

Universidad Tecnológica Nacional  
Facultad Regional Santa Fe

Departamento Ingeniería en sistemas de Información

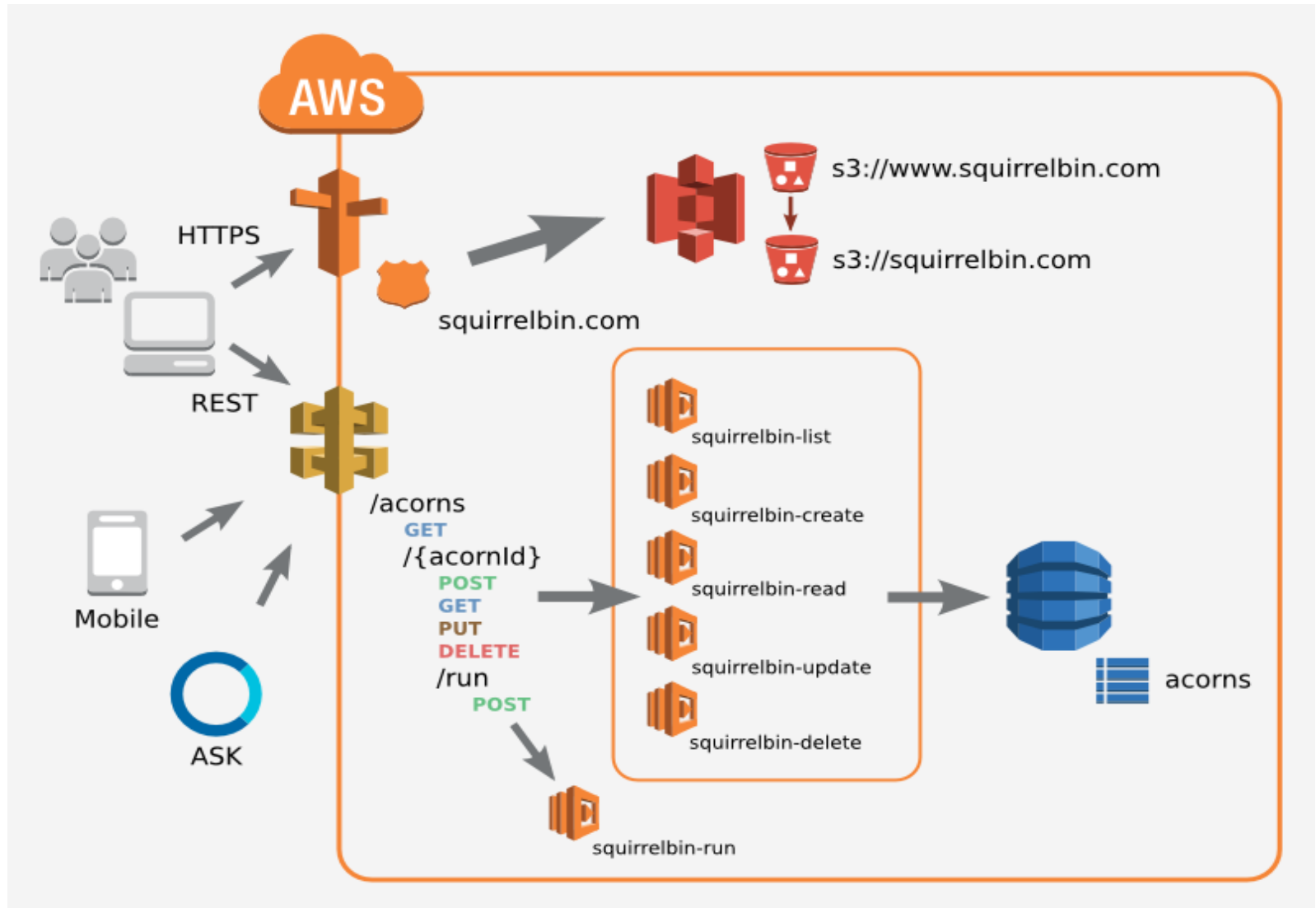
Desarrollo de aplicaciones móviles

# PERSISTENCIA

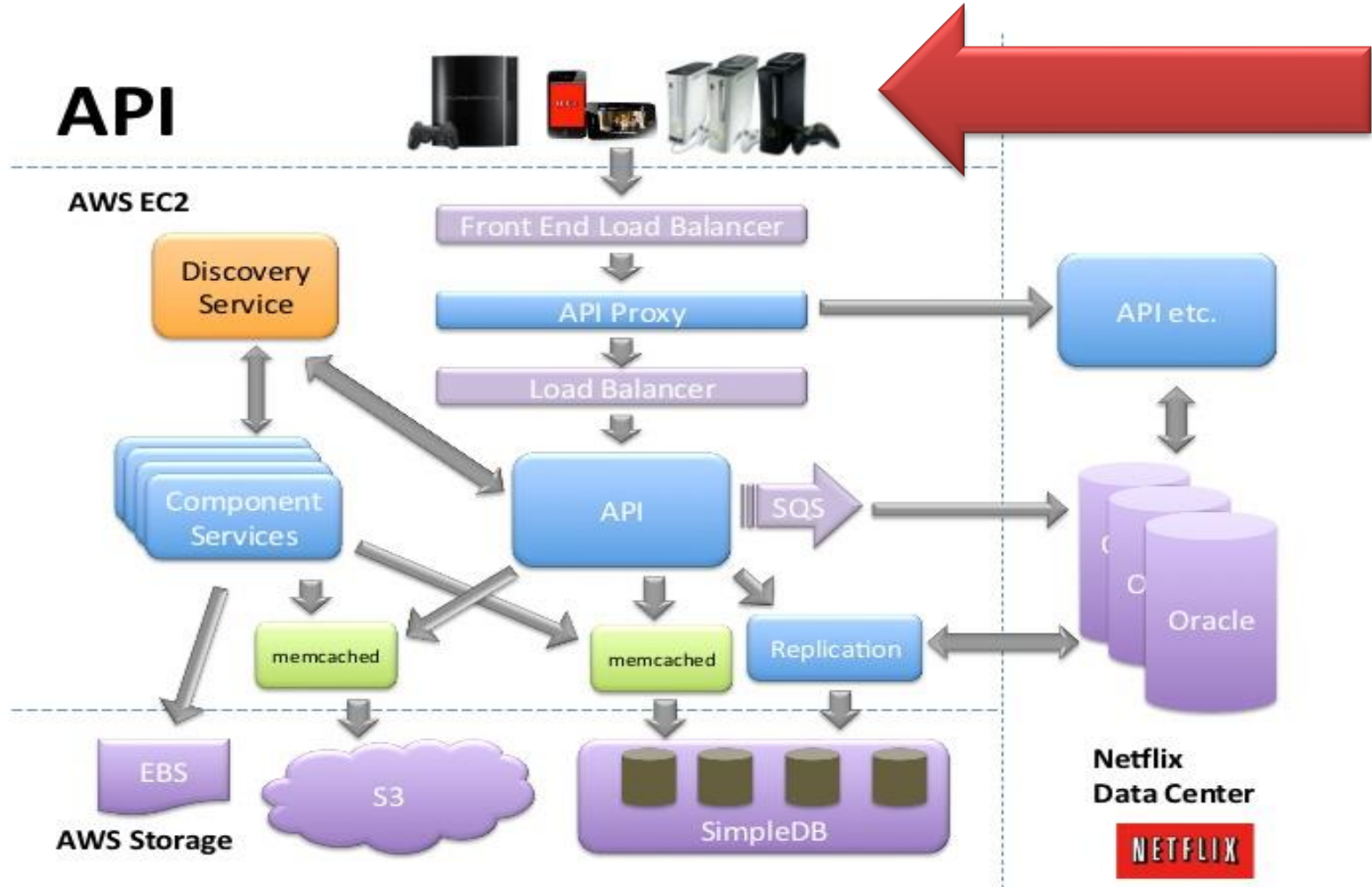
# Interactuar con datos en la nube

API Rest

# Arquitecturas basadas en la nube



# Ejemplo - Netflix



## Arquitecturas distribuidas en la nube

- Los desafíos que presentan son múltiples y todos tienen herramientas que permiten abordar a soluciones.
- Desde la perspectiva del desarrollo de una aplicación móvil, la cuestión mas importante es, “como nos comunicamos”
  - Protocolo: HTTP
  - Carga útil de datos: JSON / XML / Otros tipos mime.

