en un fichero independiente */res/layout/toolbar.xml*, que añadiremos al layout principal mediante la cláusula `<include>`.

Para que la lista desplegable no desentone con el estilo y colores de nuestra Toolbar vamos a personalizar los layouts específicos de los elementos de la lista desplegable y del control en sí, como ya explicamos en el artículo dedicado al control `Spinner`.

Para ello, definiremos en primer lugar el layout personalizado para el control (*/res/layout/appbar_filter_title.xml*), donde lo único destacable es que utilizará el mismo estilo estándar que el utilizado para el título de una action bar:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    style="@style/TextAppearance.AppCompat.Widget.ActionBar.Title"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Y en segundo lugar definiremos el layout de los elementos de la lista desplegable (*/res/layout/appbar_filter_list.xml*), donde utilizaremos como color de fondo y color de texto del estándar del tema Material Light:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    style="?android:attr/spinnerDropDownItemStyle"
    android:layout_width="match_parent"
    android:layout_height="48dp"
    android:background="@color/background_material_light"
    android:textColor="@color/primary_text_default_material_light" />
```

Por último, asociaremos todos estos elementos al spinner desde el método `onCreate()` de nuestra actividad, y por supuesto crearemos y asignaremos algún adaptador con las opciones seleccionables. Todo esto ya lo explicamos con detalle en el artículo sobre el control `Spinner`, por lo que no me detendré más en ello. Tan sólo indicar que en mi caso sólo añadiré a la lista 3 opciones de ejemplo mediante un `ArrayAdapter` sencillo:

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

    //Appbar
    Toolbar toolbar = (Toolbar) findViewById(R.id.appbar);
    setSupportActionBar(toolbar);
    getSupportActionBar().setDisplayShowTitleEnabled(false);

    //Appbar page filter
    Spinner cmbToolbar = (Spinner) findViewById(R.id.CmbToolbar);

    ArrayAdapter<String> adapter = new ArrayAdapter<>(
            getSupportActionBar().getThemedContext(),
            R.layout.appbar_filter_title,
            new String[]{"Opción 1 ", "Opción 2 ", "Opción 3 "});

    adapter.setDropDownViewResource(R.layout.appbar_filter_list);

    cmbToolbar.setAdapter(adapter);

    cmbToolbar.setOnItemSelectedListener(new
AdapterView.OnItemSelectedListener() {
        @Override
        public void onItemSelected(AdapterView<?> adapterView, View
view, int i, long l) {
            //... Acciones al seleccionar una opción de la lista
            Log.i("Toolbar 3", "Seleccionada opción " + i);
        }

        @Override
        public void onNothingSelected(AdapterView<?> adapterView) {
            //... Acciones al no existir ningún elemento seleccionado
        }
    });
```

Tras solo con esto tenemos ya listo nuestro filtro integrado en la action bar, con el aspecto de las imágenes mostradas al principio del artículo. Ya sólo quedaría responder a la selección de cada elemento de la lista, dentro del método `onItemSelected()`, con las acciones necesarias según la aplicación, por ejemplo, manipulando el resto de datos mostrados en la interfaz, o intercambiando *fragments* como veremos en breve para el caso de la navegación por pestañas.

Vamos ahora con el segundo de los métodos de navegación indicados, las pestañas o *tabs*. Las pestañas suelen ir colocadas justo bajo la action bar, y normalmente con el mismo color y elevación. Además, debemos poder

alternar entre ellas tanto pulsando sobre la pestaña elegida como desplazándonos entre ellas con el gesto de desplazar el dedo a izquierda o derecha sobre el contenido. Nota: Por este mismo motivo, cuando nuestro contenido interactúe también con gestos de este tipo (por ejemplo un mapa) no deberíamos utilizar pestañas en la interfaz.

Para conseguir esto, vamos a distinguir como mínimo 3 zonas básicas en la interfaz:

1. La action bar.
2. El bloque de pestañas.
3. El contenido de cada pestaña, que deberá ir integrado en algún contenedor que nos permita alternar entre ellas deslizando en horizontal.

