

# Desarrollar Aplicación Android De Contactos

Como ya sabes, la aplicación Android manejará un CRUD de contactos personales para los usuarios que usen la app. No será una app corta, por lo que ignoraré en el desarrollo la creación de la versión para tabletas o para la posición landscape, con el fin de reducir el artículo a un material compresible.

El login de la app no lo tocaremos en este artículo, pero de seguro será el próximo un próximo tema para el blog.

Mi objetivo es mostrarte directamente una estrategia de sincronización de datos entre el servidor y la app a través de nuestro servicio web.

Ahora, la metodología en que te mostraré el trabajo está constituida por las siguientes etapas:

- Determinar Requerimientos De La App
- Crear Wireframing de la App
- Diseño De La Base De Datos Para Mysql
- Extender Servicio REST Con Operaciones En Batch
- Desarrollo De La Aplicación
- Crear Nuevo Proyecto En Android Studio
  - Crear Lista De Contactos
  - Crear Actividad De Inserción
  - Usar Actividad De Inserción Para Modificar
  - Eliminar Contactos De La Aplicación
  - Sincronizar El Cliente Android
  - Sincronizar El Servidor Con Php y Mysql

Con esto en mente, ¡arranquemos!

## #1. Requerimiento Para Consumir Un Servicio Web

Las características de la app ya las había mencionado arriba de forma general, sin embargo anotarlas nos vendría bien.

La app de contactos debe tener las siguientes peticiones del usuario hipotético:

- *Como Usuario hipotético, deseo guardar el primer nombre, primer apellido, teléfono y correo de mis contactos personales y verlos en una lista.*
- *Como Usuario hipotético, deseo modificar los datos de un contacto existente.*
- *Como Usuario hipotético, deseo eliminar contactos.*

- Como Usuario hipotético, quiero que los cambios que haga en mi app Android se vean reflejados en mi cuenta web y viceversa.

Los tres primeros requerimientos podríamos agruparlos en «**Administrar los contactos**», sin embargo por practicidad y orden prefiero que los entiendas de esa forma.

## Selección de estrategia para sincronización

Es extremadamente importante que en esta sección aclares la **estrategia de sincronización** que usarás en tu aplicación. Una de la forma más sencilla de hacerlo es especificando que parte será la más preponderante a la hora de replicar la información.

En mi caso tanto el servidor como el cliente tendrán el mismo peso. Lo que quiere decir, que los registros locales y remotos deben conservarse en las mismas condiciones. Sumándole que el [proceso de sincronización será por demanda](#).

*Por ejemplo...*

Supón que la aplicación Android no posee ningún contacto por el momento. Del otro lado tenemos al servidor Php con un contacto ya creado desde la aplicación web.

— ¿Qué pasaría si se iniciara una sincronización en ese preciso instante según lo que mencioné?




El registro del cliente se copiaría en la base de datos SQLite de la aplicación Android.

— Ahora, ¿Qué sucedería en caso contrario?

Es decir, que el cliente tenga un contacto local creado, pero que el servidor aún no posea registros.

**Respuesta:** el registro se copiaría en la base de datos Mysql, ya que el registro es tratado como prioritario.

Este comportamiento se puede resumir en la siguiente ilustración:

	Estado inicial	Luego de cambios	Estado Final
<b>Servidor</b>	Vacío		
<b>Cliente</b>	Vacío		

Cuando se trata de las operaciones sencillas de forma aislada no existen problemas, pero las cosas se ponen complejas cuando un mismo registro es editado tanto en el cliente como en el servidor antes de la sincronización.

Este tipo de conflictos pueden ser solucionados de muchas formas posibles, pero por supuesto todo depende.

Para PeopleApp decidí usar una **estrategia de marcas de tiempo (timestamps)** para comparar las fechas de modificación y elegir al más reciente.

*Ejemplo...*

El contacto **C-111** fue editado desde el cliente el *21/12/2015 a las 5pm*. Luego se editó en el servidor ese mismo día a las *6pm*. A las *9pm* se realizó una sincronización entre ambas partes.

— *¿Qué versión crees que sobrevivió?*

La del servidor. Obviamente porque es el dato más reciente como lo había mencionado. Así que se realiza una réplica en el registro local.

## Más estrategias de sincronización

Cada app es distinta, en consecuencia no todas las estrategias serán las correctas para tu caso particular.

Un buen caso sería en donde tu **aplicación Android actúa como un receptor de las últimas noticias de tu web**. Si me preguntas que parte es más preponderante, yo diría que el servidor. Ya que el origen de los datos está basado en tu web, por lo que la app móvil no contribuye en nada a la creación de contenido.

Aquí el servidor sería maestro y el cliente un esclavo. Donde sí los registros son borrados del servidor, el cliente también tendrá que replicar estas eliminaciones.

Este caso lo vimos cuando [creamos un lector RSS](#).

De la misma forma puede ocurrir que la aplicación Android sea el lado maestro y el servidor el esclavo. Donde todo cambio en la app tendría mayor peso que el servidor.

Adicionalmente, en la resolución de conflictos podríamos elegir la versión más antigua en vez de la actual. O incluso elegir la versión del cliente o servidor según la conveniencia.

Aunque también existe la alternativa de proporcionar un control de versiones del registro para que el usuario elija el cambio que más le conviene.

Piénsatelo muy bien y actúa según las políticas para proteger los datos del usuario.

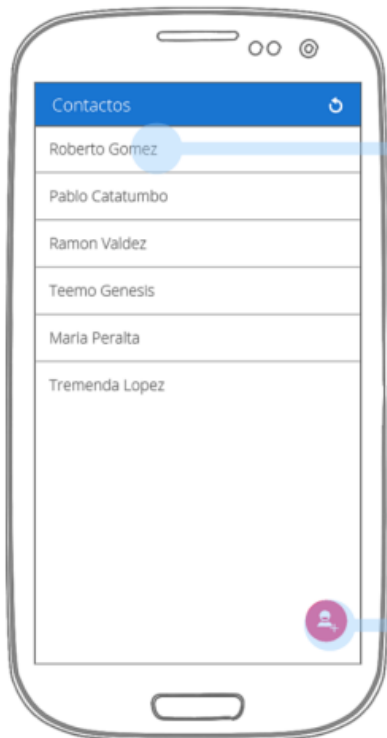
## #2. Crear Wireframing De La App

*PeopleApp* es una aplicación CRUD realmente sencilla y no requiere mucho esfuerzo mapear las relaciones de las pantallas.

De los requerimientos sabemos que la lista de pantallas que tendremos se compone de:

- Lista de contactos
- Formulario de creación de contactos
- Formulario para edición de contactos

Esto reduce nuestra planeación de UI al siguiente wireframing:



### #3. Modificación De La Base De Datos Mysql

El siguiente movimiento es agregar el soporte de versiones para los registros de la tabla 'contacto'.

Esto significa añadir una nueva columna llamada 'version' del tipo DATETIME. Si en tu caso necesitas la precisión de centésimos de segundo, entonces usa el tipo TIMESTAMP.

Al crear este campo le añadí el activador ON UPDATE CURRENT\_TIMESTAMP para que Mysql actualice automáticamente la fecha. Esto me evita crear un trigger para asignar la fecha actual.

En código, la exportación de Mysql me arroja al siguiente comando CREATE :

```
--  
-- Estructura de tabla para la tabla `contacto`  
--  
  
CREATE TABLE IF NOT EXISTS `contacto` (  
  `idContacto` varchar(255) NOT NULL,  
  `primerNombre` varchar(40) NOT NULL,  
  `primerApellido` varchar(40) DEFAULT NULL,  
  `telefono` varchar(10) DEFAULT NULL,  
  `correo` varchar(254) DEFAULT NULL,  
  `idUsuario` int(11) NOT NULL,  
  `version` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP  
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

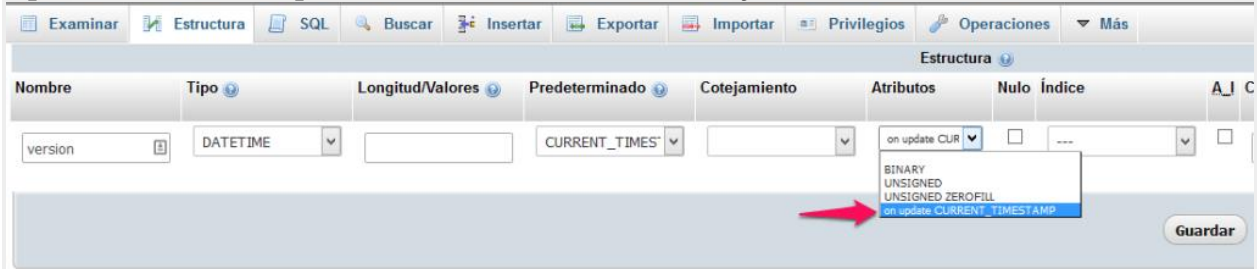
Si deseas realizarlo a través de *phpMyAdmin*, entonces solo vas a la pestaña «Estructura» y luego presionas «Continuar» en la sección de «Agregar».

The screenshot shows the phpMyAdmin interface for a database. The 'Estructura' (Structure) tab is selected, indicated by a red arrow and the number '1'. Below the menu, a table lists the columns of the 'contacto' table:

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Extra	Acción
1	idContacto	varchar(16)	latin1_swedish_ci		No	Ninguna		Cambiar Eliminar Más
2	primerNombre	varchar(40)	latin1_swedish_ci		No	Ninguna		Cambiar Eliminar Más
3	primerApellido	varchar(40)	latin1_swedish_ci		No	Ninguna		Cambiar Eliminar Más
4	telefono	varchar(10)	latin1_swedish_ci		No	Ninguna		Cambiar Eliminar Más
5	correo	varchar(254)	latin1_swedish_ci		No	Ninguna		Cambiar Eliminar Más
6	idUsuario	int(11)			No	Ninguna		Cambiar Eliminar Más

Below the table, there is a section for adding a new column. It includes a 'Agregar' button, a dropdown menu for the column name (currently showing 'idContacto'), and a 'Continuar' button. A red arrow and the number '2' point to the 'Continuar' button.

Cuando estés en el asistente para crear la columna, ponle el nombre "version", tipo DATETIME, valor por defecto CURRENT\_TIMESTAMP y selecciona el atributo ON UPDATE:



## #4. Crear Operaciones En Batch Para El Servicio REST

De la forma en que tengo el servicio web REST hasta el momento tendría que enviar gran cantidad de peticiones para insertar, modificar y eliminar contactos.

Ponte a pensar. Si tuviésemos 100 contactos añadidos en la aplicación Android, entonces requeriré 100 peticiones POST hacia el servidor para replicar la información.

Aunque parece que no hay problema, en realidad [los dispositivos móviles consumen batería por cada ciclo de envío](#) de peticiones HTTP. Con unas cuantas no habrá problema, pero si necesitamos actualizar un gran número de registros, entonces debes pensártelo dos veces.

Para solucionar este inconveniente implementaré [operaciones batch](#) en el servicio web. Estas no son mas que operaciones que se realizan en bloque o lotes de datos para optimizar tratar una sola carga de datos.

Esto te permitirá realizar las tres operaciones básicas con una sola petición de uno o más recursos, es decir, insertar, actualizar y eliminar con una solo método.

### Aplicar batch en Mysql a través de Php

Para formar la URI que controle el batch crearé un nuevo archivo llamado **sync.php** dentro del directorio **controladores**.

Su objetivo es comportarse como un recurso que represente a todos los recursos existentes de la base de datos. Así que lo consultaremos para enviarle una petición con los cambios mezclados u obtener datos.

La referencia de acciones quedaría de la siguiente forma:

URI	Descripción
-----	-------------

GET  
api.peopleapp.com/v1/sync

Obtiene todos los registros de la tabla contactos

POST  
api.peopleapp.com/v1/sync

Envía las inserciones, modificaciones y eliminaciones de registros de contactos.

Este recurso adicional mostraría un gran potencial si tuviésemos más de dos tablas. Sin embargo, con el ejemplo de contactos ya tendrás una lógica para expandir tus apps.

## Obtención de datos en Batch

a). Lo primero que haré será agregar el recurso **sync** a la lista de recursos en el archivo **index.php**. Recuerda que esto permite reconocer las operaciones.

*index.php*

```
<?php

...
require 'controladores/sync.php';
...
$recursos_existentes = array('contactos', 'usuarios', 'sync');
```

b). Debido a que enviaremos los datos de todas las tablas (menos 'usuario' ya que no necesita sincronización), entonces crearé una lista con los nombres de cada una.

El propósito es la lectura con un ciclo `foreach` de todos sus elementos para construir un objeto JSON con los datos de cada una.

*sync.php*

```
<?php

include_once 'datos/ConexionBD.php';

class sync
{
    // Constantes...

    /* Añade todas los recursos que deseas enviar separados por coma ','
     *      ejemplo: array('cliente', 'factura', 'producto')
     */
    public static $tablas = array(contactos::TABLA_CONTACTO);
```



```
}
```

c). Ahora escribiré el método general `get()` para procesar la petición GET enviada desde el cliente. En esencia, solo debes autorizar el usuario y luego comenzar a extraer todos los datos.

La autorización ya la habías visto en [el artículo anterior](#) con el método `autorizar()`. Para la consulta a la base de datos usaré un nuevo método llamado `obtenerRecursos()`, el cual recibe el filtro de la petición y el id del usuario. Sabiendo esto el código quedaría así:

```
public static function get($segmentos)
{

    $idUserio = usuarios::autorizar();
    return self::obtenerRecursos($idUserio);

}
```

d). El método `obtenerRecursos()` debe recorrer el array `tablas` para consultar todo el contenido de cada una. A medida que se van encontrando los registros, se van añadiendo a un array de respuesta, donde la clave de cada tabla será su mismo nombre.

```
private function obtenerRecursos($idUserio)
{

    try {
        // Instancia PDO
        $pdo = ConexionBD::obtenerInstancia()->obtenerBD();

        // Preparar array de parámetros
        $parametros = array($idUserio);

        // Procesar recursos a enviar
        foreach (self::$tablas as $tabla) {

            // Consulta genérica del recurso i
            $comando = 'SELECT * FROM ' . $tabla . ' WHERE ' . self::ID_USUARIO .
            '=?';

            // Preparar sentencia
```

```

        $sentencia = $pdo->prepare($comando);

        // Ejecutar sentencia preparada
        $sentencia->execute($parametros);

        // Extraer datos como array asociativo
        $respuesta[$tabla] = $sentencia->fetchAll(PDO::FETCH_ASSOC);
    }

    // Estado 200 OK
    http_response_code(200);

    $respuesta['estado'] = self::ESTADO_EXITO;
    $respuesta['mensaje'] = utf8_encode(self::MENSAJE_100);

    return $respuesta;

} catch (PDOException $e) {
    throw new ExcepcionApi(self::ESTADO_ERROR_BD, $e->getMessage());
}
}

```

Un ejemplo de una respuesta JSON con estado 100 OK sería la siguiente:

```

{
    "contacto": [
        {
            "idContacto": "C-709c040d-5cbb-4def-a248-444d1eba71f1",
            "primerNombre": "James",
            "primerApellido": "Revelo",
            "telefono": "1811918",
            "correo": "james@gmail.com",
            "idUsuario": "2",
            "version": "2015-12-23 12:24:28"
        }
    ],
    "estado": 100,
    "mensaje": "Sincronización completa"
}

```

Como ves, hay solo un array llamado 'contacto' con un registro de ejemplo. Si hubiésemos mas tablas, entonces aparecerían de la misma forma con sus respectivos contenidos, para que los proceses desde la app Android.

## Actualización De Datos En Batch

Esta tarea es un poco más compleja debido a que ya no enviarás peticiones individuales por cada operación, si no que usarás un solo canal para todo.

*¿Qué debes tener en cuenta?*

- El método HTTP a usar para la petición batch.
- La cantidad de tablas que contribuirán a la actualización masiva.
- El tipo de operaciones que enviarás por cada tabla.
- La estructura JSON o XML del cuerpo de la petición

Cuando evalué estas características decidí lo siguiente:

- El método a usar será POST, ya que permite enviar un cuerpo en la petición de forma flexible y multivariada.
- La única tabla que hay para sincronizar es contacto.
- Enviaré las inserciones, modificaciones y eliminaciones.
- Usaré un objeto JSON con 3 arreglos que representen las operaciones mencionadas arriba. Algo como:

```
{
  "inserciones": [
    ..
  ],
  "modificaciones": [
    ..
  ],
  "eliminaciones": [
    ..
  ]
}
```

```
}
```

Existen muchas maneras de realizar un batch en una petición, y de seguro verás unas muy ingeniosas que usan los parámetros de las URLs o incluso las cabeceras HTTP. Incluso la combinación de varias peticiones HTTP en una sola.

Pero como siempre digo, las prácticas variarán según tus necesidades y estilo de programación.

Veamos como formar el script Php.

a). En primer lugar crea un método llamado `post()` para tratar la petición. Ahora piensa un poco en el algoritmo a seguir para traducir la payload.

*¿Cuál sería el orden?*

Veamos:

1. Autorizar el usuario
2. Extraer el cuerpo JSON del contenido de la petición POST
3. Decodificar el objeto en un array asociativo
4. Aplicar operaciones en batch descritas en el array

Como ves, lo único que no he explicado antes es realizar las operaciones en batch, pero las primeras acciones ya sabes aplicarlas. El batch lo implementaremos en un método por separada llamado `aplicarBatch()`, el cual recibe el mensaje decodificado y el id del usuario.