# El protocolo HTTP

- Hypertext Transfer Protocol (HTTP) es un protocolo de capa de aplicación para transmisión de documentos.
- Fue diseñado para la comunicación entre los navegadores y servidores web, aunque hoy en día se usa para muchos otros propósitos.
- Sigue el modelo cliente-servidor
  - El cliente establece una conexión, realizando una petición a un servidor y espera una respuesta del mismo.
  - Se trata de un protocolo **sin estado**, lo que significa que el servidor no guarda ningún dato (estado) entre dos peticiones.
  - Generalmente usa TCP/IP, pero puede ser usado sobre cualquier capa de transporte segura o de confianza.

## Mensajes HTTP - Petición

method      URI      http version

POST /create-user HTTP/1.1

Host: localhost:3000

Connection: keep-alive

Content-type: application/json

{ "name": "John", "age: 35 }

} header

} body

# Mensajes HTTP – Partes de la Petición

- Un método HTTP, puede ser:
  - GET: el mensaje no tiene cuerpo, solo pide al servidor como respuesta los datos de un recurso representado por una URI
  - POST: envia en el cuerpo los nuevos datos con los que se debe crear un recurso en la URL dada
  - PUT : envia en el cuerpo los nuevos datos con los que se debe crear un recurso en la URL dada
  - DELETE es un mensaje sin cuerpo que solicita borrar en el servidor el recurso con la URL dada.
- El recurso pedido se identifica por una URL
- Las cabeceras HTTP opcionales, que pueden aportar información adicional a los servidores.
- Opcionalmente algunos métodos como POST envían un cuerpo de mensaje con carga útil.



## Ejemplo Petición HTTP

- Petición para pedir datos a un servidor del recurso “categoría”

```
GET 5000/categoria
cache-control: no-cache
postman-token: 108d6ce5-2a8b-45c2-80eb-
e62428f1ff5d
user-agent: PostmanRuntime/7.3.0
accept: */*
host: localhost:5000
accept-encoding: gzip, deflate
```

El encabezado de respuesta  
“content-type” me dice  
como tengo que interpretar  
los datos enviados en el  
cuerpo de la respuesta

```
HTTP/1.1 200
status: 200
x-powered-by: Express
x-content-type-options: nosniff
content-type: application/json; charset=utf-8
content-length: 192
connection: keep-alive
[ { "id": 1, "body": "Categoria 1" }, { "id": 2, "body": "Categoria 2" }, { "body":
"categoria3", "id": 3 }, { "body": "categoria3", "id": 4 } ]
```

## Petición para crear un recurso

**POST 5000/categoria**

*content-type: application/json*

*cache-control: no-cache*

*user-agent: PostmanRuntime/7.3.0*

*accept: \*/\**

*host: localhost:5000*

*accept-encoding: gzip, deflate*

*content-length: 21*

**{"body":"categoria3"}**

*El encabezado content-type le dice al servidor como interpretar los datos del body*