- El primer punto ya lo vimos en detalle en el artículo anterior, por lo que no nos repetiremos.
- Para el segundo nos vamos a ayudar del componente `TabLayout`, contenido en la nueva librería de diseño (*Design Support Library*) liberada recientemente por Google.
- Y para el último punto utilizaremos el componente `ViewPager`, que nos permitirá alternar entre fragments con el gesto de deslizamiento (por tanto, el contenido de cada pestaña lo incluiremos en fragments independientes).

Para hacer uso de la nueva librería de diseño debemos en primer lugar añadir su referencia al fichero build.gradle, de la siguiente forma:

```
1    dependencies {
2        //...
3        compile 'com.android.support:design:22.2.0'
4    }
```

Hecho esto, ya podremos utilizar en nuestro layout el nuevo componente `TabLayout`, que nos servirá para albergar el grupo de pestañas. Y con esto

no me refiero al contenido de las pestañas, sino a las pestañas en sí. Utilizar este componente es muy sencillo, en principio basta con incluirlo en nuestro layout XML, sin asignar ninguna propiedad específica salvo los ya habituales `width`, `height` e `id`. El resto de la configuración la haremos desde el código java.

```
<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"
    tools:context=".MainActivity">

    <include layout="@layout/toolbar"
        android:id="@+id/appbar" />

    <android.support.design.widget.TabLayout
        android:id="@+id/appbartabs"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <FrameLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:foreground="@drawable/header_shadow">

        <android.support.v4.view.ViewPager
            android:id="@+id/viewpager"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:background="@android:color/white" />

    </FrameLayout>

</LinearLayout>
```

Si revisáis el código anterior veréis que en principio es bastante sencillo. Usamos como contenedor principal un `LinearLayout` vertical, en el que incluimos en primer lugar el `Toolbar` (la definición concreta es idéntica a la que vimos en el artículo anterior), tras éste incluimos el `TabLayout`, que contendrá el grupo de pestañas, y por último el `ViewPager` que contendrá los fragments de cada pestaña y permitirá la navegación entre ellas con el gesto de deslizamiento. Utilizamos además el método que ya explicamos en el

artículo anterior para añadir la sombra bajo la toolbar (en este caso bajo las pestañas) en versiones de Android anteriores a la 5, incluyendo el `ViewPager` dentro de un `FrameLayout` al que asignaremos el recurso de sombra como `foreground`.

Definido el layout XML nos toca empezar a inicializar y configurar todos estos componentes en el código java de la aplicación.

Empezaremos por el `ViewPager`. Este componente va a basar su funcionamiento en un *adaptador*, de forma similar a los controles tipo lista, aunque en esta ocasión será algo más sencillo. Como dijimos antes, el contenido de cada pestaña lo incluiremos en fragments independientes y el `ViewPager` se encargará de permitirnos alternar entre ellos. Para esto, deberemos construir un adaptador que extienda de `FragmentPagerAdapter`. En este adaptador tan sólo tendremos que implementar su constructor, y sobrescribir los métodos `getCount()`, `getItem(pos)` y `getPageTitle()`. Los nombres son bastante ilustrativos, el primero se encargará de devolver el número total de pestañas, el segundo devolverá el fragment correspondiente a la posición que reciba como parámetro, y el último devolverá el título de cada pestaña.

Para no alargar mucho el ejemplo, en mi caso voy a incluir 6 pestañas para que pueda comprobarse bien el deslizamiento, pero sólo crearé 2 fragments para los contenidos, por lo que el contenido se repetirá en muchas de ellas. En una situación normal cada pestaña tendría su contenido y por tanto su fragment independiente (o al menos filtrado o personalizado de algún modo).