```
public static function post($segmentos)
{
    $idUserio = usuarios::autorizar();

    $mensajePlano = file_get_contents('php://input');

    $mensajeDecodificado = json_decode($mensajePlano, PDO::FETCH_ASSOC);

    if (!empty($mensajeDecodificado)) {
        self::aplicarBatch($mensajeDecodificado, $idUserio);
        // Contruir respuesta
        $respuesta['estado'] = self::ESTADO_EXITO;
        $respuesta['mensaje'] = utf8_encode(self::MENSAJE_100);
        http_response_code(200);
    } else {
        // Respuesta error
        throw new ExcepcionApi(self::ESTADO_MALA_SINTAXIS, self::MENSAJE_103, 422);
    }

    return $respuesta;
}
```

b). El método `aplicarBatch()` abre una transacción de bases de datos para comenzar a aplicar inserciones, modificaciones y eliminaciones de los contactos. Una vez todo se ha llevado a cabo, la transacción termina y se confirman los cambios remotos.

La aplicación de cada operación se lleva a cabo con unos nuevos métodos de la clase `contactos` que veremos en el siguiente paso:

```
private function aplicarBatch($payload, $idUser)
{
    $pdo = ConexionBD::obtenerInstancia()->obtenerBD();

    /*
     * Verificación: Confirmar que existe al menos un tipo de operación
     */
    if (!isset($payload[self::INSERCIONES]) && !isset($payload[self::MODIFICACIONES])
        && !isset($payload[self::ELIMINACIONES]))
    {
        throw new ExcepcionApi(self::ESTADO_MALA_SINTAXIS, self::MENSAJE_103, 422);
    }

    try {

        // Comenzar transacción
        $pdo->beginTransaction();

        // Inserciones
        if (isset($payload[self::INSERCIONES]))
            contactos::insertarEnBatch($pdo, $payload[self::INSERCIONES], $idUser);

        // Modificaciones
        if (isset($payload[self::MODIFICACIONES]))
            contactos::modificarEnBatch($pdo, $payload[self::MODIFICACIONES],
            $idUser);

        // Eliminaciones
        if (isset($payload[self::ELIMINACIONES])) {
            contactos::eliminarEnBatch($pdo, $payload[self::ELIMINACIONES],
            $idUser);
        }

        // Confirmar cambios
```

```

        $pdo->commit();

    } catch (PDOException $e) {
        throw new ExcepcionApi($pdo->errorCode(), $e->getMessage(), 422);
    }
}

```

Observa que yo comienzo a llamar los objetos 'inserciones', 'modificaciones' e 'inserciones' directamente desde payload. Esto lo hago porque tengo solo la tabla contactos. Pero si hay más de una, debes crear un ciclo for o foreach para procesar la lista de tablas. Donde el cuerpo de la petición vendría con la siguiente sintaxis:

```

{
  "tabla1":{
    "inserciones":[

    ],
    "modificaciones":[

    ],
    "eliminaciones":[

    ]
  },
  "tabla2":{
    "inserciones":[

    ],
    "modificaciones":[

    ],
    "eliminaciones":[

    ]
  },
  ...,
  "tablan":{
    "inserciones":[

```

```

    ],
    "modificaciones":[

    ],
    "eliminaciones":[

    ]
  }
}

```

Cada recurso a tratar en batch debe tener sus respectivos métodos `insertarEnBatch()`, `modificarEnBatch()` y `eliminarEnBatch()`. Así podrás generalizar la llamada de los métodos dentro del `for`, para evitar hacer referencia a cada nombre dentro del ciclo.

c). El siguiente paso es escribir el método `insertarEnBatch()` en **contactos.php**. Este tipo de métodos se caracteriza por usar un bucle para procesar cada comando de la base de datos. En este caso la sentencia `INSERT` en `Mysql`.

La idea es leer todos los elementos del array 'inserciones' y ejecutar una sentencia preparada con los datos de cada ítem.

Las inserciones de contactos tienen la siguiente forma en el cuerpo JSON:

```

"inserciones":[
  {
    "idContacto":"v1",
    "primerNombre":"v2",
    "primerApellido":"v3",
    "telefono":"v4",
    "correo":"v5",
    "version":"v6"
  },
  {
    ...
  },
  ...,
  {
    ...
  }
]

```

En cada iteración del ciclo que procesa las inserciones mapearemos la sentencia SQL con los nombres que vemos en el registro de ejemplo anterior de la siguiente forma:

### *Dentro de contactos.php*

```
public static function insertarEnBatch(PDO $pdo, $listaContactos, $idUser)
{
    // Sentencia INSERT
    $comando = 'INSERT INTO ' . self::TABLA_CONTACTO . ' ( ' .
        self::ID_CONTACTO . ',' .
        self::PRIMER_NOMBRE . ',' .
        self::PRIMER_APELLIDO . ',' .
        self::TELEFONO . ',' .
        self::CORREO . ',' .
        self::ID_USUARIO . ',' .
        self::VERSION . ' ) ' .
        ' VALUES(?, ?, ?, ?, ?, ?, ?) ';

    // Preparar la sentencia
    $sentencia = $pdo->prepare($comando);

    $sentencia->bindParam(1, $idContacto);
    $sentencia->bindParam(2, $primerNombre);
    $sentencia->bindParam(3, $primerApellido);
    $sentencia->bindParam(4, $telefono);
    $sentencia->bindParam(5, $correo);
    $sentencia->bindParam(6, $idUser);
    $sentencia->bindParam(7, $version);

    foreach ($listaContactos as $item) {
        $idContacto = $item[self::ID_CONTACTO];
        $primerNombre = $item[self::PRIMER_NOMBRE];
        $primerApellido = $item[self::PRIMER_APELLIDO];
        $telefono = $item[self::TELEFONO];
        $correo = $item[self::CORREO];
        $version = $item[self::VERSION];
        $sentencia->execute();
    }
}
```



```
}
```

d). De la misma manera procesamos las modificaciones, ya que el contenido del array JSON 'modificaciones' viene configurado de forma similar.

```
"modificaciones":[
  {
    "idContacto":"v1",
    "primerNombre":"v2",
    "primerApellido":"v3",
    "telefono":"v4",
    "correo":"v5",
    "version":"v6"
  },
  {
    ...
  },
  ...,
  {
    ...
  }
]
```

Esta vez usa la sentencia UPDATE con placeholders en los valores nuevos y liga los parámetros que vienen desde el cliente.

```
public static function modificarEnBatch(PDO $pdo, $arrayContactos, $idUser)
{
    // Preparar operación de modificación para cada contacto
    $comando = 'UPDATE ' . self::TABLA_CONTACTO . ' SET ' .
        self::PRIMER_NOMBRE . '=?, ' .
        self::PRIMER_APELLIDO . '=?, ' .
        self::TELEFONO . '=?, ' .
        self::CORREO . '=?, ' .
        self::VERSION . '=? ' .
        ' WHERE ' . self::ID_CONTACTO . '=? AND ' . self::ID_USUARIO . '=?';

    // Preparar la sentencia update
    $sentencia = $pdo->prepare($comando);

    // Ligar parametros
```

```

        $sentencia->bindParam(1, $primerNombre);
        $sentencia->bindParam(2, $primerApellido);
        $sentencia->bindParam(3, $telefono);
        $sentencia->bindParam(4, $correo);
        $sentencia->bindParam(5, $version);
        $sentencia->bindParam(6, $idContacto);
        $sentencia->bindParam(7, $idUser);

        // Procesar array de contactos
        foreach ($arrayContactos as $contacto) {
            $idContacto = $contacto[self::ID_CONTACTO];
            $primerNombre = $contacto[self::PRIMER_NOMBRE];
            $primerApellido = $contacto[self::PRIMER_APELLIDO];
            $telefono = $contacto[self::TELEFONO];
            $correo = $contacto[self::CORREO];
            $version = $contacto[self::VERSION];
            $sentencia->execute();
        }
    }
}

```

e). En cuanto a las eliminaciones el asunto es mucho más sencillo. Solo recibirás los identificadores de aquellos contactos a eliminar, por lo que tendremos un arreglo sencillo de strings como el siguiente:

```

"eliminaciones":[
    "C-35b60dc3-6003-4980-9b78-8982d1739ca6",
    "C-8f4b3c76-a831-4554-abc7-7a634d150d4a",
    ...,
    "C-4625d9dd-92a0-4036-bdf9-789486a38c72"
]

```

Esto solo nos deja iterar sobre una sentencia preparada basada en un DELETE con dos placeholders. Uno para el identificador del contacto y otro para el del usuario.

```

public static function eliminarEnBatch(PDO $pdo, $arrayIds, $idUser)
{
    // Crear sentencia DELETE
    $comando = 'DELETE FROM ' . self::TABLA_CONTACTO .
        ' WHERE ' . self::ID_CONTACTO . ' = ? AND ' . self::ID_USUARIO . '=?';

    // Preparar sentencia en el contenedor

```

```
$sentencia = $pdo->prepare($comando);

// Procesar todas las ids
foreach ($arrayIds as $id) {
    $sentencia->execute(array($id, $idUserario));
}

}
```

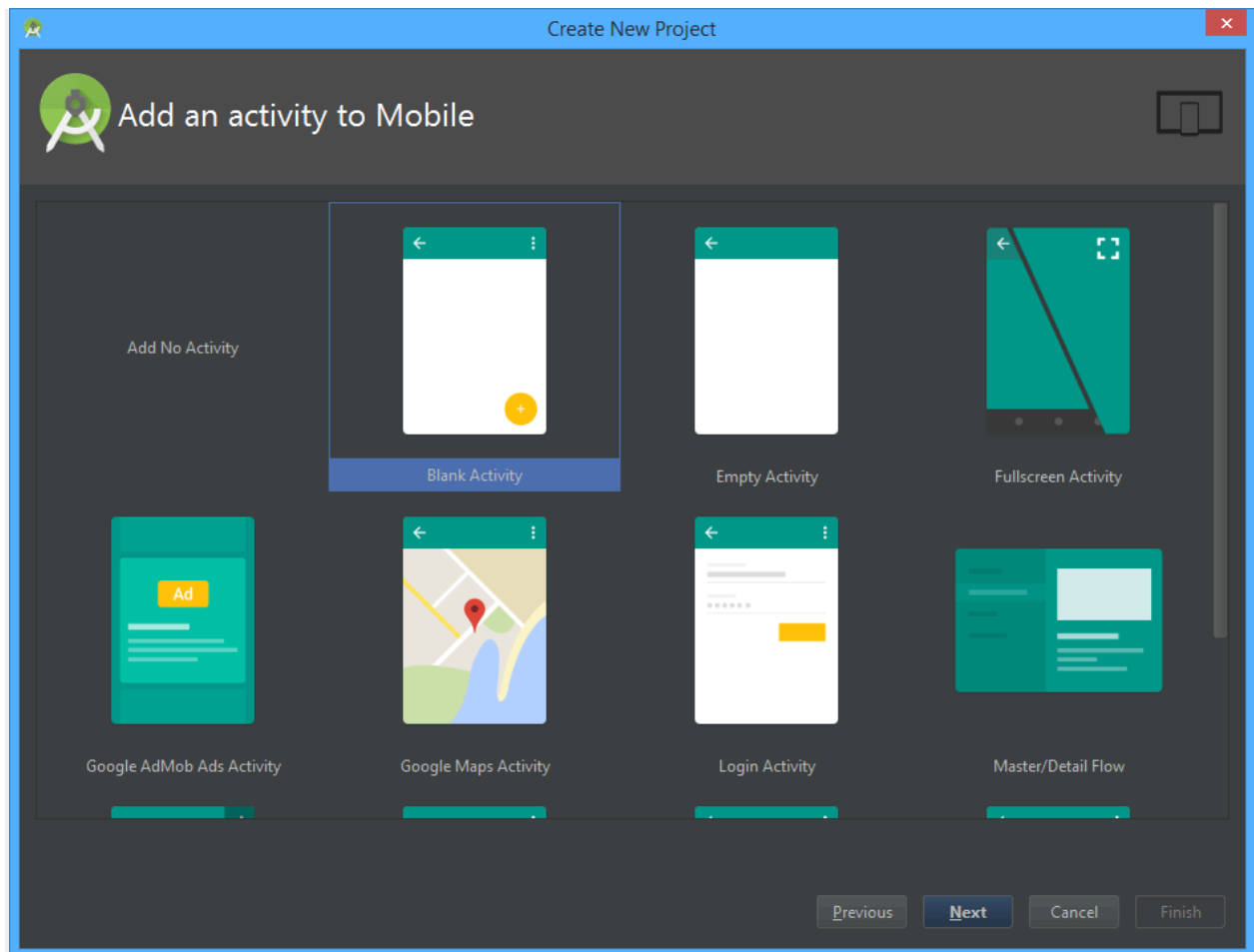
Con todas las modificaciones anteriores el servicio web REST queda preparado para obtener todos los datos a sincronizar y para modificar con un solo envío.

Ahora solo queda crear la app Android para probar estas características.

## #5. Desarrollo De La Aplicación Android

### 5.1 Crear nuevo proyecto en Android Studio

**Paso #1.** Abre Android Studio, ve a **File > New > New Project...** y asígnale el nombre de «*PeopleApp*». Confirma las características y al final añade una actividad en blanco (**Blank Activity**) para tener un avance de la lista de contactos.



**Paso #2.** Elige los esquemas de colores para el Material Design. En el wireframing vimos que tendremos como paleta principal con un degradado de azul y una paleta secundaria con rosa para los acentos.

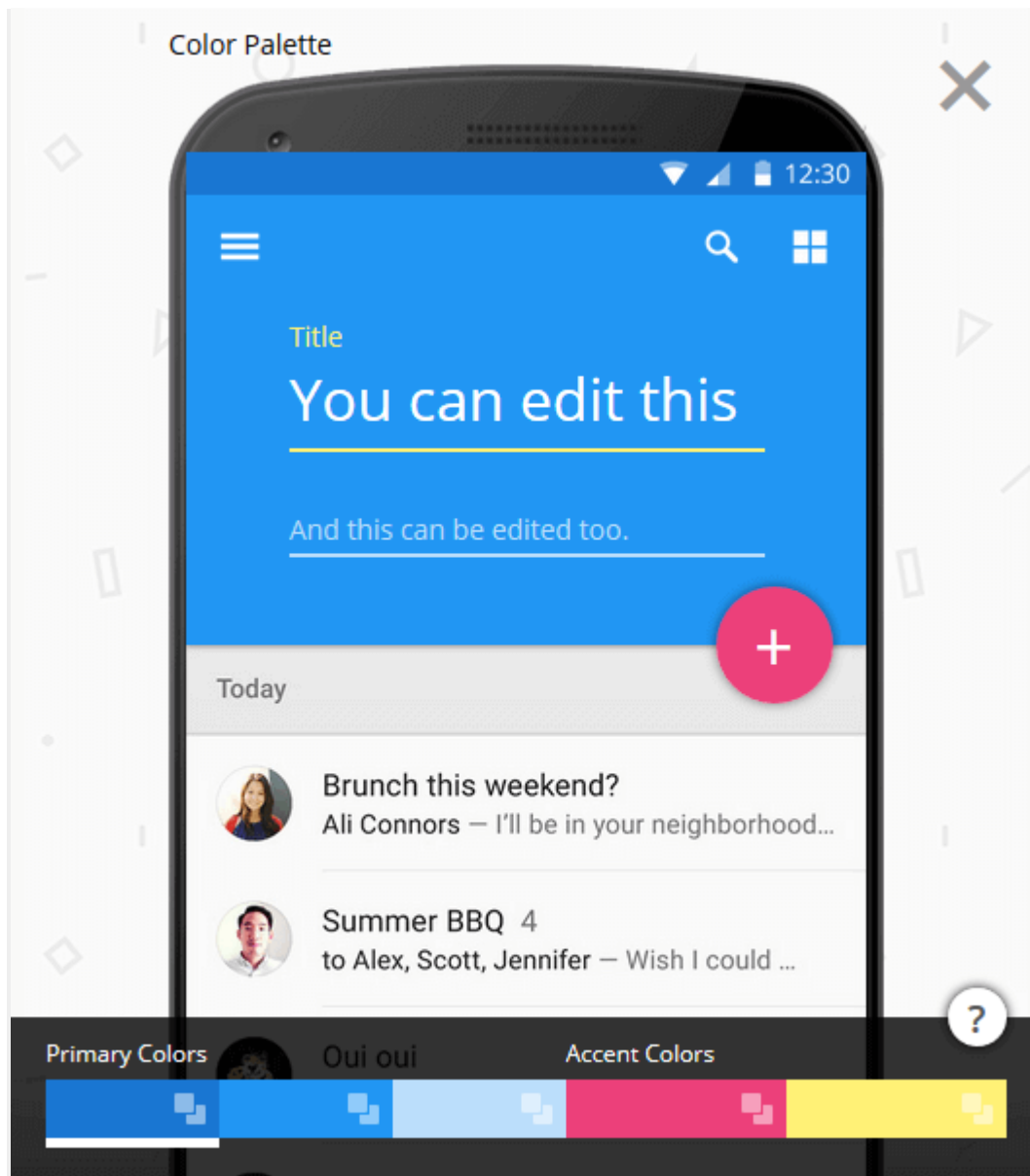
Para obtener estos colores quiero mostrarte una utilidad web que me ha parecido súper interesante. Su nombre es [Material Design Colors](#). Esta herramienta te permite elegir los degradados de los colores que deseas usar en tu app. Incluso te deja ver la previsualización de sus combinaciones.

Red	Pink	Purple	Deep...	Indigo	Blue	Light...	Cyan	Teal	Green	Light...	Lime	Yellow	Amber	Oran...	Deep...	Brown	Grey	Blue ...
30	50	70	75	80	85	90	95	98	100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
200	200	200	200	200	200	200	200	200	200	200	200	200	200	200	200	200	200	200
300	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300	300
400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400	400
500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500	500
600	600	600	600	600	600	600	600	600	600	600	600	600	600	600	600	600	600	600
700	700	700	700	700	700	700	700	700	700	700	700	700	700	700	700	700	700	700
800	800	800	800	800	800	800	800	800	800	800	800	800	800	800	800	800	800	800
900	900	900	900	900	900	900	900	900	900	900	900	900	900	900	900	900	900	900

La anterior imagen muestra los colores definidos de la siguiente forma:

- Primario oscuro: azul 700
- Primario : azul 500
- Realces : azul 100
- Acento : Rosa 400

Con la herramienta «Palette Creator» que se muestra al costado derecho se verá el resultado de esta forma:



Ahora solo debes copiar los colores elegidos en tu archivo `values/colors.xml` como se ve en el siguiente código:  
***colors.xml***

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- Paleta #1 -->
    <color name="azul700">#1976D2</color>
    <color name="azul500">#2196F3</color>
    <color name="azul100">#BBDEFB</color>
```

```
<!-- Paleta #2 -->
<color name="rosa500">#EC407A</color>

</resources>
```

**Paso #3.** Crea los strings a usar en la interfaz. Abre tu archivo **values/strings.xml** y añade las siguientes cadenas.

***strings.xml***

```
<resources>

    <!-- Nombre App -->
    <string name="app_name">People App</string>

    <!-- Titulos actividades -->
    <string name="titulo_actividad_actividad_contactos">Contactos</string>
    <string name="titulo_actividad_insertar_contacto">Añadir nuevo contacto</string>
    <string name="titulo_actividad_editar_contacto">Editar contacto</string>

    <!-- Formulario Actividad inserción -->
    <string name="hint_campo_texto_nombre">Primer nombre</string>
    <string name="hint_campo_texto_apellido">Primer apellido</string>
    <string name="hint_campo_texto_telefono">Teléfono</string>
    <string name="hint_campo_texto_correo">Correo</string>

    <!-- Tooltips acciones toolbar -->
    <string name="accion_sync">Sincronizar</string>
    <string name="accion_eliminar">Eliminar</string>
    <string name="accion_guardar">Guardar</string>
    <string name="accion_descartar">Descartar</string>

    <!-- Cuenta Sync Adapter -->
    <string name="tipo_cuenta">com.herprogramacion.peopleapp.cuenta</string>

    <!-- Autoridad del provider -->
    <string name="autoridad_provider">com.herprogramacion.peopleapp</string>

</resources>
```

Al inicio tenemos el nombre de todas las pantallas que usaremos para tratar los contactos. Luego los textos emergentes para las acciones de la toolbar.