HTTP/1.1 201

status: 201

x-powered-by: Express

vary: Origin, X-HTTP-Method-Override, Accept-Encoding

access-control-allow-credentials: true

cache-control: no-cache

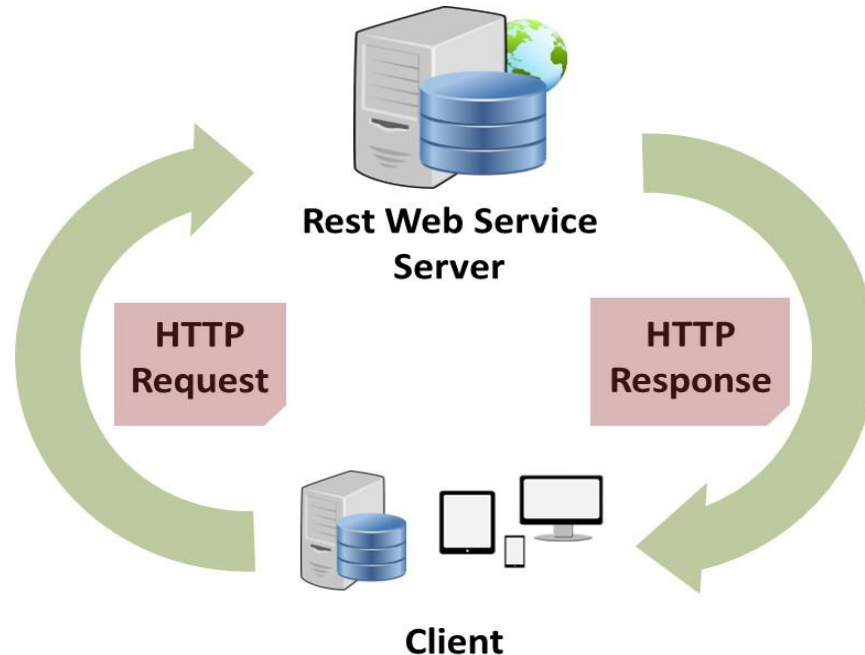
pragma: no-cache

expires: -1

access-control-expose-headers: Location

# Aplicaciones Distribuidas con Interface Mobile

- Por lo general las aplicaciones cuya UI está en JavaScript, distribuyen gran parte de su lógica de negocio y de acceso a datos, en un “back-end” que habitualmente ejecuta en un servidor.
- La comunicación por lo general se establece a través del protocolo HTTP y el contenido más habitual de intercambio es mensajes JSON o XML



# Aplicaciones Distribuidas con Interface Mobile

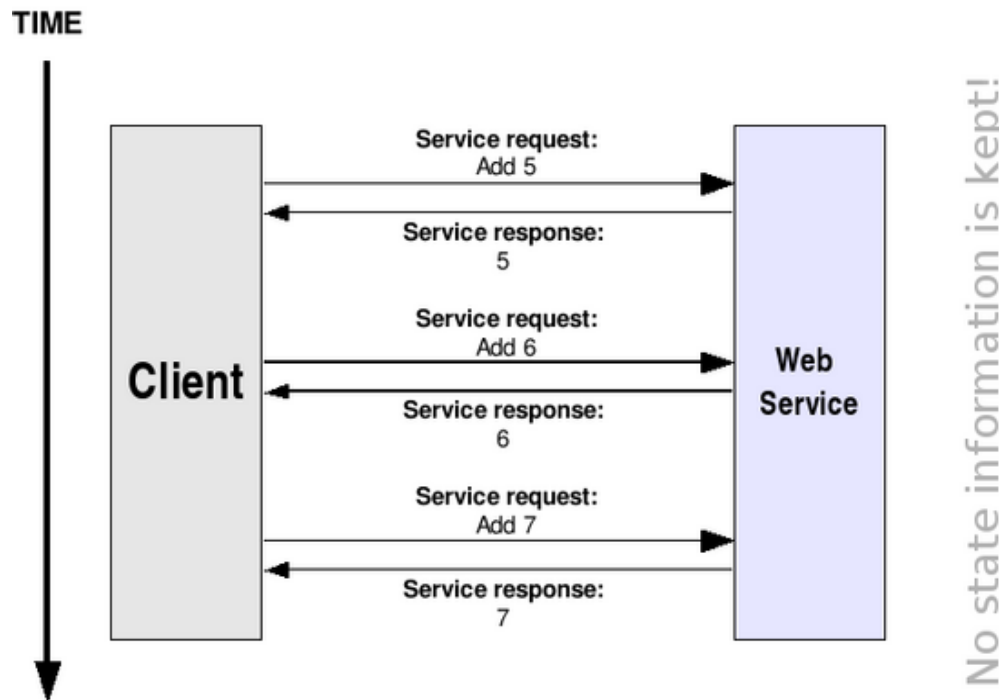
- Por lo general las aplicaciones cuya UI está en Aplicaciones Mobiles, distribuyen gran parte de su lógica de negocio y de acceso a datos, en un “back-end” que habitualmente ejecuta en un servidor.
- La comunicación por lo general se establece a través del protocolo HTTP y el contenido más habitual de intercambio es mensajes JSON o XML.
- El servidor actualmente suele alojarse en proveedores de infraestructura en la nube (AWS / Azure / Google Cloud / Cloudera / etc) que permiten configurar una arquitectura elástica que escale en sus prestaciones según la intensidad de la carga de trabajo, con servicio distribuido, tolerante a fallas.