Veamos cómo quedaría la implementación del adaptador, que es bastante directa según lo ya explicado:

```
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentManager;
import android.support.v4.app.FragmentPagerAdapter;
```

```
public class MiFragmentPagerAdapter extends FragmentPagerAdapter {
    final int PAGE_COUNT = 6;
    private String tabTitles[] =
            new String[] {"Tab Uno", "Tab Dos", "Tab Tres", "Tab
Cuatro", "Tab Cinco", "Tab Seis"};

    public MiFragmentPagerAdapter(FragmentManager fm) {
        super(fm);
    }

    @Override
    public int getCount() {
        return PAGE_COUNT;
    }

    @Override
    public Fragment getItem(int position) {

        Fragment f = null;

        switch(position) {
            case 0:
            case 2:
            case 4:
                f = Fragment1.newInstance();
                break;
            case 1:
            case 3:
            case 5:
                f = Fragment2.newInstance();
                break;
        }

        return f;
    }

    @Override
    public CharSequence getPageTitle(int position) {
        return tabTitles[position];
    }
}
```

Los nombres de cada pestaña los almaceno en un array clásico. Por su parte, las clases`Fragment1` y `Fragment2` son fragments muy sencillos, que muestran simplemente una etiqueta de texto con su nombre..

Para asociar este adaptador al componente *viewpager* llamaremos a su método `setAdapter()` desde el `onCreate()` de nuestra actividad principal:

```
//Establecer el PageAdapter del componente ViewPager
ViewPager viewPager = (ViewPager) findViewById(R.id.viewpager);
```

```
viewPager.setAdapter(new MiFragmentPagerAdapter(
        getSupportFragmentManager()));
```

Seguimos ahora con el `TabLayout`, tras obtener la referencia al control, tendremos que indicar el tipo de pestañas que queremos utilizar, entre *fijas* (`TabLayout.MODE_FIXED`) o *deslizantes* (`TabLayout.MODE_SCROLLABLE`), mediante su método `setTabMode()`. La modalidad de pestañas *fijas* (pestañas no deslizantes, todas del mismo tamaño) no debería utilizarse cuando el número de pestañas excede de 3 o 4, ya que probablemente la falta de espacio haga que los títulos aparezcan incompletos, no siendo nada buena la experiencia de usuario.

Por último, sólo nos quedará *enlazar* nuestro `ViewPager` con el `TabLayout`, lo que conseguiremos llamando al método `setupWithViewPager()`. Este último método hace bastante trabajo por nosotros, ya que se encargará de:

1. Crear las pestañas necesarias en el `TabLayout`, con sus títulos correspondientes, a partir de la información porporcionada por el adaptador del `ViewPager`.
2. Asegurar el cambio de la pestaña seleccionada en el `TabLayout` cuando el usuario se desplaza entre los fragments del `ViewPager`. Para ello define automáticamente el listener `ViewPager.OnPageChangeListener`.
3. Y asegurar también la propagación de eventos en "sentido contrario", es decir, asegurará que cuando se selecciona directamente una pestaña del `TabLayout`, el `ViewPager` muestra el fragment correspondiente. Para ello definirá el listener `TabLayout.OnTabSelectedListener`.

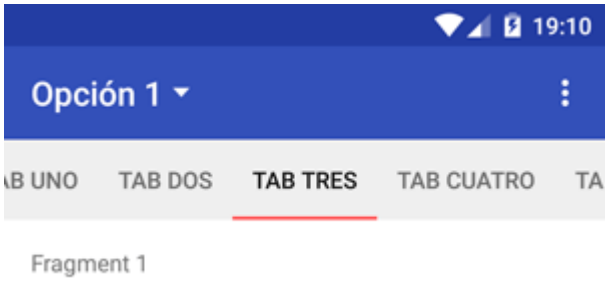
Es importante tener todo esto en cuenta ya que si necesitamos personalizar de alguna forma la respuesta a estos eventos, o no estamos utilizando un

`ViewPager`, quizá tengamos que hacer este trabajo por nosotros mismos. Si se diera este caso, tenéis disponible los métodos `newTab()` y `addTab()` del `TabLayout` con el que podréis crear y añadir manualmente las pestañas al control.