Al final del archivo también tenemos el tipo de cuenta que usaremos en el sync adapter para la sincronización y la autoridad del content provider.

**Paso #4.** Agrega los colores al tema central. Ahora es turno de modificar los estilos que Android Studio nos ha creado por defecto. En este caso asignaremos los colores que creamos en el tema base de la siguiente forma:  
*values/styles.xml*

```
<resources>

    <!-- Tema Base -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/azul500</item>
        <item name="colorPrimaryDark">@color/azul700</item>
        <item name="colorAccent">@color/rosa500</item>

        <item name="windowActionBar">false</item>
        <item name="windowNoTitle">true</item>
    </style>

    <style name="AppTheme.NoActionBar">
        <item name="windowActionBar">false</item>
        <item name="windowNoTitle">true</item>
    </style>

    <style name="AppTheme.AppBarOverlay"
parent="ThemeOverlay.AppCompat.Dark.ActionBar" />
        <style name="AppTheme.PopupOverlay" parent="ThemeOverlay.AppCompat.Light" />

</resources>
```

**Paso #5.** Agrega todas las librerías necesarias para la interfaz, las conexiones de Red y parsing JSON. La configuración parcial del proyecto la termino añadiendo al archivo **build.gradle** las librerías necesarias para desarrollar la app. Entre estas tenemos la librería de diseño para usar **fabs buttons**, **snack bars**, **app bars**, etc. También el **recycler view** para la lista de contactos. La versión no oficial de **volley** para peticiones HTTP y **Gson** para el parsing JSON.  
*build.gradle*

```
dependencies {
    compile fileTree(include: ['*.jar'], dir: 'libs')
```



```
testCompile 'junit:junit:4.12'
compile 'com.android.support:appcompat-v7:23.1.1'
compile 'com.android.support:design:23.1.1'
compile 'com.android.support:recyclerview-v7:23.1.1'
compile 'com.mcxiaoke.volley:library:1.0.19'
compile 'com.google.code.gson:gson:2.4'
}
```

**Paso #6.** Abre tu archivo **values/dimens.xml** y copia el siguiente código si deseas tener las mismas proporciones que usaré en la UI:

***dimens.xml***

```
<resources>
    <!-- Default screen margins, per the Android Design guidelines. -->
    <dimen name="activity_horizontal_margin">16dp</dimen>
    <dimen name="activity_vertical_margin">16dp</dimen>

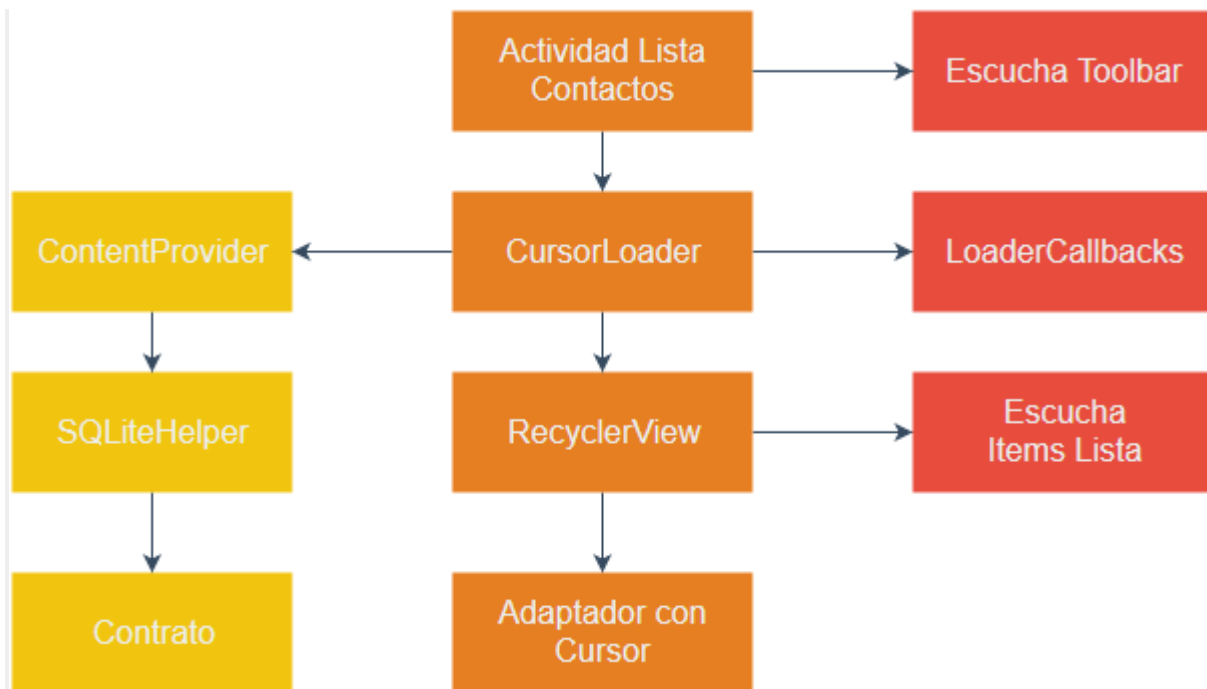
    <dimen name="margenes_fab">16dp</dimen>

    <!-- Listas -->
    <dimen name="altura_item_lista_simple">48dp</dimen>
    <dimen name="tamano_texto_item_lista">16sp</dimen>
    <dimen name="padding_horizontal_item_lista">16dp</dimen>
    <dimen name="padding_superior_item_lista">8dp</dimen>
</resources>
```

## 5.2 Crear lista de contactos

La creación de la lista de contactos implica varios componentes que relacionan la vista con el modelo de datos y a su vez entregan el manejo al usuario.

Para simplificar las tareas de desarrollo que realizaré añadiré el siguiente diagrama con los elementos que necesitaré. Los componentes del tono amarillo pertenecen al modelo, los de tono naranja a la vista y los de tono rojo nos proporcionan controladores de eventos:



Como ves, detrás del reflejo de la lista se encuentran una gran cantidad de componentes que soportan su funcionamiento. El lado bueno es que cada uno de estos ya los hemos estudiado en varios de [los artículos pasados](#), así que no hay nada de qué preocuparse.

Teniendo en cuenta esta arquitectura, comenzaremos por desarrollar los componentes del modelo.

**Paso #1.** Crea el contrato para la base de datos. Añade una nueva clase Java llamada Contrato y añade el siguiente código:

***Contrato.java***

```

public class Contrato {

    interface ColumnasSincronizacion {
        String MODIFICADO = "modificado";
        String ELIMINADO = "eliminado";
        String INSERTADO = "insertado";
    }

    interface ColumnasContacto {
        String ID_CONTACTO = "idContacto"; // Pk
        String PRIMER_NOMBRE = "primerNombre";
        String PRIMER_APELLIDO = "primerApellido";
        String TELEFONO = "telefono";
    }
}

```

```

        String CORREO = "correo";
        String VERSION = "version";
    }

    // Autoridad del Content Provider
    public final static String AUTORIDAD = "com.herprogramacion.peopleapp";

    // Uri base
    public final static Uri URI_CONTENIDO_BASE = Uri.parse("content://" + AUTORIDAD);

    /**
     * Controlador de la tabla "contacto"
     */
    public static class Contactos
        implements BaseColumns, ColumnasContacto, ColumnasSincronizacion {

        public static final Uri URI_CONTENIDO =
            URI_CONTENIDO_BASE.buildUpon().appendPath(RECURSO_CONTACTO).build();

        public final static String MIME_RECURSO =
            "vnd.android.cursor.item/vnd." + AUTORIDAD + "/" + RECURSO_CONTACTO;

        public final static String MIME_COLECCION =
            "vnd.android.cursor.dir/vnd." + AUTORIDAD + "/" + RECURSO_CONTACTO;

        /**
         * Construye una {@link Uri} para el {@link #ID_CONTACTO} solicitado.
         */
        public static Uri construirUriContacto(String idContacto) {
            return URI_CONTENIDO.buildUpon().appendPath(idContacto).build();
        }

        public static String generarIdContacto() {
            return "C-" + UUID.randomUUID();
        }

        public static String obtenerIdContacto(Uri uri) {
            return uri.getLastPathSegment();
        }
    }

```

```

    }
}

// Recursos
public final static String RECURSO_CONTACTO = "contactos";

}

```

En primera instancia tenemos dos interfaces con los nombres de las columnas que existirán en la tabla contacto. ColumnasSincronizacion contiene tres banderas para determinar si los registros locales han sido insertados, modificados o eliminados. Donde el valor 1 determina el alza de la bandera y 0 la ausencia de cambio. A su vez defino la autoridad del content provider con mi paquete java y de una vez la uri de contenido general.

Al final tendremos una clase controladora para el recurso «**contactos**». Su objetivo es proporcionar información sobre la uri de contenido, los tipos mime y la construcción de uris para soportar identificadores.

Adicionalmente tiene un método llamado generarIdContacto(), el cual se encarga de crear una nueva id para los nuevos contactos locales, basada en la sintaxis 'C-UUID'. Donde [UUID es un identificador único](#) de 16 bytes obtenido con la clase UUID. Muestra: «C-54fba0b9-0fe6-4edf-86a4-1e5f0f561faf»

Si estás perdido en este punto, entonces te recomiendo leer mi artículo para [crear un content provider personalizado](#).

**Paso #2.** Crea la clase auxiliar para el acceso a bases de datos. En este paso debes crear una nueva clase que extienda de SQLiteOpenHelper y sobrescribir los controladores onCreate() y onUpgrade() para determinar las acciones al crear la base de datos y al actualizar su versión.

***HelperContactos.java***

```

public class HelperContactos extends SQLiteOpenHelper {

    static final int VERSION = 1;

    static final String NOMBRE_BD = "people_app.db";

    interface Tablas {
        String CONTACTO = "contacto";
    }
}

```

```

public HelperContactos(Context context) {
    super(context, NOMBRE_BD, null, VERSION);
}

@Override
public void onCreate(SQLiteDatabase db) {
    db.execSQL(
        "CREATE TABLE " + Tablas.CONTACTO + "("
        + Contactos._ID + " INTEGER PRIMARY KEY AUTOINCREMENT,"
        + Contactos.ID_CONTACTO + " TEXT UNIQUE,"
        + Contactos.PRIMER_NOMBRE + " TEXT NOT NULL,"
        + Contactos.PRIMER_APELLIDO + " TEXT,"
        + Contactos.TELEFONO + " TEXT,"
        + Contactos.CORREO + " TEXT,"
        + Contactos.VERSION + " DATE DEFAULT CURRENT_TIMESTAMP,"
        + Contactos.INSERTADO + " INTEGER DEFAULT 1,"
        + Contactos.MODIFICADO + " INTEGER DEFAULT 0,"
        + Contactos.ELIMINADO + " INTEGER DEFAULT 0)");
    }

@Override
public void onUpgrade(SQLiteDatabase db, int i, int i1) {
    try {
        db.execSQL("DROP TABLE IF EXISTS " + Tablas.CONTACTO);
    } catch (SQLException e) {
        // Manejo de excepciones
    }
    onCreate(db);
}
}

```

Recuerda incluir siempre la columna "\_id" manualmente o su equivalente en BaseColumns.\_ID. El framework de Android lo necesita en varios componentes para un correcto funcionamiento.

Es importante que añadas un índice UNIQUE a la columna idContacto para establecer restricciones de **unicidad** entre los contactos con base a la base de datos Mysql. En cuanto a las columnas de sincronización, declara valores por defecto con DEFAULT para dejarlas inactivas al crear un nuevo registro. "insertado" puedes dejarla levantada por defecto, ya que una inserción local provoca este estado. Sin

embargo, cuando la inserción venga por producto de la sincronización, cambia el valor a `0` en la sentencia `INSERT`.

Si tienes dudas sobre la creación de bases de datos SQLite, entonces lee mi artículo [Tutorial De Bases De Datos SQLite En Android](#).

**Paso #3.** Crea un Content Provider personalizado para operar los datos a través de URIs.

Para la tabla contactos tenemos dos formas de URI a la cual referimos dependiendo si nos referimos a todos los registros (colección) o a un contacto específico (recurso).

```
content://com.herprogramacion.peopleapp/contactos
```

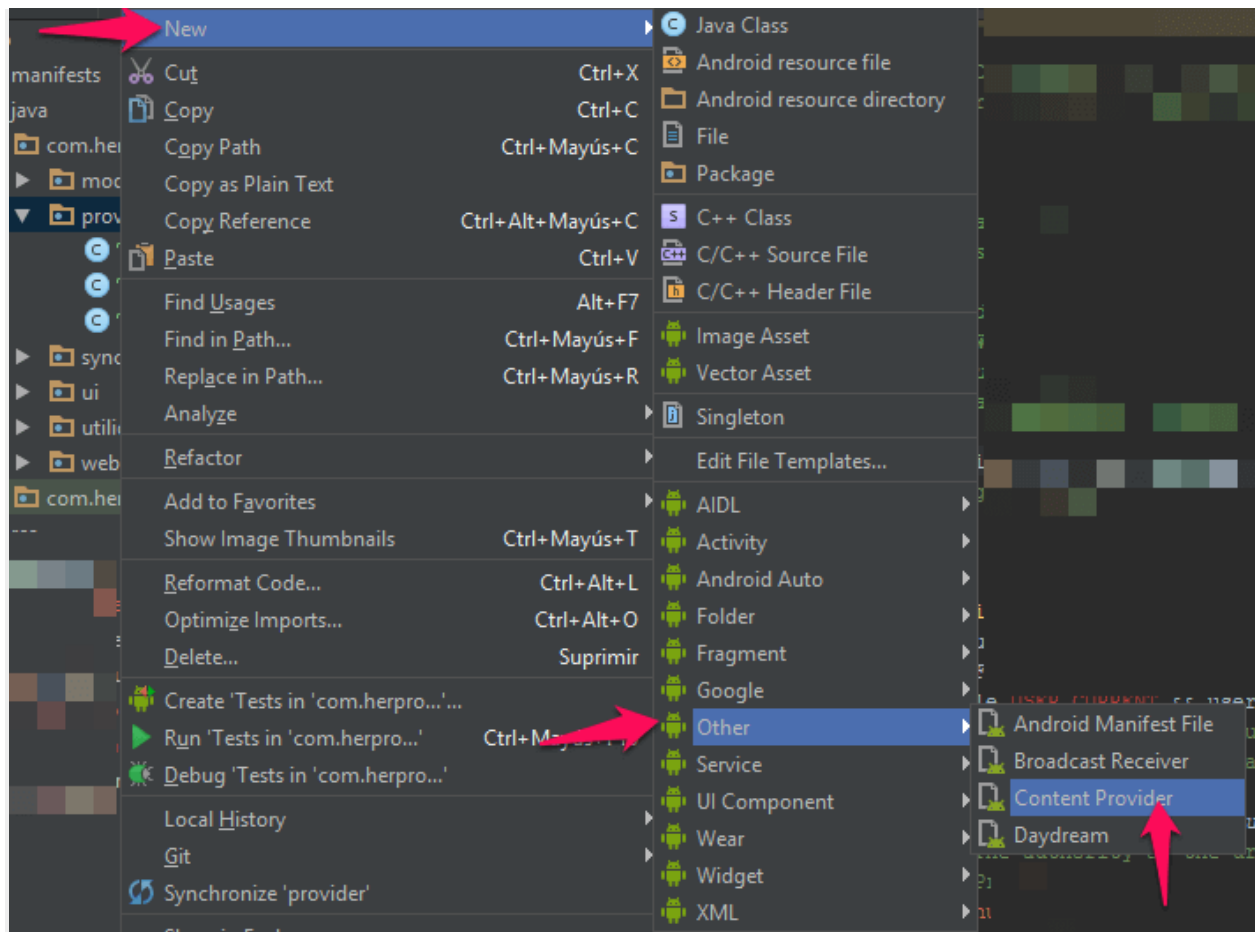
```
content://com.herprogramacion.peopleapp/contactos/:id
```

Para diferenciar estas dos estructuras recuerda que la clase `UriMatcher` provee la asociación de códigos únicos con patrones de uris. Así que si te basas en este componente, la sobrescritura de los

métodos `query()`, `insert()`, `update()`, `delete()`, `bulkInsert()` o `applyBatch()`.

Teniendo en cuenta estas recomendaciones, ya puedes crear el content provider.

Una de las formas de hacerlo es a través de la plantilla de Android Studio al presionar click derecho en el paquete y elegir **New > Other > Content Provider**. Con ello no tendrás que añadir la declaración del componente en el **AndroidManifest.xml**.