# Aplicaciones Distribuidas con Interface Mobile

- Actualmente para desarrollar en la nube se sigue un paradigma de IaaS (Infrastructure as a Code) con herramientas como Docker y Kubernetes
- Las aplicaciones cloud native poseen arquitecturas basadas en microservicios.
- Pero en términos generales, la nube no es mas que acceder a un servidor remoto a través de un protocolo de red (generalmente HTTP)

# Comunicacion Mobile / Backend a través de servicios web

- Un servicio web es el par de aplicaciones “cliente” y “servidor” que se comunican a través del protocolo estándar de la web, HTTP, para realizar las tareas necesarias que el proceso o la aplicación cliente le solicita a la aplicación servidor.



# EL API Rest

- REST es un estilo arquitectural de definir servicios web que usa los estándares de la web.
  - Está disponible en su implementación en la mayoría de los servidores de aplicaciones (PHP / Java / NodeJS / .NET / etc).
  - Se adaptaron mejor a los nuevos frameworks, a la evolución de JavaScript y a HTML5 que SOAP.
  - Además son más sencillos de producir y consumir.
- Sus principales principios son:
  - Todo es un recurso que puede ser identificado por una URL
  - Un recurso puede ser representado por múltiples formatos, definido adecuadamente por el tipo MIME correspondiente. Se usan métodos estándar entre el cliente y el servidor para negociar el contenido.
  - Usa todo el protocolo HTTP para intercambiar información.
  - La comunicación es sin estado.

# REST

- Su nombre deriva de “Transferencia de Estado Representacional (REST - Representational State Transfer)” .
- La idea principal de rest es que a través de una URL publicamos recursos.
- ¿Qué son los recursos? Entidades del dominio cuyo estado en un momento dado puede ser accedido para escritura o lectura a través de una URL usando el protocolo HTTP.
- A su vez cada recurso posee un estado interno, que no puede ser accedido directamente desde el exterior. Lo que sí es accesible desde el exterior es una o varias representaciones de dicho estado. Por representación se entiende un formato de datos concreto usado para la transferencia de una copia del estado público del recurso entre el cliente y el servidor.



# REST

- La arquitectura Rest se basa en 4 principios fundamentales
  - utiliza los métodos HTTP de manera explícita
    - se usa POST para crear un recurso en el servidor
    - se usa GET para obtener un recurso
    - se usa PUT para cambiar el estado de un recurso o actualizarlo
    - se usa DELETE para eliminar un recurso
    - Se usa PATCH para actualizar un dato de un recurso
  - no mantiene estado
  - Una sintaxis universal para identificar los recursos. En un sistema REST, cada recurso es direccionable únicamente a través de su URI.
  - transfiere XML, JavaScript Object Notation (JSON), o ambos

# REST

- De acuerdo al tipo MIME solicitado como respuesta (Header HTTP content-type), el servicio web nos retornará la representación adecuada del objeto.
- No existe una única representación para un objeto. Por ejemplo la siguiente URL
  - GET /api/v2/empleado/1
- Deberá retornar distintas representaciones según el content-type sea:
  - Application/json: los datos del empleado en formato JSON
  - Application/xml : los datos del empleado en formato XML
  - Application/html : informe HTML del empleado
  - Application/png : la foto del empleado
  - Application/pdf : un reporte con la ficha del empleado

## Conexiones HTTP en Android

- Para cualquier operación que requiera trabajar con una conexión de red, nuestra aplicación necesita incluir los siguientes permisos
  - `<uses-permission android:name="android.permission.INTERNET" />`
  - `<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />`
- Hasta la versión 6 de Android, se usaban dos librerías para gestionar a través del código y programáticamente la conexión a internet
  - `HttpClient` (de apache)
  - `URLConnection`
- A partir de la versión 6 `HttpClient` sigue estando disponible pero está deprecada por lo que se recomienda el uso de `URLConnection` para interactuar con servidores HTTP.