Veamos cómo quedaría todo esto dentro del `onCreate()` de la actividad principal:

```
TabLayout tabLayout = (TabLayout)
findViewById(R.id.appbartabs);
tabLayout.setTabMode(TabLayout.MODE_SCROLLABLE);
tabLayout.setupWithViewPager(viewPager);
```

Si ejecutamos la aplicación en este momento deberíamos obtener un resultado idéntico al mostrado al comienzo del artículo.



Como puede comprobarse, todo funciona correctamente salvo por el hecho de que las pestañas no comparten el color de la action bar superior. Esto tiene fácil solución haciendo uso de otro de los nuevos componentes presentados con la nueva librería de diseño, el `AppBarLayout`. Este nuevo tipo de layout tendrá especial interés de cara a facilitar la tarea de *animar* tanto la action bar como sus componentes relacionados (por ejemplo las pestañas) como respuesta a otros eventos, como el desplazamiento en una lista. Esto lo veremos en el siguiente artículo, pero por el momento ya nos va a servir también para hacer más consistente el aspecto de nuestras pestañas con la action bar de la actividad y con el tema definido para la aplicación.

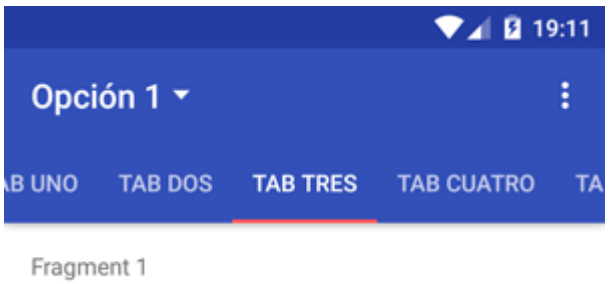
Para ello bastará con incluir el `Toolbar` y el `TabLayout` de nuestra interfaz dentro de un elemento `AppBarLayout`. En este nuevo AppBarLayout tan sólo asignaremos la propiedad `android:theme`, tal como ya vimos en el artículo anterior sobre el control `Toolbar`.

```
<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"
    tools:context=".MainActivity">

    <android.support.design.widget.AppBarLayout
        android:id="@+id/appbarlayout"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar" >

        <include layout="@layout/toolbar"
            android:id="@+id/appbar" />

        <android.support.design.widget.TabLayout
            android:id="@+id/appbartabs"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />

    </android.support.design.widget.AppBarLayout>

    <FrameLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:foreground="@drawable/header_shadow">

        <android.support.v4.view.ViewPager
            android:id="@+id/viewpager"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:background="@android:color/white" />

    </FrameLayout>

</LinearLayout>
```

Lo único a tener en cuenta, para evitar efectos indeseados, es que debemos eliminar la asignación del atributo `elevation` del control `Toolbar`. El nuevo `AppBarLayout` se encargará de establecer la elevación estandar al conjunto `Toolbar+TabLayout` y mostrar la sombra correspondiente sin

necesidad de añadir elementos adicionales (por ahora parece que solo en APIs > 21).

Si volvemos a ejecutar ahora la aplicación, el resultado tendrá ya el aspecto y funcionamiento deseados, las pestañas utilizarán el mismo color que la action bar (*primary color*) y el indicador de la pestaña seleccionada utilizará por defecto el *accent color* definido en el tema:



Por el momento no es posible establecer por código un color alternativo para el indicador de selección o los divisores de pestañas, pero no creo que tardemos mucho en tener disponible esta característica en la librería, ya que sí estaba presente en el antiguo componente `SlidingTabLayout`, precursor "no oficial" del actual `TabLayout`.

Puedes consultar y/o descargar el código completo de los ejemplos desarrollados en este artículo accediendo a la página del curso en GitHub.