El nombre del archivo será **ProviderContactos.java**. Una vez que tengas la plantilla, agrega la siguiente definición de los controladores:

### ***ProviderContactos.java***

```
/**
 * {@link ContentProvider} que encapsula el acceso a la base de datos de contactos
 */
public class ProviderContactos extends ContentProvider {

    // Comparador de URIs de contenido
    public static final UriMatcher uriMatcher;

    // Identificadores de tipos
    public static final int CONTACTOS = 100;
    public static final int CONTACTOS_ID = 101;

    static {
        uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    }
}
```

```

        uriMatcher.addURI(Contrato.AUTORIDAD, "contactos", CONTACTOS);
        uriMatcher.addURI(Contrato.AUTORIDAD, "contactos/*", CONTACTOS_ID);
    }

    private HelperContactos manejadorBD;
    private ContentResolver resolver;

    @Override
    public boolean onCreate() {
        manejadorBD = new HelperContactos(getContext());
        resolver = getContext().getContentResolver();
        return true;
    }

    @Override
    public String getType(Uri uri) {
        switch (uriMatcher.match(uri)) {
            case CONTACTOS:
                return Contactos.MIME_COLECCION;
            case CONTACTOS_ID:
                return Contactos.MIME_RECURSO;
            default:
                throw new IllegalArgumentException("Tipo desconocido: " + uri);
        }
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
                        String[] selectionArgs, String sortOrder) {
        // Obtener base de datos
        SQLiteDatabase db = manejadorBD.getWritableDatabase();
        // Comparar Uri
        int match = uriMatcher.match(uri);

        Cursor c;

        switch (match) {
            case CONTACTOS:
                // Consultando todos los registros
                c = db.query(Tablas.CONTACTO, projection,

```



```

        selection, selectionArgs,
        null, null, sortOrder);
    c.setNotificationUri(resolver, Contactos.URI_CONTENIDO);
    break;
case CONTACTOS_ID:
    // Consultando un solo registro basado en el Id del Uri
    String idContacto = Contactos.obtenerIdContacto(uri);
    c = db.query(Tablas.CONTACTO, projection,
        Contactos.ID_CONTACTO + "=" + "'" + idContacto + "'"
            + (!TextUtils.isEmpty(selection) ?
                " AND (" + selection + ')': ""),
        selectionArgs, null, null, sortOrder);
    c.setNotificationUri(resolver, uri);
    break;
default:
    throw new IllegalArgumentException("URI no soportada: " + uri);
}
return c;
}

@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {
    SQLiteDatabase db = manejadorBD.getWritableDatabase();

    int match = uriMatcher.match(uri);
    int filasAfectadas;

    switch (match) {
        case CONTACTOS:

            filasAfectadas = db.delete(Tablas.CONTACTO,
                selection,
                selectionArgs);

            resolver.notifyChange(uri, null, false);

            break;
        case CONTACTOS_ID:

            String idContacto = Contactos.obtenerIdContacto(uri);

```

```

        filasAfectadas = db.delete(Tablas.CONTACTO,
            Contactos.ID_CONTACTO + "=" + "'" + idContacto + "'"
            + (!TextUtils.isEmpty(selection) ?
                " AND (" + selection + ')': ""),
            selectionArgs);

        resolver.notifyChange(uri, null, false);
        break;
    default:
        throw new IllegalArgumentException("Contacto desconocido: " +
            uri);
    }
    return filasAfectadas;
}

@Override
public Uri insert(Uri uri, ContentValues values) {
    // Validar la uri
    if (uriMatcher.match(uri) != CONTACTOS) {
        throw new IllegalArgumentException("URI desconocida : " + uri);
    }

    ContentValues contentValues;
    if (values != null) {
        contentValues = new ContentValues(values);
    } else {
        contentValues = new ContentValues();
    }

    // Inserción de nueva fila
    SQLiteDatabase db = manejadorBD.getWritableDatabase();

    long _id = db.insert(Tablas.CONTACTO, null, contentValues);

    if (_id > 0) {

        resolver.notifyChange(uri, null, false);

        String idContacto = contentValues.getAsString(Contactos.ID_CONTACTO);

        return Contactos.construirUriContacto(idContacto);
    }
}

```

```

    }

    throw new SQLException("Falla al insertar fila en : " + uri);
}

@Override
public int update(Uri uri, ContentValues values, String selection,
                  String[] selectionArgs) {

    SQLiteDatabase db = manejadorBD.getWritableDatabase();

    int filasAfectadas;

    switch (uriMatcher.match(uri)) {
        case CONTACTOS:

            filasAfectadas = db.update(Tablas.CONTACTO, values,
                                      selection, selectionArgs);

            resolver.notifyChange(uri, null, false);

            break;
        case CONTACTOS_ID:

            String idContacto = Contactos.obtenerIdContacto(uri);

            filasAfectadas = db.update(Tablas.CONTACTO, values,
                                      Contactos.ID_CONTACTO + "=" + "'" + idContacto + "'"
                                      + (!TextUtils.isEmpty(selection) ?
                                       " AND (" + selection + ')': "" ),
                                      selectionArgs);

            resolver.notifyChange(uri, null, false);

            break;
        default:
            throw new IllegalArgumentException("URI desconocida: " + uri);
    }
}

```

```

        return filasAfectadas;
    }
}

```

#### *Puntos a resaltar:*

- El código para un acceso a todos los contactos es 100 y el de accesos individuales es 101.
- Llama el método `notifyChange()` del `ContentResolver` para alertar al framework que el contenido de cada uri cambió.
- Si deseas que los adaptadores se actualicen automáticamente después de una consulta en `query()`, llama al método `setNotificationUri()` del cursor obtenido. Este recibe una instancia del content resolver y la uri a la que se le hará seguimiento.

**Paso #4.** En este punto comenzaremos a crear los componentes de la interfaz. El primero que tenemos que reformular es la actividad de lista de contactos. Aunque Android Studio nos entregó un [layout con App Bar](#) y un fab button, aún falta [añadir un RecyclerView para crear la lista](#). El layout de la actividad de contactos debe haberse con la siguiente estructura:

#### ***actividad\_lista\_contactos.xml***

```

<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/coordinador"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true">

    <android.support.design.widget.AppBarLayout
        android:id="@+id/appbar"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/AppTheme.AppBarOverlay">

        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/AppTheme.PopupOverlay" />

```

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

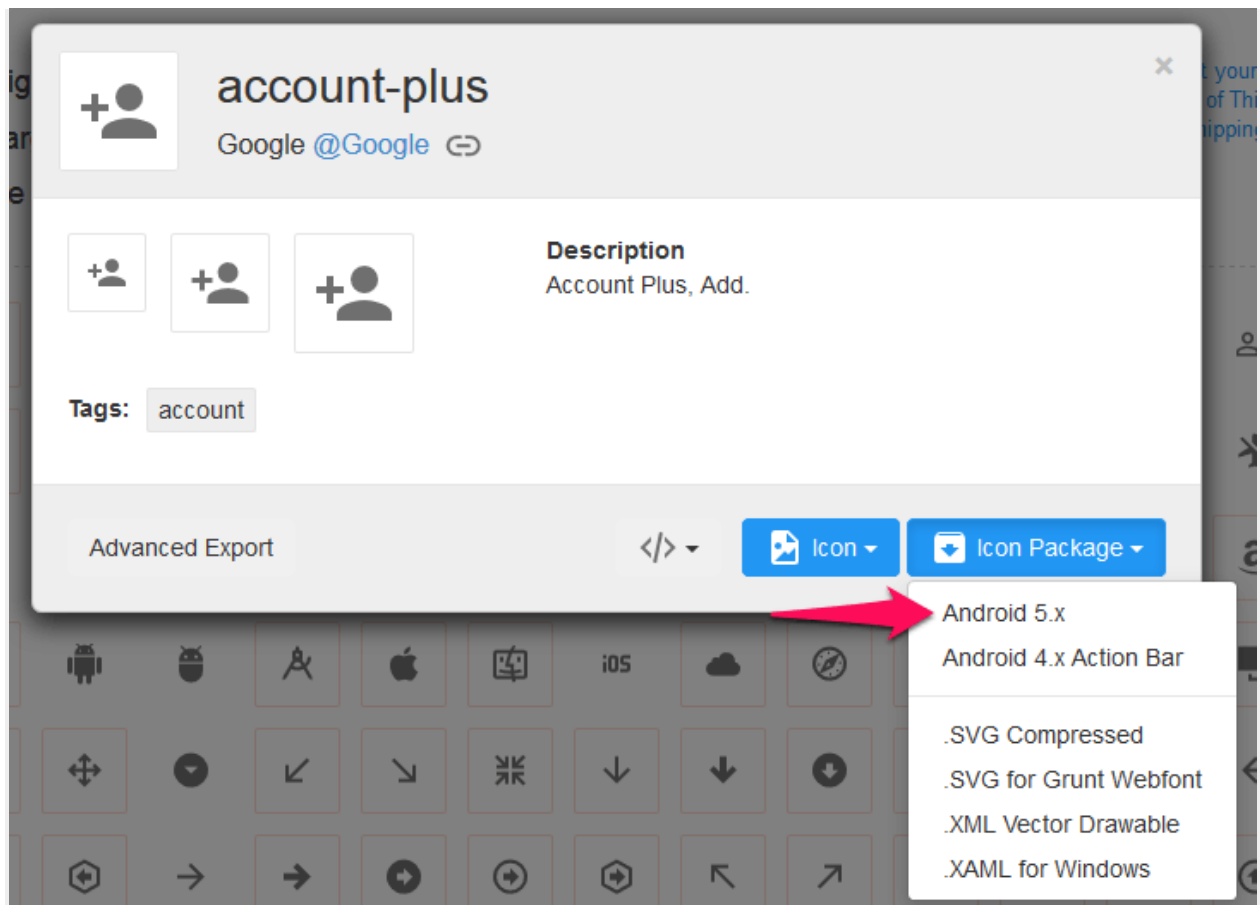
<include layout="@layout/contenido_actividad_lista_contactos" />

<android.support.design.widget.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom|end"
    android:layout_margin="@dimen/margenes_fab"
    android:src="@drawable/icono_nuevo_contacto" />

</android.support.design.widget.CoordinatorLayout>
```

La etiqueta <include> la está generando Android Studio para extraer el contenido principal de la actividad en otro layout complementario, que permite mejorar la visualización y el orden.

El layout general lo dejaremos quieto. Solo modifica el icono del fab button con un icono de añadir contacto. Para descargarlo puedes ir a la herramienta [Material Design Icons](#) y presionar **Icon Package > Android 5.x**.



Ahora abre el segundo layout de contenido. Por lo general se su nombre empezará con «**content\_**». Reemplaza su contenido con un RecyclerView como nodo raíz para establecer nuestra lista.

***contenido\_actividad\_lista\_contactos.xml***

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.RecyclerView
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/reciclador"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingTop="@dimen/padding_superior_item_lista"
    app:layout_behavior="@string/appbar_scrolling_view_behavior" />
```

**Paso #5.** Añadir adaptador personalizado con soporte de cursor. El recycler será poblado por los datos que tengamos en la tabla contacto, por ende es necesario usar un cursor al adaptador que creemos.

El adaptador debes construirlo de la misma en que haces siempre, es decir, sobrescribiendo los métodos `getItemCount()`, `onCreateViewHolder()` y `onBindViewHolder()`. Solo que esta vez el contenido será una instancia tipo `Cursor`, por lo que debes usar los métodos del cursor para obtener los datos y ligarlos a la interfaz.

Veamos:

### ***AdaptadorContactos.java***

```
/**
 * Adaptador para la lista de contactos
 */
public class AdaptadorContactos extends
RecyclerView.Adapter<AdaptadorContactos.ViewHolder> {
    private Cursor items;

    // Instancia de escucha
    private OnItemClickListener escucha;

    /**
     * Interfaz para escuchar clicks del recycler
     */
    interface OnItemClickListener {
        public void onClick(ViewHolder holder, String idContacto);
    }

    public class ViewHolder extends RecyclerView.ViewHolder
        implements View.OnClickListener {
        // Campos respectivos de un item
        public TextView nombre;

        public ViewHolder(View v) {
            super(v);
            nombre = (TextView) v.findViewById(R.id.nombre_contacto);
            v.setOnClickListener(this);
        }

        @Override
        public void onClick(View view) {
            escucha.onClick(this, obtenerIdContacto(getAdapterPosition()));
        }
    }
}
```

```

private String obtenerIdContacto(int posicion) {
    if (items != null) {
        if (items.moveToPosition(posicion)) {
            return UConsultas.obtenerString(items, Contactos.ID_CONTACTO);
        } else {
            return null;
        }
    } else {
        return null;
    }
}

public AdaptadorContactos(OnItemClickListener escucha) {
    this.escucha = escucha;
}

@Override
public ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    View v = LayoutInflater.from(parent.getContext())
        .inflate(R.layout.item_contacto, parent, false);
    return new ViewHolder(v);
}

@Override
public void onBindViewHolder(ViewHolder holder, int position) {

    items.moveToPosition(position);

    String primerNombre;
    String primerApellido;

    primerNombre = UConsultas.obtenerString(items, Contactos.PRIMER_NOMBRE);
    primerApellido = UConsultas.obtenerString(items, Contactos.PRIMER_APELLIDO);

    holder.nombre.setText(String.format("%s %s", primerNombre, primerApellido));
}

@Override
public int getItemCount() {

```



```

        if (items != null)
            return items.getCount();
        return 0;
    }

    public void swapCursor(Cursor nuevoCursor) {
        if (nuevoCursor != null) {
            items = nuevoCursor;
            notifyDataSetChanged();
        }
    }

    public Cursor getCursor() {
        return items;
    }
}

```

Si observas el código notarás lo siguiente:

- El método `getItemCount()` se basa en el resultado de `size()` del cursor `items`.
- Dentro de `onBindViewHolder()` usamos `moveToPosition()` en el cursor para apuntar al registro actual. Luego se obtiene el nombre y apellido del contacto (como se vio en el wireframe) con el método de utilidad `UConsultas.obtenerString()`, para asignarlo a cada ítem de la lista.
- Usaremos un método `swapCursor()` para intercambiar los datos del cursor actual con uno nuevo.
- Relacionaremos la columna `'idContacto'` con la posición dentro del adaptador con el método `obtenerIdContacto()`. Esto con el fin de relacionar ambos valores de forma directa.

Ahora te preguntará, *¿cómo funciona la escucha declarada dentro del recycler?*

Sencillo. Debido a que el view holder representa cada fila en forma de view, entonces convertimos a este elemento en una escucha del tipo `View.OnClickListener`.

Lo siguiente es crear una interfaz que represente al elemento que escuchará lo que escuchó el view holder de forma secuencial. Esta es la interfaz `OnItemClickListener` la cual tendrá análogamente un controlador `onClick()` que recibe el view holder y a `idContacto`.

Como ves, el controlador `OnClickListener.onClick()` en el view holder llama a su vez al controlador `OnItemClickListener.onClick()` con los valores actuales.

*¿Y quién será la escucha?*

*¡La actividad de contactos!*

Por eso el constructor del RecyclerView tiene como parámetro aquel componente que actuará como escucha. Así que cuando crees una nueva instancia de este, el parámetro será la actividad pasando el operador `this`.

– **Layout del item de lista:** En el método se ve una referencia de inflado a un layout que se llama `item_contacto.xml`. Su contenido es solo un `TextView` sencillo como se muestra a continuación:

***item\_contacto.xml***

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

<TextView xmlns:android="http://schemas.android.com/apk/res/android"

    android:id="@+id/nombre_contacto"

    android:layout_width="match_parent"

    android:layout_height="match_parent"

    android:background="?attr/selectableItemBackground"

    android:text="Nombre Contacto" />
```

Si en este momento ejecutara la app, la lista no me mostraría nada porque no hay registros. Por ello agregaremos 2 registros de pruebas dentro de `HelperContactos.java` en `onCreate()`.

```
@Override
public void onCreate(SQLiteDatabase db) {
    db.execSQL(
        "CREATE TABLE " + Tablas.CONTACTO + "("
            + Contactos._ID + " INTEGER PRIMARY KEY AUTOINCREMENT,"
            + Contactos.ID_CONTACTO + " TEXT UNIQUE,"
            + Contactos.PRIMER_NOMBRE + " TEXT NOT NULL,"
            + Contactos.PRIMER_APELLIDO + " TEXT,"
            + Contactos.TELEFONO + " TEXT,"
            + Contactos.CORREO + " TEXT,"
            + Contactos.VERSION + " DATE DEFAULT CURRENT_TIMESTAMP,"
            + Contactos.INSERTADO + " INTEGER DEFAULT 1,"
            + Contactos.MODIFICADO + " INTEGER DEFAULT 0,"
            + Contactos.ELIMINADO + " INTEGER DEFAULT 0)");

    // Registro ejemplo #1
    ContentValues valores = new ContentValues();
```

```

valores.put(Contactos.ID_CONTACTO, Contactos.generarIdContacto());
valores.put(Contactos.PRIMER_NOMBRE, "Roberto");
valores.put(Contactos.PRIMER_APELLIDO, "Gomez");
valores.put(Contactos.TELEFONO, "4444444");
valores.put(Contactos.CORREO, "robertico@mail.com");
valores.put(Contactos.VERSION, UTiempo.obtenerTiempo());

db.insertOrThrow(Tablas.CONTACTO, null, valores);

// Registro ejemplo #2
valores.clear();
valores.put(Contactos.ID_CONTACTO, Contactos.generarIdContacto());
valores.put(Contactos.PRIMER_NOMBRE, "Pablo");
valores.put(Contactos.PRIMER_APELLIDO, "Catatumbo");
valores.put(Contactos.TELEFONO, "5555555");
valores.put(Contactos.CORREO, "pablito@mail.com");
valores.put(Contactos.VERSION, UTiempo.obtenerTiempo());
db.insertOrThrow(Tablas.CONTACTO, null, valores);
}

```

## Paso #6. Prepara la actividad de lista de contactos.

Hay varias cosas por hacer si queremos mostrar nuestra lista poblada.

- Implemente las interfaces `LoaderCallbacks` y `AdaptadorContactos.OnItemClickListener`.
  - Prepara la lista obteniendo el `recycler` y asignado un `LayoutManager` y el adaptador.
  - Inicia un nuevo `Loader` y consulta todos los contactos en `onCreateLoader()`. Luego usa el método `swapCursor()` del adaptador en `onLoadFinished()` para actualizar los datos.
  - Escribe la declaración del método `onClick()` de la interfaz del adaptador.
  - Guarda la clave de API para algunos de los usuarios que tienes en la base de datos en tu archivo de preferencias. Luego usaremos la clave para enviar peticiones con *Volley*. Si descargas el código completo podrás ver la clase `UPreferencias`, donde tengo un método prefabricado para guardar y obtener esta clave.
- Leyendo las tareas anteriores puedes comenzar a editar tu actividad. O guíate por el siguiente código:

***ActividadListaContactos.java***

```

public class ActividadListaContactos extends AppCompatActivity
    implements LoaderManager.LoaderCallbacks<Cursor>,
    AdaptadorContactos.OnItemClickListener {

    private static final String TAG = ActividadListaContactos.class.getSimpleName();

    private RecyclerView reciclador;
    private LinearLayoutManager layoutManager;
    private AdaptadorContactos adaptador;

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

        // Agregar toolbar
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
        setTitle(R.string.titulo_actividad_lista_contactos);

        FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);
        fab.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                // Acciones
            }
        });

        prepararLista();

        getSupportLoaderManager().restartLoader(1, null, this);

        // Reemplaza con tu clave
        UPreferencias.guardarClaveApi(this, "60d5b4e60cb6a70898f0cd17174e9edd");
    }

    private void prepararLista() {
        reciclador = (RecyclerView) findViewById(R.id.reciclador);
        layoutManager = new LinearLayoutManager(this);
    }
}

```

```

        adaptador = new AdaptadorContactos(this);

        reciclador.setLayoutManager(layoutManager);
        reciclador.setAdapter(adaptador);
    }

    @Override
    public Loader<Cursor> onCreateLoader(int id, Bundle args) {
        return new CursorLoader(
            this,
            Contactos.URI_CONTENIDO,
            null, Contactos.ELIMINADO + "=?", new String[]{"0"}, null);
    }

    @Override
    public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
        adaptador.swapCursor(data);
    }

    @Override
    public void onLoaderReset(Loader<Cursor> loader) {
        adaptador.swapCursor(null);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.menu_lista_contactos, menu);
        return true;
    }