# Funcionamiento

- Obtener una nueva instancia de `URLConnection` invocando a `URL.openConnection()` y casteando el resultado a `URLConnection`.
- Preparar el request. La propiedad primaria para un request es la URI.
- Luego setear el método, por defecto ejecutará con métodos GET
  - si queremos usar POST o PUT usar la propiedad `setDoOutput(true)`
  - Para DELETE o PUT, (o también para el resto de los métodos) se puede setear `setRequestMethod(String)`.
  - Incluir luego request headers, credenciales, tipos de respuesta preferidos, y cookies.
- Para recibir datos, obtener un `InputStream`.
- Para transmitir datos, utilizar `getOutputStream()`.
- Procesar la respuesta.
- Desconectar.

# Abrir conexiones

Desde el emulador la IP de la computadora local es 10.0.2.2

```
URLConnection urlConnection=null;
try {
    URL url = new URL("http://192.168.0.105:3000/posts/1");
    conn= (URLConnection) url.openConnection();
    InputStream in = new BufferedInputStream(conn.getInputStream());
    InputStreamReader isw = new InputStreamReader(in);
    StringBuilder sb = new StringBuilder();
```

```
int data = isw.read();
while (data != -1) {
    char current = (char) data;
    sb.append(current);
    data = isw.read();
}
```

Leer la respuesta del servidor en un string.  
Lo que leemos es el BODY de la respuesta HTTP

## Ejemplo de lectura - GET

```
private void leerNoticias(){
    HttpURLConnection urlConnection=null;
    try {
        URL url = new URL("http://192.168.0.105:3000/posts/1");
        urlConnection= (HttpURLConnection) url.openConnection();
        InputStream in = new BufferedInputStream(urlConnection.getInputStream());
        InputStreamReader isw = new InputStreamReader(in);
        StringBuilder sb = new StringBuilder();

        int data = isw.read();
        while (data != -1) {
            char current = (char) data;
            sb.append(current);
            data = isw.read();
        }
        Log.d("TEST-ARR", sb.toString());
    }
    catch (IOException e) {
        e.printStackTrace();
    }
    finally {
        if(urlConnection!=null) urlConnection.disconnect();
    }
}
```

```
{
  "id": 1,
  "title": "json-server",
  "author": "martin"
}
```

## Ejemplo de escritura POST

```
1. HttpURLConnection urlConnection=null;
2.  try {
3.      JSONObject nuevaNoticia = new JSONObject();
4.      nuevaNoticia.put("title", "creado desde android " + System.currentTimeMillis());
5.      nuevaNoticia.put("author", "martin");

1.      URL url = new URL("http://192.168.0.105:3000/posts/");
2.      urlConnection = (HttpURLConnection) url.openConnection();

1.      urlConnection.setDoOutput(true);
2.      urlConnection.setChunkedStreamingMode(0);
3.      urlConnection.setRequestMethod("POST");
4.      urlConnection.setRequestProperty("Content-Type","application/json");
```

## Ejemplo de escritura POST

```
1.  DataOutputStream printout = new DataOutputStream(urlConnection.getOutputStream());
2.      printout.writeBytes(URLEncoder.encode(nuevaNoticia.toString(),"UTF-8"));
3.      printout.flush ();
4.      printout.close ();
5.  }catch (JSONException e2) {
6.      e2.printStackTrace();
7.  } catch (IOException e1) {
8.      e1.printStackTrace();
9.  }finally {
10.      if(urlConnection != null) urlConnection.disconnect();
11.  }
```



## Ejemplo de escritura POST

```
1.  DataOutputStream printout = new
    DataOutputStream(urlConnection.getOutputStream());
2.      printout.writeBytes(URLEncoder.encode(nuevaNoticia.toString(),"UTF-8"));
3.      printout.flush ();
4.      printout.close ();
5.  }catch (JSONException e2) {
6.      e2.printStackTrace();
7.  } catch (IOException e1) {
8.      e1.printStackTrace();
9.  }finally {
10.     if(urlConnection != null) urlConnection.disconnect();
11. }
```

# Retrofit

- Retrofit es un cliente Rest para java y android.
- Simplifica la interacción con un servidor REST.
- Configuramos mediante anotaciones como ejecutar las operaciones.
- Permite hacer peticiones GET, POST, PUT, PATCH, DELETE y HEAD; gestionar diferentes tipos de parámetros y parsear automáticamente la respuesta a un POJO (Plain Old Java Object).
- Se integra con Gson o Jackson para parsear los datos en JSON

## Configurar dependencias

- En el archivo Build.gradle(de Module:app)
  - compile 'com.squareup.retrofit2:retrofit:2.0.2'
  - compile 'com.squareup.retrofit2:converter-gson:2.0.2'
    - *En este caso se usa el converter gson pero podríamos configurar Jackson.*
- Para trabajar con Retrofit se necesitan crear 3 clases.
  - Clase modelo que se utiliza para asignar los datos JSON a
  - Interfaces que definen las posibles operaciones HTTP.
    - Cada método de una interfaz representa una posible llamada a la API. Debe tener una anotación HTTP (GET, POST, etc.) para especificar el tipo de solicitud y la URL relativa. El valor de retorno ajusta la respuesta en un objeto de llamada con el tipo del resultado esperado.
  - Clase Retrofit.Builder: instancia que utiliza la interfaz y la API Builder que permite definir el punto final de URL para la operación HTTP.

# Configurar una interface con métodos REST

- Configuramos una interface con:
  - Anotaciones @GET / @POST / @PUT / @DELETE dependiendo lo que deseamos que realice el método.
  - Las respuestas retornan un objeto Call
  - Cuando pasamos como carga útil un objeto lo marcamos como @Body

```
import retrofit2.Call;
import retrofit2.http.Body;
import retrofit2.http.GET;
import retrofit2.http.POST;

public interface ProyectoRest2 {
    @GET("proyectos/")
    Call<List<Proyecto>> listarTodos();

    @POST("proyectos/")
    Call<Proyecto> crearProyecto(@Body Proyecto p);
}
```

## Otras anotaciones

- PatParam: en las API Rest, los path param, son las partes variables de una URL. Por ejemplo
  - <http://server/api/clientes/99>
  - <http://server/api/clientes/100>

```
@GET("clientes/{id}") Call<List<Cliente>> clienteById(@Path("id") String idCliente)
```

QueryParam: en las API Rest, los query param, son las partes variables de una URL que van luego del ? Separadas por & como clave valor

<http://server/api/productos?stockMinimo=40&precioMax=99>

```
@GET("productos")  
Call<Producto> buscarProdPrecioStock(  
    @Query("stockMinimo") Integer id, @Query("precioMax") Integer max)
```

## Crear una instancia de retrofit

- Para poder usar la interface construida, debemos pedirle a retrofit que nos retorne una instancia implementada.
- Para ello en primer lugar debemos configurar retrofit
  - Crear el conversor Java  $\leftrightarrow$  JSON
  - Configurar el conversor
  - Configurar la url de endpoint
  - Obtener la instancia de la interface

## Ejemplo

```
Gson gson = new GsonBuilder()
    .setLenient()
    .create();

Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("http://10.0.2.2:5000/")
    .addConverterFactory(GsonConverterFactory.create(gson))
    .build();

proyectoRest = retrofit.create(ProyectoRest2.class);
```

## Ejecutar

- Ahora con la interface podemos invocar cualquier método generado.
- `proyectoRest.listarTodos();`
- Retorna un objeto de tipo `Call<T>` donde el tipo genérico `T` es el objeto que retorna el recurso resto.
- El objeto `call`, **no ejecuta la llamada al API Rest, sino que debemos hacerlo con una de las siguientes opciones**
  - Sincrónico: `execute`
    - <https://square.github.io/retrofit/2.x/retrofit/retrofit2/Call.html#execute-->
  - Asincrónico: `enqueue`
    - <https://square.github.io/retrofit/2.x/retrofit/retrofit2/Call.html#enqueue-retrofit2.Callback->



## Ejemplo con execute

- Ejecuta la petición HTTP en el momento en que el método se invoca.
- Cuando obtiene la respuesta retorna un objeto Response que representa una respuesta HTTP
- Debe invocarse siempre en un hilo secundario.

```
public List<Proyecto> getAll() {  
    Call<List<Proyecto>> invocacionSyn = proyectoRest.listarTodos();  
    Response<List<Proyecto>> res = null;  
    try{  
        res = invocacionSyn.execute();  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
    return res.body();  
}
```