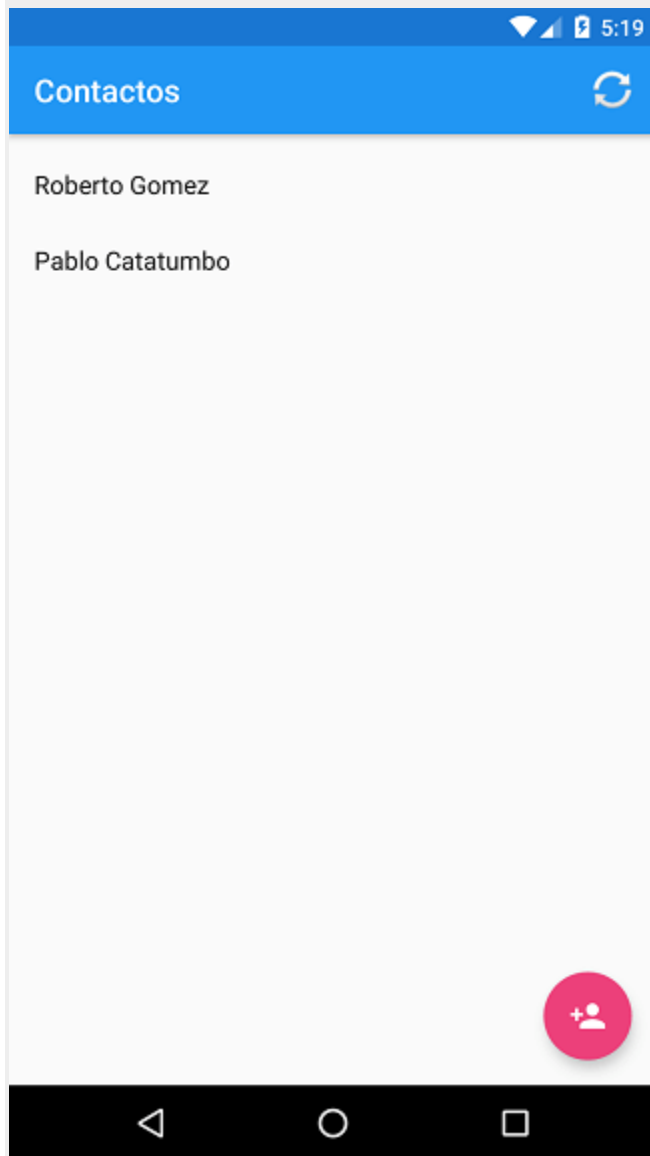
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        int id = item.getItemId();
        if (id == R.id.accion_sync) {
            // Sincronización
            return true;
        }

        return super.onOptionsItemSelected(item);
    }

```

```
@Override
public void onClick(AdaptadorContactos.ViewHolder holder, String idContacto) {
    // Acciones
}
}
```

Ejecuta y verás la siguiente imagen:



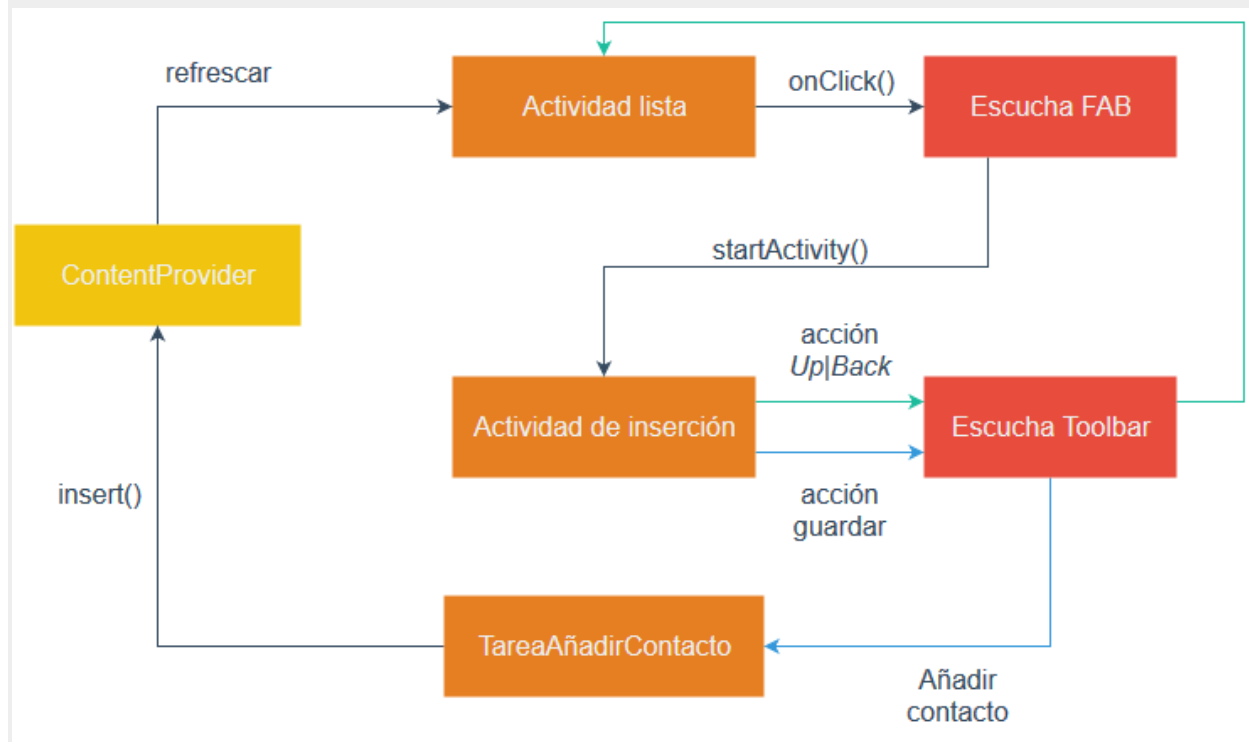
Este sería nuestro primer resultado parcial.

## 5.3 Crear Actividad De Inserción

Como viste en el wireframe, insertar un contacto se basa en una nueva pantalla con 4 campos de texto para guardar el primer nombre, primer apellido, teléfono y correo.

La acción se desprende desde el momento en que el usuario presiona el fab button de agregar un contacto. Luego el usuario diligencia los campos y decide si guarda los elementos o no.

Observa el siguiente diagrama de la arquitectura para comprender un poco mejor esta situación.



Nota que si la el icono de confirmación en la toolbar es presionado, entonces se inicia una tarea asíncrona llamada **TareaAñadirContacto**, dentro de la cual usaremos el comando **ContentResolver.insert()** para añadir los datos del formulario a la tabla contacto. Esto evitará alterar el hilo principal con la inserción.

Con este esquema global en la mente comenzaré a codificar los componentes.

**Paso #1.** Crea una nueva actividad en blanco llamada **ActividadInsercionContacto**. Luego modifica el layout eliminado el fab que no usaremos. Después modela el wireframe de inserción que vimos al inicio.

***actividad\_insercion\_contacto.xml***

```

<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools" android:layout_width="match_parent"
    android:layout_height="match_parent" android:fitsSystemWindows="true"
    tools:context=".ui.ActividadInsercionContacto">

    <android.support.design.widget.AppBarLayout android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:theme="@style/AppTheme.AppBarOverlay">

        <android.support.v7.widget.Toolbar android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/AppTheme.PopupOverlay" />

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

    <include layout="@layout/contenido_actividad_insercion_contacto" />

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

```

Para el contenido son 4 campos de textos con sus respectivas etiquetas y además iconos significativos de cada elemento. A mi manera de ver un [Grid Layout](#) es buena elección para esta situación, ya que me permite ubicar todos los views de forma simétrica y ocupando espacios expandidos cuando se requiera. Así que los distribuiré de la siguiente forma:

#### ***contenido\_actividad\_insercion\_contacto.xml***

```

<?xml version="1.0" encoding="utf-8"?>
<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:columnCount="2"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"

```



```

        android:paddingRight="@dimen/activity_horizontal_margin"
        android:paddingTop="@dimen/activity_vertical_margin"
        android:rowCount="4"
        app:layout_behavior="@string/appbar_scrolling_view_behavior"
        tools:context=".ui.ActividadInsercionContacto"
        tools:showIn="@layout/actividad_insercion_contacto">

<ImageView
    android:id="@+id/icono_nombre"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_column="0"
    android:layout_gravity="center_vertical"
    android:layout_row="0"
    android:src="@drawable/icono_cuenta" />

<android.support.design.widget.TextInputLayout
    android:id="@+id/mascara_campo_nombre"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_column="1"
    android:layout_gravity="center_horizontal"
    android:layout_row="0">

    <EditText
        android:id="@+id/campo_primer_nombre"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:ems="10"
        android:hint="@string/hint_campo_texto_nombre"
        android:inputType="textPersonName" />
</android.support.design.widget.TextInputLayout>

<android.support.design.widget.TextInputLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_column="1"
    android:layout_gravity="center"
    android:layout_marginBottom="@dimen/activity_vertical_margin"
    android:layout_row="1">

```

```

<EditText
    android:id="@+id/campo_primer_apellido"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:ems="10"
    android:hint="@string/hint_campo_texto_apellido"
    android:inputType="textPersonName" />
</android.support.design.widget.TextInputLayout>

<ImageView
    android:id="@+id/icono_telefono"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_column="0"
    android:layout_gravity="center_vertical"
    android:layout_marginBottom="@dimen/activity_vertical_margin"
    android:layout_row="2"
    android:src="@drawable/icono_telefono" />

<android.support.design.widget.TextInputLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_column="1"
    android:layout_gravity="center_horizontal"
    android:layout_marginBottom="@dimen/activity_vertical_margin"
    android:layout_row="2">

    <EditText
        android:id="@+id/campo_telefono"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:ems="10"
        android:hint="@string/hint_campo_texto_telefono"
        android:inputType="phone" />
    </android.support.design.widget.TextInputLayout>

<ImageView
    android:id="@+id/icono_correo"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"

```

```

        android:layout_column="0"
        android:layout_gravity="center_vertical"
        android:layout_row="3"
        android:src="@drawable/icono_correo" />

<android.support.design.widget.TextInputLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_column="1"
    android:layout_gravity="center_horizontal"
    android:layout_row="3">

    <EditText
        android:id="@+id/campo_correo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:ems="10"
        android:hint="@string/hint_campo_texto_correo"
        android:inputType="textEmailAddress" />

    </android.support.design.widget.TextInputLayout>
</GridLayout>

```

Ahora modifica el archivo de menú y agrega un action button para la confirmación de la siguiente forma:

### ***menu\_insercion\_contacto.xml***

```

<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <item
        android:id="@+id/accion_confirmar"
        android:icon="@drawable/icono_ok"
        android:title="@string/accion_guardar"
        app:showAsAction="ifRoom" />
    </item>
</menu>

```

Con estos movimientos ya dispones de los elementos visuales que activarán la inserción.

**Paso #2.** Modifica la actividad de inserción para que recolecte los datos asignados por el usuario en cada EditText. Cada uno de estos deben ser añadidos a

un ContentValues que se pasará a la tarea asíncrona, la cual usará al content resolver para insertar estos datos.

### ***ActividadInsercionContacto.java***

```
public class ActividadInsercionContacto extends AppCompatActivity{

    // Referencias UI
    private EditText campoPrimerNombre;
    private EditText campoPrimerApellido;
    private EditText campoTelefono;
    private EditText campoCorreo;

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

        agregarToolBar();

        // Encontrar Referencias UI
        campoPrimerNombre = (EditText) findViewById(R.id.campo_primer_nombre);
        campoPrimerApellido = (EditText) findViewById(R.id.campo_primer_apellido);
        campoTelefono = (EditText) findViewById(R.id.campo_telefono);
        campoCorreo = (EditText) findViewById(R.id.campo_correo);

    }

    private void agregarToolBar() {
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
        if (getSupportActionBar() != null) {
            getSupportActionBar().setDisplayHomeAsUpEnabled(true);
        }
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.menu_insercion_contacto, menu);
        return true;
    }
}
```

```

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    switch (id) {
        case R.id.accion_confirmar:
            insertar();
            break;
    }
    return super.onOptionsItemSelected(item);
}

private void insertar() {

    // Extraer datos de UI
    String primerNombre = campoPrimerNombre.getText().toString();
    String primerApellido = campoPrimerApellido.getText().toString();
    String telefono = campoTelefono.getText().toString();
    String correo = campoCorreo.getText().toString();

    // Validaciones y pruebas de cordura
    if (!esNombreValido(primerNombre)) {
        TextInputLayout mascaraCampoNombre = (TextInputLayout)
findViewById(R.id.mascara_campo_nombre);
        mascaraCampoNombre.setError("Este campo no puede quedar vacío");
    } else {

        ContentValues valores = new ContentValues();

        valores.put(Contactos.ID_CONTACTO, Contactos.generarIdContacto());
        valores.put(Contactos.PRIMER_NOMBRE, primerNombre);
        valores.put(Contactos.PRIMER_APELLIDO, primerApellido);
        valores.put(Contactos.TELEFONO, telefono);
        valores.put(Contactos.CORREO, correo);
        valores.put(Contactos.VERSION, UTiempo.obtenerTiempo());

        // Iniciar inserción|actualización
        new TareaAnadirContacto(getContentResolver(),
valores).execute(uriContacto);
    }
}

```

```

        finish();
    }
}

private boolean esNombreValido(String nombre) {
    return !TextUtils.isEmpty(nombre);
}

static class TareaAnadirContacto extends AsyncTask<Void, Void, Void> {
    private final ContentResolver resolver;
    private final ContentValues valores;

    public TareaAnadirContacto(ContentResolver resolver, ContentValues valores) {
        this.resolver = resolver;
        this.valores = valores;
    }

    @Override
    protected Void doInBackground(Void... args) {
        resolver.insert(Contactos.URI_CONTENIDO, valores);
        return null;
    }
}
}
}

```

Si aún desconoces el uso de tareas asíncronas, entonces este artículo te ayudará: [Tareas Asíncronas En Android](#).

Cabe aclarar que solo puse una validación de datos con respecto al primer nombre. Lo que hago es comprobar si el contenido del EditText está vacío a través del método esNombreValido().

Si no lo es, entonces obtengo la referencia de su InputTextLayout para asignarle un mensaje de error con setError(). De lo contrario procederemos a iniciar la inserción.

**Paso #3.** Ahora dentro del controlador de clicks para fab que tienes en ActividadListaContactos, inicia la actividad de inserción.

```

FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);
fab.setOnClickListener(new View.OnClickListener() {

```

```

@Override
public void onClick(View view) {
    Intent intent = new Intent(ActividadListaContactos.this,
ActividadInsercionContacto.class);
    startActivity(intent);
}
});

```

Ejecuta el proyecto para probar guardar un nuevo contacto y presionar el Up Button en forma de descarte de cambios.

Primer nombre

Este campo no puede quedar vacío

Primer apellido

Estremera

Teléfono

Correo

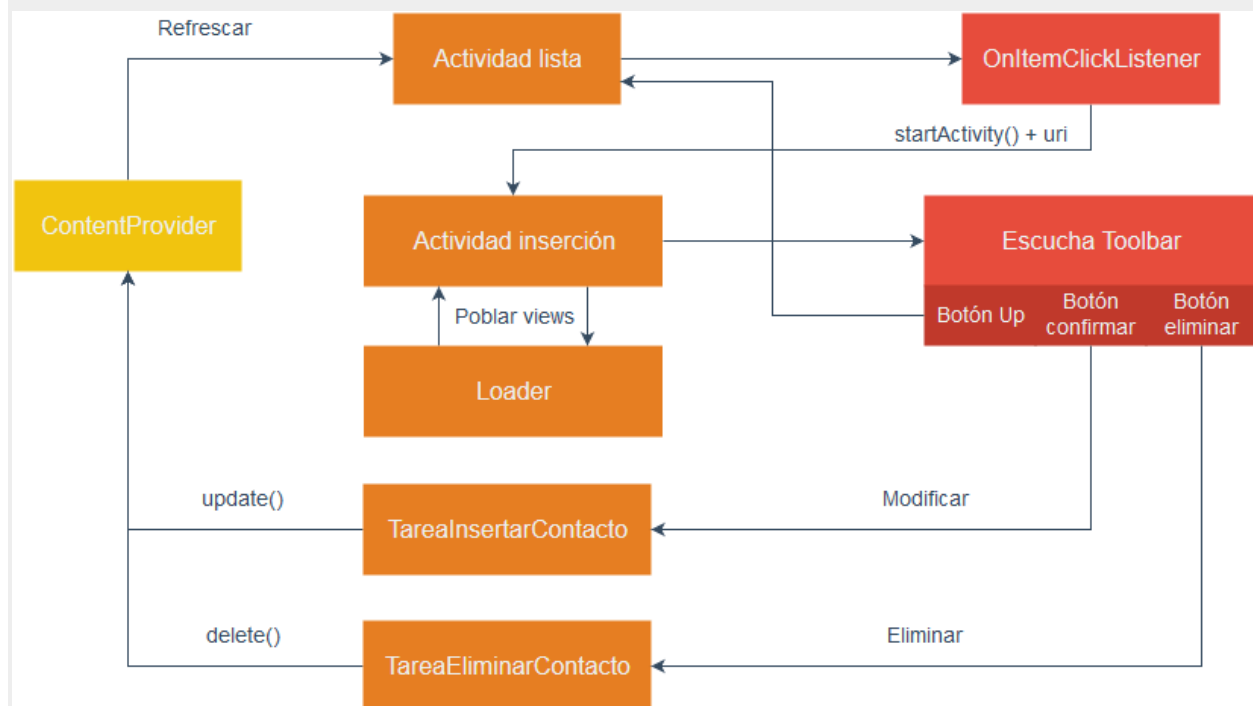
## 5.4 Modificar y Eliminar Contactos

Para modificar un contacto pensé en reciclar la actividad de inserción, ya que se requiere el mismo formulario, pero con los campos de texto previamente cargados.

De esta forma ahorraría gran cantidad de tiempo y código. Solo tendría que recibir la uri del contacto y comenzar una carga de datos para asignarlos a cada edit text. Incluso se puede usar la misma TareaInsertarContacto para la modificación.

En cuanto a la eliminación, usaré un action button. Al momento de presionarlo se iniciará una tarea asíncrona que use el método `ContentResolver.delete()` para marcar el contacto como eliminado o eliminarlo definitivamente si aún no ha sido sincronizado.

Déjame ilustrarte con un diagrama:



*¡Codifiquemos!*

**Paso #1.** Dirígete a `ActividadListaContactos.java` e inicia la

actividad `ActividadInsercionContacto` a través de `startActivity()`. El intent que lleve consigo debe tener la uri del contacto seleccionado, la cual puedes construir con el método `Contactos.construirUriContacto()`.

**Dentro de `ActividadListaContactos.java`**

```
void mostrarDetalles(Uri uri) {
    Intent intent = new Intent(this, ActividadInsercionContacto.class);
    if (null != uri) {
        intent.putExtra(ActividadInsercionContacto.URI_CONTACTO, uri.toString());
    }
}
```



```

    }
    startActivity(intent);
}

@Override
public void onClick(AdaptadorContactos.ViewHolder holder, String idContacto) {
    mostrarDetalles(Contactos.construirUriContacto(idContacto));
}

```

**Paso #2. Ahora desde ActividadInsercionContacto.java recibe el intent y extrae la uri dentro de onCreate().**

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    //Codigo preliminar...

    String uri = getIntent().getStringExtra(URI_CONTACTO);
    if (uri != null) {
        uriContacto = Uri.parse(uri);
        getSupportLoaderManager().restartLoader(1, null, this);
    }
}

```

Con este elemento iniciaremos un nuevo loader que consulte los datos del contacto a detallar. Al terminar la carga de datos (dentro de onLoadFinished()) se setean los valores del cursor en cada campo de texto.

```

private void poblarViews(Cursor data) {
    if (!data.moveToNext()) {
        return;
    }

    campoPrimerNombre.setText(UConsultas.obtenerString(data,
Contactos.PRIMER_NOMBRE));
    campoPrimerApellido.setText(UConsultas.obtenerString(data,
Contactos.PRIMER_APELLIDO));
    campoTelefono.setText(UConsultas.obtenerString(data, Contactos.TELEFONO));
    campoCorreo.setText(UConsultas.obtenerString(data, Contactos.CORREO));
}

@Override

```

```

public Loader<Cursor> onCreateLoader(int id, Bundle args) {
    return new CursorLoader(this, uriContacto, null, null, null, null);
}

@Override
public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
    poblarViews(data);
}

@Override
public void onLoaderReset(Loader<Cursor> loader) {
}

```

**Paso #3.** El siguiente movimiento es actualizar la tarea asíncrona de inserción para que reciba una uri en su método execute(). Con esto podremos identificar dos casos posibles, donde el valor de null significa que haremos una inserción (insert()) y una uri existente indica actualizar el registro (update()).

```

static class TareaAnadirContacto extends AsyncTask<Uri, Void, Void> {
    private final ContentResolver resolver;
    private final ContentValues valores;

    public TareaAnadirContacto(ContentResolver resolver, ContentValues valores) {
        this.resolver = resolver;
        this.valores = valores;
    }

    @Override
    protected Void doInBackground(Uri... args) {
        Uri uri = args[0];
        if (null != uri) {
            /*
                Verificación: Si el contacto que se va a actualizar aún no ha sido
sincronizado,
                es decir su columna 'insertado' = 1, entonces la columna 'modificado' no
debe ser
                alterada
            */
            Cursor c = resolver.query(uri, new String[]{Contactos.INSERTADO}, null,
null, null);

```

```

        if (c != null && c.moveToNext()) {

            // Verificación de sincronización
            if (UConsultas.obtenerInt(c, Contactos.INSERTADO) == 0) {
                valores.put(Contactos.MODIFICADO, 1);
            }

            valores.put(Contactos.VERSION, UTiempo.obtenerTiempo());
            resolver.update(uri, valores, null, null);
        }

    } else {
        resolver.insert(Contactos.URI_CONTENIDO, valores);
    }
    return null;
}
}

```

Obviamente esto obliga a que dentro del método insertar() se añada la uri del contacto de la siguiente forma:

```

new TareaAnadirContacto(getContentResolver(), valores).execute(uriContacto);

```

**Paso #4.** La eliminación requiere de un nuevo action button para disparar el evento. Eso significa una modificación del archivo **menu\_insercion\_contacto.xml**. Veamos:

***menu\_insercion\_contacto.xml***

```

<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <item
        android:id="@+id/accion_confirmar"
        android:icon="@drawable/icono_ok"
        android:title="@string/accion_guardar"
        app:showAsAction="ifRoom" />

    <item
        android:id="@+id/accion_eliminar"
        android:title="@string/accion_eliminar"

```

```

        android:visible="false"
        app:showAsAction="never" />
    </menu>

```

accion\_eliminar por defecto estará oculto, ya que en la inserción normal no debe aparecer. Pero en la modificación contactos es necesario. Para ello verifica dentro de onCreateOptionsMenu() si uriContacto es diferente de null y luego cambia la visibilidad del ítem.

```

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.menu_insercion_contacto, menu);

    // Verificación de visibilidad
    if (uriContacto != null) {
        menu.findItem(R.id.accion_eliminar).setVisible(true);
    }

    return true;
}

```

**Paso #5.** Crear una nueva tarea asíncrona llamada TareaEliminarContacto para eliminar el registro actual a través de la uri como parámetro. Cuando la tengas lista, simplemente registra el evento en onOptionsItemSelected() y ejecútala.

```

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    switch (id) {
        case R.id.accion_confirmar:
            insertar();
            break;
        case R.id.accion_eliminar:
            eliminar();
            break;
    }
    return super.onOptionsItemSelected(item);
}

private void eliminar() {
    if (uriContacto != null) {
        new TareaEliminarContacto(getContentResolver()).execute(uriContacto);
    }
}

```

```

        finish();
    }
}

static class TareaEliminarContacto extends AsyncTask<Uri, Void, Void> {
    private final ContentResolver resolver;

    public TareaEliminarContacto(ContentResolver resolver) {
        this.resolver = resolver;
    }

    @Override
    protected Void doInBackground(Uri... args) {

        /*
        Verificación: Si el registro no ha sido sincronizado aún, entonces puede
eliminararse
directamente. De lo contrario se marca como 'eliminado' = 1
        */
        Cursor c = resolver.query(args[0], new String[]{Contactos.INSERTADO}
            , null, null, null);

        int insertado;

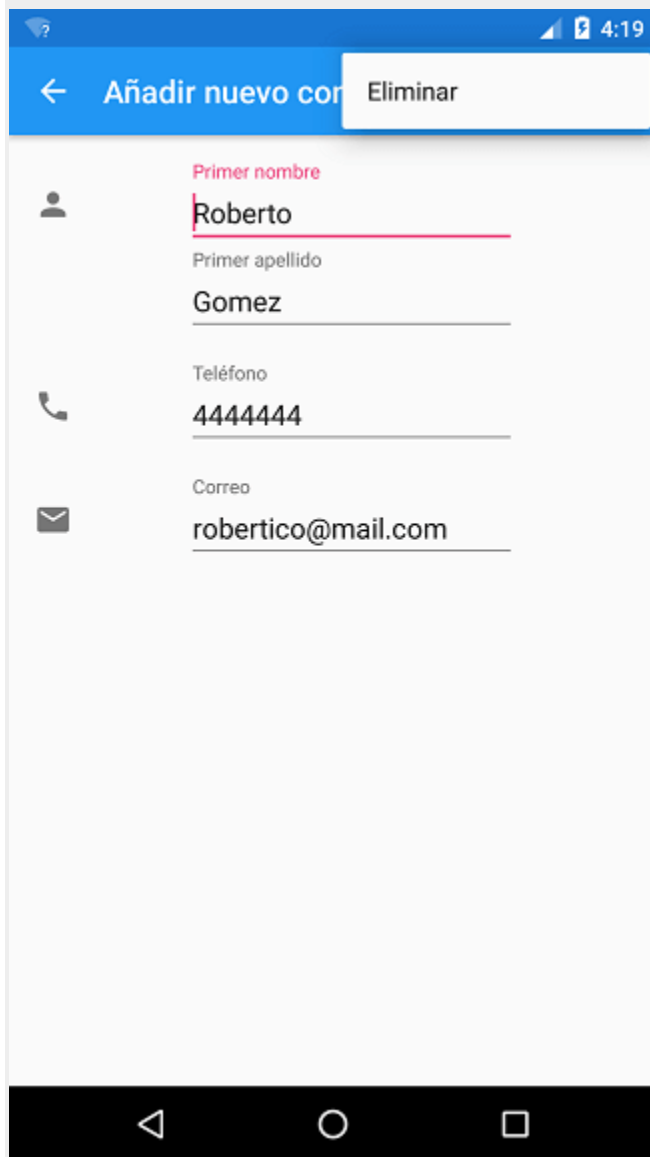
        if (c != null && c.moveToNext()) {
            insertado = UConsultas.obtenerInt(c, Contactos.INSERTADO);
        } else {
            return null;
        }

        if (insertado == 1) {
            resolver.delete(args[0], null, null);
        } else if (insertado == 0) {
            ContentValues valores = new ContentValues();
            valores.put(Contactos.ELIMINADO, 1);
            resolver.update(args[0], valores, null, null);
        }

        return null;
    }
}

```

Con estas acciones creadas, ya es posible clicar cada item de la lista y ver su contenido guardado.



The screenshot shows an Android application interface for adding or editing a contact. At the top, there is a blue header bar with a back arrow, the text "Añadir nuevo con", and a white button labeled "Eliminar". Below the header, the contact information is displayed in a form-like structure. On the left, there are icons for a person, a phone, and an email. To the right of these icons are text input fields. The first field is labeled "Primer nombre" in red and contains the text "Roberto". The second field is labeled "Primer apellido" and contains "Gomez". The third field is labeled "Teléfono" and contains "4444444". The fourth field is labeled "Correo" and contains "robertico@mail.com". The bottom of the screen shows the standard Android navigation bar with back, home, and recent apps buttons.

## 5.5 Sincronizar el cliente Android

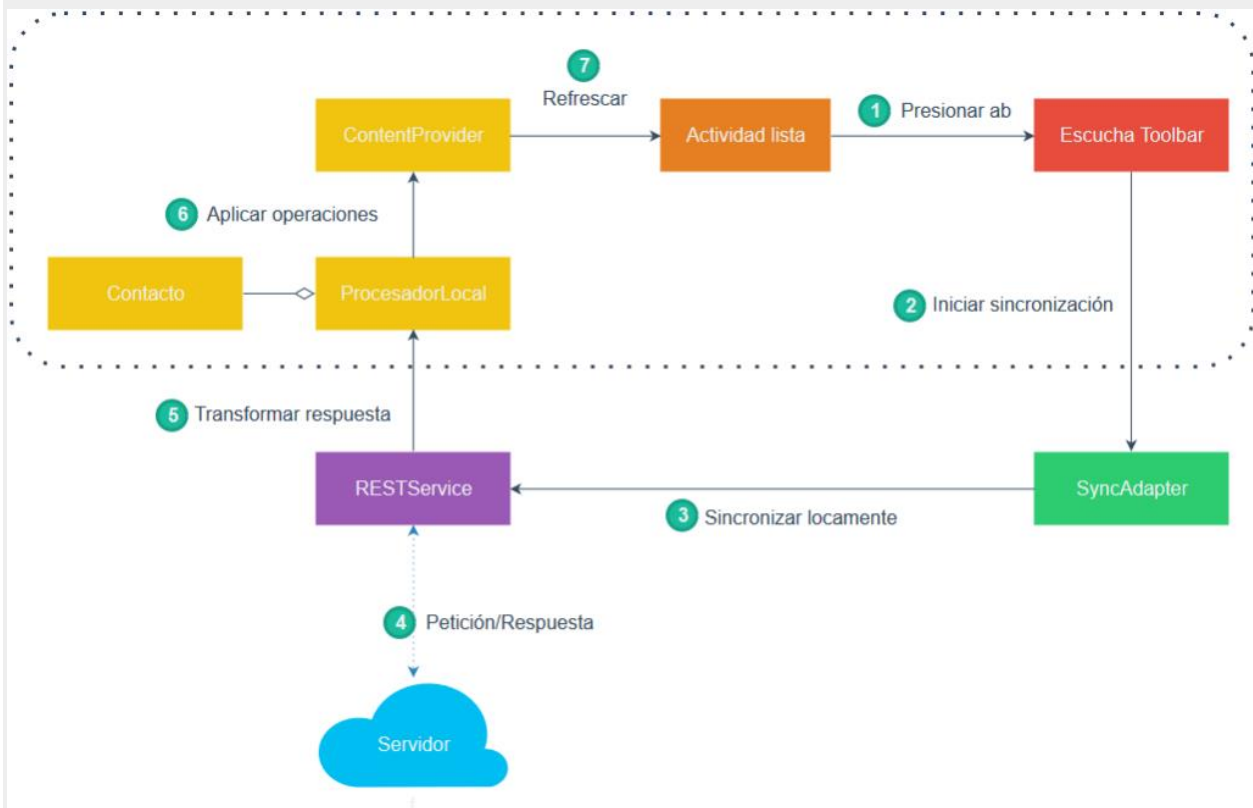
La sincronización local o del cliente Android se basa en el resultado que arroje la petición GET hacia el recurso `sync` que vimos en el apartado número 4.

La idea es comparar la carga de contactos que envía el servidor con los contactos locales en SQLite. De este resultado quedarán aquellos elementos que deben ser agregados, modificados y eliminados de la base de datos local.

La sincronización la implementaremos con el [framework de SyncAdapters de Android](#) para reducir la dificultad de uso de servicios y momentos de sincronización.

Debido a que la sincronización es por *demanda* (manual), necesitamos un action button para disparar la acción. Además me parece ideal avisar a la interfaz que la sincronización terminó exitosamente o se detuvo por errores.

Observa cómo quedaría la arquitectura:



El procesador local es una entidad que se encarga de transformar el formato JSON a elementos Contacto. Luego hace un proceso de comparación para determinar qué cambios son necesarios en el cliente. De esta manera, envía operaciones al content provider para que se actualice SQLite y por ende la actividad de contactos. Comencemos en orden.

**Paso #1.** El usuario necesita un action button que le permite iniciar la sincronización, así que solo abre el archivo `menu_lista_contactos.xml` y agrégalo para que se mantenga visible.

**`menu_lista_contactos.xml`**

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
```

```

xmlns:app="http://schemas.android.com/apk/res-auto">
<item
    android:id="@+id/accion_sync"
    android:icon="@drawable/icono_sync"
    android:title="@string/accion_sync"
    app:showAsAction="ifRoom" />
</menu>

```

**Paso #2.** Lo siguiente es disparar el evento de sincronización al presionar `accion_sync`. Para ello construye un método llamado `sincronizar()` y convoca al método `ContentResolver.requestSync()` para iniciar una sincronización manual.

```

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    if (id == R.id.accion_sync) {
        sincronizar();
        return true;
    }

    return super.onOptionsItemSelected(item);
}

private void sincronizar() {

    // Verificación para evitar iniciar más de una sync a la vez
    Account cuentaActiva = UCuentas.obtenerCuentaActiva(this);
    if (ContentResolver.isSyncActive(cuentaActiva, Contrato.AUTORIDAD)) {
        Log.d(TAG, "Ignorando sincronización ya que existe una en proceso.");
        return;
    }

    Log.d(TAG, "Solicitando sincronización manual");
    Bundle bundle = new Bundle();
    bundle.putBoolean(ContentResolver.SYNC_EXTRAS_EXPEDITED, true);
    bundle.putBoolean(ContentResolver.SYNC_EXTRAS_MANUAL, true);
    ContentResolver.requestSync(cuentaActiva, Contrato.AUTORIDAD, bundle);
}

```



El método `UCuentas.obtenerCuentaActiva()` extrae la cuenta activa guardada en las preferencias. Puedes ver su contenido si descargas el código.

Recuerda que en mi artículo [¿Cómo Sincronizar SQLite con MySQL En Android?](#) te expliqué que la sincronización por demanda se da si pasamos como argumentos `SYNC_EXTRAS_EXPEDITED` y `SYNC_EXTRAS_MANUAL` con el valor de `true`.

Por el momento no sucederá nada porque no tenemos un **SyncAdapter** creado todavía, así que procederé a crearlo.

**Paso #2.** Crea un **SyncAdapter** para realizar las sincronizaciones. Recuerda que también es necesario un autenticador para que el framework funcione correctamente.

El autenticador requiere de una definición XML para la información general de las cuentas. Para construirlo, crea un nuevo archivo dentro de la carpeta `res/xml` llamado **autenticador.xml** y añade el siguiente código.

***autenticador.xml***

```
<?xml version="1.0" encoding="utf-8"?>
<account-authenticator xmlns:android="http://schemas.android.com/apk/res/android"
    android:accountType="@string/tipo_cuenta"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:smallIcon="@mipmap/ic_launcher" />
```

Ahora añade al **AndroidManifest.xml** el permiso para autenticaciones:

```
<!-- Autenticación -->

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

Ahora crea la clase autenticadora dentro del paquete **sync**, que extienda de **AbstractAccountAuthenticator**. Como no usaremos autenticación de cuentas, solo sobrescribe en blanco los métodos.

***Autenticador.java***

```
/*
 * Autenticador auxiliar para la aplicación
 */
```

```
public class Autenticador extends AbstractAccountAuthenticator {

    public Autenticador(Context context) {
        super(context);
    }

    @Override
    public Bundle editProperties(
        AccountAuthenticatorResponse r, String s) {
        throw new UnsupportedOperationException();
    }

    @Override
    public Bundle addAccount(
        AccountAuthenticatorResponse r,
        String s,
        String s2,
        String[] strings,
        Bundle bundle) throws NetworkErrorException {
        return null;
    }

    @Override
    public Bundle confirmCredentials(
        AccountAuthenticatorResponse r,
        Account account,
        Bundle bundle) throws NetworkErrorException {
        return null;
    }

    @Override
    public Bundle getAuthToken(
        AccountAuthenticatorResponse r,
        Account account,
        String s,
        Bundle bundle) throws NetworkErrorException {
        throw new UnsupportedOperationException();
    }

    @Override
    public String getAuthTokenLabel(String s) {
```

```

        throw new UnsupportedOperationException();
    }

    @Override
    public Bundle updateCredentials(
        AccountAuthenticatorResponse r,
        Account account,
        String s, Bundle bundle) throws NetworkErrorException {
        throw new UnsupportedOperationException();
    }

    @Override
    public Bundle hasFeatures(
        AccountAuthenticatorResponse r,
        Account account, String[] strings) throws NetworkErrorException {
        throw new UnsupportedOperationException();
    }
}

```

Para finalizar el segmento de la autenticación ve a **File > New > Service > Service** y nombra al nuevo servicio **ServicioAutenticacion**. Este componente también será auxiliar, por lo que sus métodos van sobrescritos por defecto.

### ***ServicioAutenticacion.java***

```

public class ServicioAutenticacion extends Service {
    // Instancia del autenticador
    private Autenticador autenticador;

    @Override
    public void onCreate() {
        // Nueva instancia del autenticador
        autenticador = new Autenticador(this);
    }

    /*
     * Ligando el servicio al framework de Android
     */
    @Override
    public IBinder onBind(Intent intent) {
        return autenticador.getIBinder();
    }
}

```

```
}
```

Lo siguiente es ligar la información de autenticación con el servicio dentro del manifiesto.

```
<service
    android:name=".sync.ServicioAutenticacion"
    android:enabled="true"
    android:exported="true" >
    <intent-filter>
        <action android:name="android.accounts.AccountAuthenticator" />
    </intent-filter>

    <meta-data
        android:name="android.accounts.AccountAuthenticator"
        android:resource="@xml/autenticador" />
</service>
```

Con eso la ya hemos satisfecho la sección de autenticación. Ahora comencemos con la sincronización.

Lo primero es añadir los permisos para que el framework sincronice:

```
<!-- Sincronización -->

<uses-permission android:name="android.permission.WRITE_SYNC_SETTINGS"/>

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

Ahora crea una nueva clase Java llamada SyncAdapter que herede de AbstractThreadedSyncAdapter. Crea dos constructores, uno con la firma para versiones inferiores a la 3.0 y otra para las recientes. En seguida sobrescribe el método onPerformSync() en blanco.

***SyncAdapter.java***

```

public class SyncAdapter extends AbstractThreadedSyncAdapter {

    private static final String TAG = SyncAdapter.class.getSimpleName();

    private ContentResolver resolver;

    public SyncAdapter(Context context, boolean autoInitialize) {
        super(context, autoInitialize);
        resolver = context.getContentResolver();
    }

    /**
     * Constructor para mantener compatibilidad en versiones inferiores a 3.0
     */
    public SyncAdapter(
        Context context,
        boolean autoInitialize,
        boolean allowParallelSyncs) {
        super(context, autoInitialize, allowParallelSyncs);
        resolver = context.getContentResolver();
    }

    @Override
    public void onPerformSync(Account account,
        Bundle extras,
        String authority,
        ContentProviderClient provider,
        final SyncResult syncResult) {

        Log.i(TAG, "Comenzando a sincronizar:" + account);

    }
}

```

Agrega un nuevo [bound service](#) al paquete `sync`, desde el menú File y llámalo `ServicioSincronizacion`. Dentro de este `onCreate()` crea la instancia del sync adapter y luego pasa su interfaz de comunicación en `onBind()`.

***ServicioSincronizacion.java***

```

public class ServicioSincronizacion extends Service {
    // Instancia del sincronizar adapter

```

```

private static SyncAdapter syncAdapter = null;
// Objeto para prevenir errores entre hilos
private static final Object lock = new Object();

@Override
public void onCreate() {
    synchronized (lock) {
        if (syncAdapter == null) {
            syncAdapter = new SyncAdapter(getApplicationContext(), true);
        }
    }
}

/**
 * Retorna interfaz de comunicación para uso interno del framework
 */
@Override
public IBinder onBind(Intent intent) {
    return syncAdapter.getSyncAdapterBinder();
}
}

```

No olvides que este servicio requiere una configuración previa que se describe en un archivo xml con la etiqueta <sync-adapter>. Sabiendo esto, crea un nuevo archivo dentro de res/xml que se llame **sync\_adapter.xml** y agrega las siguientes características:

**sync\_adapter.xml**

```

<?xml version="1.0" encoding="utf-8"?>
<sync-adapter xmlns:android="http://schemas.android.com/apk/res/android"
    android:accountType="@string/tipo_cuenta"
    android:allowParallelSyncs="false"
    android:contentAuthority="@string/autoridad_provider"
    android:supportsUploading="true"
    android:userVisible="false" />

```

A su vez necesitamos ligar esta configuración al servicio en el **AndroidManifest.xml** en sus metadatos.

```

<service
    android:name=".sync.ServicioSincronizacion"
    android:enabled="true"

```

```

        android:exported="true" >
        <intent-filter>
            <action android:name="android.content.SyncAdapter" />
        </intent-filter>

        <meta-data
            android:name="android.content.SyncAdapter"
            android:resource="@xml/sync_adapter" />
    </service>

```

Hasta aquí el sync adapter se ejecuta cuando el usuario presiona el botón de sincronización, pero no sucede absolutamente nada, ya que `onPerformSync()` no tiene acciones por ejecutar. Para ello tenemos que crear un componente que acceda al servidor primero.

**Paso #3.** Crea una nueva clase llamada `RESTService` dentro del paquete `web`. El propósito de esta es usar a Volley para crear dos métodos representativos del servicio web: `get()` y `post()`.

***RESTService.java***

```

public class RESTService {

    private static final String TAG = RESTService.class.getSimpleName();

    private final Context contexto;

    public RESTService(Context contexto) {
        this.contexto = contexto;
    }

    public void get(String uri, Response.Listener<JSONObject> jsonListener,
                    Response.ErrorListener errorListener,
                    final HashMap<String, String> cabeceras) {

        // Crear petición GET
        JSONObjectRequest petition = new JSONObjectRequest(
            uri,
            null,
            jsonListener,
            errorListener
        ) {

```

```

        @Override
        public Map<String, String> getHeaders() throws AuthFailureError {
            return cabeceras;
        }
    };

    // Añadir petición a la pila
    VolleySingleton.getInstance(contexto).addToRequestQueue(peticion);
}

public void post(String uri, String datos, Response.Listener<JSONObject>
jsonListener,
                Response.ErrorListener errorListener, final HashMap<String,
String> cabeceras) {

    // Crear petición POST
    JsonObjectRequest peticion = new JsonObjectRequest(
        Request.Method.POST,
        uri,
        datos,
        jsonListener,
        errorListener
    ) {
        @Override
        public Map<String, String> getHeaders() throws AuthFailureError {
            return cabeceras;
        }
    };

    // Añadir petición a la pila
    VolleySingleton.getInstance(contexto).addToRequestQueue(peticion);
}
}

```

Básicamente lo que hago es encapsular peticiones `JsonObjectRequest` de Volley en métodos que permitan tener control desde afuera. Para Volley uso el siguiente singleton.

***VolleySingleton.java***

```

public final class VolleySingleton {

```



```

// Atributos
private static VolleySingleton singleton;
private RequestQueue requestQueue;
private static Context context;

private VolleySingleton(Context context) {
    VolleySingleton.context = context;
    requestQueue = getRequestQueue();
}

/**
 * Retorna la instancia unica del singleton
 *
 * @param context contexto donde se ejecutarán las peticiones
 * @return Instancia
 */
public static synchronized VolleySingleton getInstance(Context context) {
    if (singleton == null) {
        singleton = new VolleySingleton(context.getApplicationContext());
    }
    return singleton;
}

/**
 * Obtiene la instancia de la cola de peticiones
 *
 * @return cola de peticiones
 */
private RequestQueue getRequestQueue() {
    if (requestQueue == null) {
        requestQueue = Volley.newRequestQueue(context.getApplicationContext());
    }
    return requestQueue;
}

/**
 * Añade la petición a la cola
 *
 * @param req petición

```

```

    * @param <T> Resultado final de tipo T
    */
    public <T> void addToRequestQueue(Request<T> req) {
        getRequestQueue().add(req);
    }
}

```

Ambos tienen la sobrescritura del método `getHeaders()`, donde se asignan las cabeceras que se indiquen en el último parámetro de cada método.

Si aún no sabes cómo funciona volley, entonces lee mi artículo sobre [creación de peticiones HTTP con Volley](#).

Teniendo el método `get()` ya puedes enviar la petición al servidor desde `onPerformSync()` de la siguiente forma:

```

@Override
public void onPerformSync(Account account,
                          Bundle extras,
                          String authority,
                          ContentProviderClient provider,
                          final SyncResult syncResult) {

    Log.i(TAG, "Comenzando a sincronizar:" + account);

    // Sincronización local
    syncLocal();

}

private void syncLocal() {
    // Construcción de cabeceras
    HashMap<String, String> cabeceras = new HashMap<>();
    cabeceras.put("Authorization", UPreferencias.obtenerClaveApi(getContext()));

    // Petición GET
    new RESTService(getContext()).get(URL_SYNC_BATCH,
        new Response.Listener<JSONObject>() {
            @Override
            public void onResponse(JSONObject response) {
                // Procesar GET
            }
        }
    ),
}

```

```

        new Response.ErrorListener() {
            @Override
            public void onErrorResponse(VolleyError error) {
                // Procesar Error
            }
        }, cabeceras);
    }
}

```

Donde URL\_SYNC\_BATCH es la ubicación del recurso sync en el servidor local.

```

// Recurso sync (10.0.3.2 -> Genymotion; 10.0.2.2 -> AVD)
public static final String URL_SYNC_BATCH =
"http://10.0.3.2/api.peopleapp.com/v1/sync";

```

**Paso #4.** La petición que se lanzó desde syncLocal() no procesa la respuesta o error que esta produzca. La arquitectura muestra que usaremos un componente llamado ProcesadorLocal, el cual se encarga de transformar el contenido JSON en POJOs Java para realizar una comparación.

No obstante, primero es necesario crear la clase que represente lógicamente a cada contacto para que el procesador la implemente.

El objetivo de la clase Contacto es mostrar los atributos de cada registro para realizar el [parsing JSON](#) respectivo. Su contenido sería este:

**Contacto.java**

```

public class Contacto {

    public String idContacto;
    public String primerNombre;
    public String primerApellido;
    public String telefono;
    public String correo;
    public String version;
    public int modificado;

    public Contacto(String idContacto, String primerNombre,
                    String primerApellido, String telefono, String correo, String
version, int modificado) {
        this.idContacto = idContacto;
        this.primerNombre = primerNombre;
        this.primerApellido = primerApellido;
        this.telefono = telefono;
    }
}

```

```

        this.correo = correo;
        this.version = version;
        this.modificado = modificado;
    }

    public void aplicarSanidad() {
        idContacto = idContacto == null ? "" : idContacto;
        primerNombre = primerNombre == null ? "" : primerNombre;
        primerApellido = primerApellido == null ? "" : primerApellido;
        telefono = telefono == null ? "" : telefono;
        correo = correo == null ? "" : correo;
        version = version == null ? UTiempo.obtenerTiempo() : version;
        modificado = 0;
    }

    public int esMasReciente(Contacto match) {
        SimpleDateFormat formato = new SimpleDateFormat("yyyy-MM-dd hh:mm:ss");
        try {
            Date fechaA = formato.parse(version);
            Date fechaB = formato.parse(match.version);

            return fechaA.compareTo(fechaB);

        } catch (ParseException e) {
            e.printStackTrace();
        }
        return 0;
    }

    public boolean compararCon(Contacto otro) {
        return idContacto.equals(otro.idContacto) &&
            primerNombre.equals(otro.primerNombre) &&
            primerApellido.equals(otro.primerApellido) &&
            telefono.equals(otro.telefono) &&
            correo.equals(otro.correo);
    }
}

```

Esta clase tiene tres métodos indispensable para sincronizar:

- `aplicarSanidad()`: Lo uso al momento de parsear la respuesta JSON en pojos para evitar tener campos con el valor null que perjudiquen las comparaciones que se hacen en `compararCon()`.
- `esMasReciente()`: Comparar dos objetos `Contacto` para determinar cual es mas reciente basado en el atributo `version`.
- `compararCon()`: Determina si dos contactos son iguales.

**Paso #5.** Como ya lo he venido diciendo, el procesador transforma la respuesta JSON en unidades de datos entendibles en la lógica del programa (instancias de `Contacto`).

Luego compara los registros de SQLite existentes con los contactos resultantes del parsing y determina que inserciones, modificaciones y eliminaciones deben realizarse localmente.

El algoritmo general sería así:

- a) Parsear array JSON de contactos a instancias `Contacto`
- b) Consultar los contactos locales
- c) Comparar cada contacto local con la lista de remotos, donde
  - Si el contacto local existe en la lista de remotos, entonces se descarta de inserción
  - Si el contacto local existe, entonces buscar diferencias con el remoto para programar actualización
  - Si el contacto local no existen en la lista remota, entonces programar eliminación
- d) Enviar todas las operaciones programadas

Aspectos a destacar:

- Usa la clase `ContentProviderOperation` y sus métodos estáticos `newInsert()`, `newUpdate()` y `newDelete()` para representar una operación que será ejecutada como parte de un batch en el `ContentProvider`, con `applyBatch()`.
- La librería **Gson** te permite reflejar un string con formato JSON en un objeto que coincida con los atributos que poseen.
- Almacena la lista de contactos remotos en un `HashMap` que use a `idContacto` como clave, para variar el contenido fácilmente según las comparaciones.

El código producido para el procesador sería el siguiente:

***ProcesadorLocal.java***

```
public class ProcesadorLocal {
```

```

private static final String TAG = ProcesadorLocal.class.getSimpleName();

private interface ConsultaContactos {

    // Proyección para consulta de contactos
    String[] PROYECCION = {
        Contactos.ID_CONTACTO,
        Contactos.PRIMER_NOMBRE,
        Contactos.PRIMER_APELLIDO,
        Contactos.TELEFONO,
        Contactos.CORREO,
        Contactos.VERSION,
        Contactos.MODIFICADO
    };

    // Indices de columnas
    int ID_CONTACTO = 0;
    int PRIMER_NOMBRE = 1;
    int PRIMER_APELLIDO = 2;
    int TELEFONO = 3;
    int CORREO = 4;
    int VERSION = 5;
    int MODIFICADO = 6;

}

// Mapa para filtrar solo los elementos a sincronizar
private HashMap<String, Contacto> contactosRemotos = new HashMap<>();

// Conversor JSON
private Gson gson = new Gson();

public ProcesadorLocal() {

}

public void procesar(JSONArray arrayJsonContactos) {
    // Añadir elementos convertidos a los contactos remotos
    for (Contacto contactoActual : gson
        .fromJson(arrayJsonContactos.toString(), Contacto[].class)) {
        contactoActual.aplicarSanidad();
    }
}

```

```

        contactosRemotos.put(contactoActual.idContacto, contactoActual);
    }
}

public void procesarOperaciones(ArrayList<ContentProviderOperation> ops,
ContentResolver resolver) {

    // Consultar contactos locales
    Cursor c = resolver.query(Contactos.URI_CONTENIDO,
        ConsultaContactos.PROYECCION,
        Contactos.INSERTADO + "=?",
        new String[]{"0"}, null);

    if (c != null) {

        while (c.moveToNext()) {

            // Convertir fila del cursor en objeto Contacto
            Contacto filaActual = deCursorAContacto(c);

            // Buscar si el contacto actual se encuentra en el mapa de
mapacontactos
            Contacto match = contactosRemotos.get(filaActual.idContacto);

            if (match != null) {
                // Esta entrada existe, por lo que se remueve del mapeado
                contactosRemotos.remove(filaActual.idContacto);

                // Crear uri de este contacto
                Uri updateUri =
Contactos.construirUriContacto(filaActual.idContacto);

                /*
                Aquí se aplica la resolución de conflictos de modificaciones de un
mismo recurso
                tanto en el servidor como en la app. Quién tenga la versión más
actual, será tomado
                como preponderante
                */
                if (!match.compararCon(filaActual)) {
                    int flag = match.esMasReciente(filaActual);

```

```

        if (flag > 0) {
            Log.d(TAG, "Programar actualización del contacto " +
updateUri);

            // Verificación: ¿Existe conflicto de modificación?
            if (filaActual.modificado == 1) {
                match.modificado = 0;
            }
            ops.add(construirOperacionUpdate(match, updateUri));

        }

    }

} else {
    /*
        Se deduce que aquellos elementos que no coincidieron, ya no
existen en el servidor,
        por lo que se eliminarán
    */
    Uri deleteUri =
Contactos.construirUriContacto(filaActual.idContacto);
    Log.i(TAG, "Programar Eliminación del contacto " + deleteUri);
    ops.add(ContentProviderOperation.newDelete(deleteUri).build());
}

}

c.close();
}

// Insertar los items resultantes ya que se asume que no existen en el
contacto
for (Contacto contacto : contactosRemotos.values()) {
    Log.d(TAG, "Programar Inserción de un nuevo contacto con ID = " +
contacto.idContacto);
    ops.add(construirOperacionInsert(contacto));
}

}

private ContentProviderOperation construirOperacionInsert(Contacto contacto) {
    return ContentProviderOperation.newInsert(Contactos.URI_CONTENIDO)
        .withValue(Contactos.ID_CONTACTO, contacto.idContacto)

```



```

        .withValue(Contactos.PRIMER_NOMBRE, contacto.primerNombre)
        .withValue(Contactos.PRIMER_APELLIDO, contacto.primerApellido)
        .withValue(Contactos.TELEFONO, contacto.telefono)
        .withValue(Contactos.CORREO, contacto.correo)
        .withValue(Contactos.VERSION, contacto.version)
        .withValue(Contactos.INSERTADO, 0)
        .build();
    }

    private ContentProviderOperation construirOperacionUpdate(Contacto match, Uri
updateUri) {
        return ContentProviderOperation.newUpdate(updateUri)
            .withValue(Contactos.ID_CONTACTO, match.idContacto)
            .withValue(Contactos.PRIMER_NOMBRE, match.primerNombre)
            .withValue(Contactos.PRIMER_APELLIDO, match.primerApellido)
            .withValue(Contactos.TELEFONO, match.telefono)
            .withValue(Contactos.CORREO, match.correo)
            .withValue(Contactos.VERSION, match.version)
            .withValue(Contactos.MODIFICADO, match.modificado)
            .build();
    }

    /**
     * Convierte una fila de un Cursor en un nuevo Contacto
     *
     * @param c cursor
     * @return objeto contacto
     */
    private Contacto deCursorAContacto(Cursor c) {
        return new Contacto(
            c.getString(ConsultaContactos.ID_CONTACTO),
            c.getString(ConsultaContactos.PRIMER_NOMBRE),
            c.getString(ConsultaContactos.PRIMER_APELLIDO),
            c.getString(ConsultaContactos.TELEFONO),
            c.getString(ConsultaContactos.CORREO),
            c.getString(ConsultaContactos.VERSION),
            c.getInt(ConsultaContactos.MODIFICADO)
        );
    }
}

```

La respuesta de la petición GET es procesada con el método `tratarGet()`, donde usamos el `ProcesadorLocal` y luego aplicar los cambios a través de `ContentResolver.applyBatch()`.

Por otro lado, los errores puedes tratarlos en otro método diferente como el que yo nombré en el código anterior (`tratarErrores()`). Recuerda que la respuesta de un error trae consigo un objeto `Json` que contiene un atributo `'estado'` y un `'mensaje'`. Basándome en eso, creé una clase pequeña que represente esa respuesta para tener un parsing con `Gson`, llamada `RespuestaApi`.

```
public class RespuestaApi {
    private int estado;
    private String mensaje;

    public RespuestaApi(int code, String body) {
        this.estado = code;
        this.mensaje = body;
    }

    public RespuestaApi() {

    }

    public int getEstado() {
        return estado;
    }

    public String getMensaje() {
        return mensaje;
    }

    @Override
    public String toString() {
        return "(" + getEstado() + "): " + getMensaje();
    }
}
```

Con esta puedes leer el error de la siguiente forma:

```
private void tratarErrores(VolleyError error) {
    // Crear respuesta de error por defecto
    RespuestaApi respuesta = new RespuestaApi(ESTADO_PETICION_FALLIDA,
```

```

        "Petición incorrecta");

// Verificación: ¿La respuesta tiene contenido interpretable?
if (error.networkResponse != null) {

    String s = new String(error.networkResponse.data);
    try {
        respuesta = gson.fromJson(s, RespuestaApi.class);
    } catch (JsonSyntaxException e) {
        Log.d(TAG, "Error de parsing: " + s);
    }

}

if (error instanceof NetworkError) {
    respuesta = new RespuestaApi(ESTADO_TIEMPO_ESPERA
        , "Error en la conexión. Intentalo de nuevo");
}

// Error de espera al servidor
if (error instanceof TimeoutError) {
    respuesta = new RespuestaApi(ESTADO_TIEMPO_ESPERA, "Error de espera del
servidor");
}

// Error de parsing
if (error instanceof ParseError) {
    respuesta = new RespuestaApi(ESTADO_ERROR_PARSING, "La respuesta no es formato
JSON");
}

// Error conexión servidor
if (error instanceof ServerError) {
    respuesta = new RespuestaApi(ESTADO_ERROR_SERVIDOR, "Error en el servidor");
}

if (error instanceof NoConnectionError) {
    respuesta = new RespuestaApi(ESTADO_ERROR_SERVIDOR
        , "Servidor no disponible, prueba mas tarde");
}

```

```

        Log.d(TAG, "Error Respuesta:" + (respuesta != null ? respuesta.toString() : "()")
            + "\nDetalles:" + error.getMessage());

        enviarBroadcast(false, respuesta.getMensaje());

    }

```

En este momento ya es posible sincronizar localmente nuestra app. Para probarlo, agrega dos nuevos registros en la base de datos Mysql y presiona el botón de sincronizar. Deberían de copiarse ambos registros en la aplicación Android.

Luego prueba editar los registros del servidor y vuelve a sincronizar. También prueba la eliminación para ver si se replica.

## 5.6 Sincronizar base de datos Mysql del servidor

La sincronización del servidor la iniciaré al final de la sincronización local. Cuando esta termine, entonces notificaré al usuario de que este proceso culminó con satisfacción.

Ahora el problema es encontrar la forma de recolectar todos los registros de la base de datos que estén marcados como insertados, modificados y eliminados. Luego parsearlos a JSON y enviarlos en la petición con la forma que requiere el método `post()` en el archivo **sync.php**.

La solución la encontrarás en la creación de la clase `ProcesadorRemoto`, la cual es tiene naturaleza similar a `ProcesadorLocal`. Esta vez haremos el proceso inverso, obteniendo cada una de las operaciones remotas para agruparlas en un objeto JSON.

Básate en esta serie de pasos para conseguir las operaciones:

1. Usa el método `ContentResolver.query()` para extraer registros marcados. Por ejemplo, los contactos insertados se consiguen con la condición `'WHERE insertado = 1'`.
2. Itera sobre el cursor con un `while` y mapea cada fila como un `Map`. Esto permitirá realizar fácilmente el parsing con `Gson`.
3. Agrega el mapeado a una lista.
4. Retorna en la lista si esta tiene uno o más elementos.

**Paso #1.** Así que crea una nueva clase dentro del paquete **sync** llamada `ProcesadorRemoto` y agrega el siguiente código.

***ProcesadorRemoto.java***

```

public class ProcesadorRemoto {
    private static final String TAG = ProcesadorRemoto.class.getSimpleName();

    // Campos JSON
    private static final String INSERCIONES = "inserciones";
    private static final String MODIFICACIONES = "modificaciones";
    private static final String ELIMINACIONES = "eliminaciones";

    private Gson gson = new Gson();

    private interface ConsultaContactos {

        // Proyección para consulta de contactos
        String[] PROYECCION = {
            Contactos.ID_CONTACTO,
            Contactos.PRIMER_NOMBRE,
            Contactos.PRIMER_APELLIDO,
            Contactos.TELEFONO,
            Contactos.CORREO,
            Contactos.VERSION
        };
    }

    public String crearPayload(ContentResolver cr) {
        HashMap<String, Object> payload = new HashMap<>();

        List<Map<String, Object>> inserciones = obtenerInserciones(cr);
        List<Map<String, Object>> modificaciones = obtenerModificaciones(cr);
        List<String> eliminaciones = obtenerEliminaciones(cr);

        // Verificación: ¿Hay cambios locales?
        if (inserciones == null && modificaciones == null && eliminaciones == null) {
            return null;
        }

        payload.put(INSERCIONES, inserciones);
        payload.put(MODIFICACIONES, modificaciones);
        payload.put(ELIMINACIONES, eliminaciones);
    }
}

```

```

        return gson.toJson(payload);
    }

    public List<Map<String, Object>> obtenerInserciones(ContentResolver cr) {
        List<Map<String, Object>> ops = new ArrayList<>();

        // Obtener contactos donde 'insertado' = 1
        Cursor c = cr.query(Contactos.URI_CONTENIDO,
            ConsultaContactos.PROYECCION,
            Contactos.INSERTADO + "=?",
            new String[]{"1"}, null);

        // Comprobar si hay trabajo que realizar
        if (c != null && c.getCount() > 0) {

            Log.d(TAG, "Inserciones remotas: " + c.getCount());

            // Procesar inserciones
            while (c.moveToNext()) {
                ops.add(mapearInsercion(c));
            }

            return ops;

        } else {
            return null;
        }
    }

    public List<Map<String, Object>> obtenerModificaciones(ContentResolver cr) {

        List<Map<String, Object>> ops = new ArrayList<>();

        // Obtener contactos donde 'modificado' = 1
        Cursor c = cr.query(Contactos.URI_CONTENIDO,
            ConsultaContactos.PROYECCION,
            Contactos.MODIFICADO + "=?",
            new String[]{"1"}, null);

        // Comprobar si hay trabajo que realizar

```

```

        if (c != null && c.getCount() > 0) {

            Log.d(TAG, "Existen " + c.getCount() + " modificaciones de contactos");

            // Procesar operaciones
            while (c.moveToNext()) {
                ops.add(mapearActualizacion(c));
            }

            return ops;

        } else {
            return null;
        }
    }

    public List<String> obtenerEliminaciones(ContentResolver cr) {

        List<String> ops = new ArrayList<>();

        // Obtener contactos donde 'eliminado' = 1
        Cursor c = cr.query(Contactos.URI_CONTENIDO,
            ConsultaContactos.PROYECCION,
            Contactos.ELIMINADO + "=?",
            new String[]{"1"}, null);

        // Comprobar si hay trabajo que realizar
        if (c != null && c.getCount() > 0) {

            Log.d(TAG, "Existen " + c.getCount() + " eliminaciones de contactos");

            // Procesar operaciones
            while (c.moveToNext()) {
                ops.add(UConsultas.obtenerString(c, Contactos.ID_CONTACTO));
            }

            return ops;

        } else {
            return null;
        }
    }

```

```

    }

}

/**
 * Desmarca los contactos locales que ya han sido sincronizados
 *
 * @param cr content resolver
 */
public void desmarcarContactos(ContentResolver cr) {
    // Establecer valores de la actualización
    ContentValues valores = new ContentValues();
    valores.put(Contactos.INSERTADO, 0);
    valores.put(Contactos.MODIFICADO, 0);

    String seleccion = Contactos.INSERTADO + " = ? OR " +
        Contactos.MODIFICADO + "= ?";
    String[] argumentos = {"1", "1"};

    // Modificar banderas de insertados y modificados
    cr.update(Contactos.URI_CONTENIDO, valores, seleccion, argumentos);

    seleccion = Contactos.ELIMINADO + "=?";
    // Eliminar definitivamente
    cr.delete(Contactos.URI_CONTENIDO, seleccion, new String[]{"1"});
}

private Map<String, Object> mapearInsercion(Cursor c) {
    // Nuevo mapa para reflejarlo en JSON
    Map<String, Object> mapaContacto = new HashMap<String, Object>();

    // Añadir valores de columnas como atributos
    UDatos.agregarStringAMapa(mapaContacto, Contactos.ID_CONTACTO, c);
    UDatos.agregarStringAMapa(mapaContacto, Contactos.PRIMER_NOMBRE, c);
    UDatos.agregarStringAMapa(mapaContacto, Contactos.PRIMER_APELLIDO, c);
    UDatos.agregarStringAMapa(mapaContacto, Contactos.TELEFONO, c);
    UDatos.agregarStringAMapa(mapaContacto, Contactos.CORREO, c);
    UDatos.agregarStringAMapa(mapaContacto, Contactos.VERSION, c);
}

```



```

        return mapaContacto;
    }

    private Map<String, Object> mapearActualizacion(Cursor c) {
        // Nuevo mapa para reflejarlo en JSON
        Map<String, Object> mapaContacto = new HashMap<String, Object>();

        // Añadir valores de columnas como atributos
        UDatos.agregarStringAMapa(mapaContacto, Contactos.ID_CONTACTO, c);
        UDatos.agregarStringAMapa(mapaContacto, Contactos.PRIMER_NOMBRE, c);
        UDatos.agregarStringAMapa(mapaContacto, Contactos.PRIMER_APELLIDO, c);
        UDatos.agregarStringAMapa(mapaContacto, Contactos.TELEFONO, c);
        UDatos.agregarStringAMapa(mapaContacto, Contactos.CORREO, c);
        UDatos.agregarStringAMapa(mapaContacto, Contactos.VERSION, c);

        return mapaContacto;
    }
}

```

Nota que el método `crearPayload()` acumula los resultados de las tres operaciones en un solo mapa que será enviado en con el método `post()` de la clase `RETSservice`. Adicionalmente tenemos un método llamado `desmarcarContactos()` cuya función es poner en 0 las banderas de todos los registros sincronizados. Con ello evito que se envíen a otra operación sin tener nada que ver.

**Paso #2.** Envía la petición `POST` justo al final de las acciones del método `syncLocal()`. Pon el resultado del método `crearPayload()` de una instancia del procesador remoto, como segundo parámetro del método `post()`.

```

private void syncRemota() {
    procRemoto = new ProcesadorRemoto();

    // Construir payload con todas las operaciones remotas pendientes
    String datos = procRemoto.crearPayload(cr);

    if (datos != null) {
        Log.d(TAG, "Payload de contactos:" + datos);

        HashMap<String, String> cabeceras = new HashMap<>();
        cabeceras.put("Authorization", UPreferencias.obtenerClaveApi(getContext()));

        new RETSservice(getContext()).post(URL_SYNC_BATCH, datos,

```

```

        new Response.Listener<JSONObject>() {
            @Override
            public void onResponse(JSONObject response) {
                tratarPost();
            }
        },
        new Response.ErrorListener() {
            @Override
            public void onErrorResponse(VolleyError error) {
                tratarErrores(error);
            }
        }
        , cabeceras);
    } else {
        Log.d(TAG, "Sin cambios locales");
        enviarBroadcast(true, "Sincronización completa");
    }
}

```

Al recibir una respuesta exitosa del servidor desmarca los pedidos, de lo contrario procesa los errores con `tratarErrores()`.

```

private void tratarPost() {
    // Desmarcar inserciones locales
    procRemoto.desmarcarContactos(cr);
}

```

## 5.7 Notificar estados de sincronización con SnackBars

Si ejecutas la app hasta el momento, disfrutarás de la sincronización local y remota. El **Logcat** te mostrará los sucesos que vayan ocurriendo en la petición HTTP, el parsing JSON y las operaciones en el `ContentProvider`.

— *¿Pero qué hay del usuario?*

— *¿Cómo sabrá el que la sincronización terminó o falló?*

Existen varios elementos visuales que podrían ayudarte, tales como los Toasts, SnackBars, notificaciones, barras de progreso o indicadores personalizados en los views. Hay gran variedad para escoger.

En mi caso haré que las snack bars soporten mis avisos.

El problema de notificar estados es que no sabemos en qué momento irán a terminar las acciones en el background que ejecuta el SyncAdapter. Lo que no permite ejecutar acciones de forma secuencial.

Pero no te preocupes. La solución la hallarás en un componente llamado [LocalBroadcastManager](#). Este se encarga de enviar [intents entre los componentes](#) locales de tu app, como lo es la comunicación entre el sync adapter y la actividad contactos.

Para comenzar a enviar notificaciones solo sigue estos pasos:

1. Crea una nueva clase que extienda de `BroadcastReceiver` dentro de la actividad o componente que recibirá mensajes. Dictamina las acciones que se ejecutarán al recibir el mensaje dentro de `onReceive()`.
2. Registra el receiver con el método `registerReceiver()` del `LocalBroadcastManager`. Este registro requiere un filtro `IntentFilter` que especifique todas las acciones de las que estará pendiente el receptor.
3. Desregístralo en un lugar donde no sea necesario recibir las notificaciones con `LocalBroadcastManager.unregisterReceiver()`.
4. Finalmente envía mensajes desde otro componente con el método `LocalBroadcastManager.sendBroadcast()`, instanciando intents con la acción objetivo y añadiendo argumentos que requieran procesarse en el receptor.

*Veamos.*

**Paso #1.** Abre `ActividadListaContactos` y agrega una nueva clase anónima del tipo `BroadcastReceiver`, dentro de `onCreate()`. Luego sobrescribe `onReceive()` para que produzca una `Snackbar` con el mensaje de texto que vendrá desde otros componentes.

```
private BroadcastReceiver receptorSync;

@Override
protected void onCreate(Bundle savedInstanceState) {
    // Más código...

    receptorSync = new BroadcastReceiver() {

        @Override
        public void onReceive(Context context, Intent intent) {
            String mensaje = intent.getStringExtra("extra.mensaje");
            Snackbar.make(findViewById(R.id.coordinador),
                mensaje, Snackbar.LENGTH_LONG).show();
        }
    };
};
```

```
}
```

**Paso #2.** Registra el receiver en el método `onResume()` de la actividad con un filtro del tipo `Intent.ACTION_SYNC` de la siguiente forma:

```
@Override
protected void onResume() {
    super.onResume();
    // Registrar receptor
    IntentFilter filtroSync = new IntentFilter(Intent.ACTION_SYNC);
    LocalBroadcastManager.getInstance(this).registerReceiver(receptorSync,
    filtroSync);
}
```

Cabe aclarar que `ACTION_SYNC` es solo una constante de la clase `Intent` para especificar acciones de sincronización, sin embargo puedes crear tu propia constante para determinar la acción.

**Paso #3.** El desregistro puedes hacerlo en el método `onPause()`, ya que no es de utilidad recibir mensajes cuando la actividad está siendo superpuesta por otro componente y mucho menos cuando está en segundo plano.

```
@Override
protected void onPause() {
    super.onPause();
    // Desregistrar receptor
    LocalBroadcastManager.getInstance(this).unregisterReceiver(receptorSync);
}
```

**Paso #4.** Abre tu sync adapter y envía intents desde el método `tratarErrores()`. También puedes hacerlo al terminar la sincronización completa en `tratarPost()`. Por ejemplo...

```
private void enviarBroadcast(boolean estado, String mensaje) {
    Intent intentLocal = new Intent(Intent.ACTION_SYNC);
    intentLocal.putExtra(EXTRA_RESULTADO, estado);
    intentLocal.putExtra(EXTRA_MENSAJE, mensaje);
    LocalBroadcastManager.getInstance(getContext()).sendBroadcast(intentLocal);
}

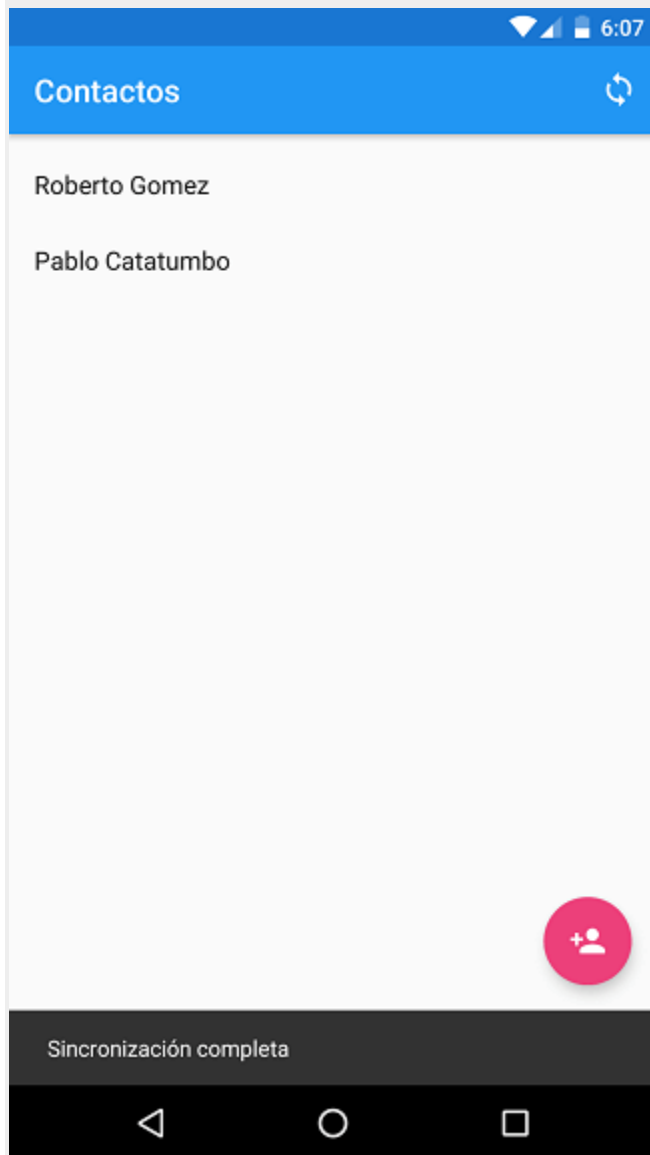
private void tratarPost() {
    // Desmarcar inserciones locales
}
```

```
procRemoto.desmarcarContactos(cr);  
enviarBroadcast(true, "Sincronización completa");  
}
```

El código anterior envía un intent con el mensaje «**Sincronización completa**» en caso de que la sincronización termine de forma satisfactoria. Recuerda enviar el mensaje con `LocalBroadcastManager.sendBroadcast()`.

Si percibes que tus mensajes no se reciben, intenta revisar que las acciones de los intents coincidan.

Al ejecutar la app con todos los puntos de envío de mensajes, podrás ver las snack bars de la siguiente forma.



## Conclusión

Los servicios REST son muy útiles para organizar tus prácticas de intercambio de datos entre aplicaciones. *¿Notaste la gran diferencia entre el uso de URIs para localizar recursos, en vez de enviar peticiones sin forma, hacia múltiples archivos Php?*

Recuerda que cada aplicación demanda un concepto diferente de sincronización de datos dependiendo del modelo de negocio o características del servicio que preste tu app a los usuarios. En mi caso, el ejemplo de PeopleApp puede funcionar sin conexión a internet y permite al usuario elegir en que momento sincronizar sus datos.

Cabe adicionar la importancia de las operaciones en batch en el servicio web para que tu aplicación envíe pocas peticiones HTTP hacia el servidor. Con este enfoque, es posible optimizar la batería del dispositivo móvil al momento de sincronizar.

Por último, te recomiendo investigar muy bien tus requerimientos de sincronización y las estrategias que aplicarás. Estos factores son fundamentales para elegir el rumbo de desarrollo que tomará tu proyecto y proteger la información de cada usuario.

## ¿Necesitas Otro Ejemplo De Servicio Web?

Hace unos días lancé un tutorial detallado para crear un servicio web REST para productos, clientes y pedidos. Donde consumo sus recursos desde una aplicación llamada App Productos. [Échale un vistazo](#) a todo lo que incluye (tutorial PDF, código completo en Android Studio, código completo PHP, script MySQL con 100 productos de ejemplo).

A promotional banner with a teal background. On the left, there is a black smartphone displaying a mobile application interface with a list of items. To the right of the phone, the text 'Tutorial: Crear Una App Android Para Productos, Clientes Y Pedidos' is written in white. Below this text, there is an orange rectangular button with the white text 'CONSÍGUELO YA'.

Tutorial: Crear Una App Android Para  
Productos, Clientes Y Pedidos

CONSÍGUELO YA

ShareTweet