# El objeto Response

<a href="#"><u>T</u></a>	<a href="#"><u>body()</u></a> The deserialized response body of a <a href="#"><u>successful</u></a> response.
int	<a href="#"><u>code()</u></a> HTTP status code.
static <T> <a href="#"><u>Response</u></a> <T>	<a href="#"><u>error</u></a> (int code, okhttp3.ResponseBody body)Create a synthetic error response with an HTTP status code of code and body as the error body.
static <T> <a href="#"><u>Response</u></a> <T>	<a href="#"><u>error</u></a> (okhttp3.ResponseBody body, okhttp3.Response rawResponse)Create an error response from rawResponse with body as the error body.
ResponseBody	<a href="#"><u>errorBody()</u></a> The raw response body of an <a href="#"><u>unsuccessful</u></a> response.
Headers	<a href="#"><u>headers()</u></a> HTTP headers.
Boolean	<a href="#"><u>isSuccessful()</u></a> Returns true if <a href="#"><u>code()</u></a> is in the range [200..300).

# El objeto Response

<u>String</u>	<u>message</u> () HTTP status message or null if unknown.
Response	<u>raw</u> () The raw response from the HTTP client.
static <T> <u>Response</u> <T>	<u>success</u> (T body) Create a synthetic successful response with body as the deserialized body.
static <T> <u>Response</u> <T>	<u>success</u> (T body, okhttp3.Headers headers) Create a synthetic successful response using headers with body as the deserialized body.
static <T> <u>Response</u> <T>	<u>success</u> (T body, okhttp3.Response rawResponse) Create a successful response from rawResponse with body as the deserialized body.

## Ejecutar asincrónicamente

- El método enqueue encola el request para ejecutarlo asincrónicamente en un hilo secundario.
- Recibe como argumento una función de callback que se ejecutará cuando se obtenga la respuesta del API Rest.

```
Call<List<Proyecto>> reqAsyn =ProyectoRepository
    .getInstance(this).getAllAsyn();
reqAsyn.enqueue(new Callback<List<Proyecto>>() {
    @Override
    public void onResponse(Call<List<Proyecto>> call,
        Response<List<Proyecto>> response) {

    }

    @Override
    public void onFailure(Call<List<Proyecto>> call, Throwable t) {

    }
});
```

## Detalle de procesamiento de la respuesta

```
public void onResponse(Call<List<Proyecto>> call,
                    Response<List<Proyecto>> response) {
    switch (response.code()) {
        case 200:
            List<Proyecto> data = response.body();
            adaptador.notifyDataSetChanged(data);
            break;
        case 401:
        case 403:
        case 500:
            break;
        default:
            break;
    }
}
```

**SQLITE**

## SQLite

- Si tenemos que guardar unos pocos valores, es suficiente con "Shared Preferences".
- Si lo que se quiere es guardar valores de configuración de la aplicación, se debe diseñar la actividad de configuración extendiendo la clase "PreferenceActivity" que se encargará de cargar y guardar los valores automáticamente. (utilizando Shared Preferences, pero en este caso, esto es transparente.)
- Si se quiere guardar unos cuantos más datos se debe usar SQLite.

# SQLite

- Las aplicaciones de Android pueden utilizar bases de datos SQLite para almacenar su estado.
- SQLite es una base de datos relacional y embebida
  - Combina una interface SQL limpia, con un bajo uso de memoria y una velocidad “decente”.
  - Es de dominio público, por lo que puede ser usado tanto por empresas como por proyectos opensource.
- Para Android, SQLite está incluido en el runtime de Android, por lo que cualquier aplicación Android puede crear bases de datos SQLite.
  - No requiere configuración, no tiene un servidor de base de datos ejecutándose en un proceso separado.
  - Android soporta SQLite mediante el paquete de Java `android.database.sqlite`.



# SQLite

- SQLite, como lo sugiere su nombre, usa un dialect de SQL para consultas de manipulación (SELECT, INSERT, UPDATE.), y definición de datos (CREATE TABLE...etc).
- SQLite tiene diferencias mínimas con los estándares SQL-92 y SQL-99.
- Es eficiente en el uso de espacio, lo que permte que Android lo incluya en su runtime de manera completa (sin necesidad de elegir un subconjunto arbitrario de funcionalidades).
- La principal diferencia es el tipado de datos, o la ausencia del mismo.

# SQLite

- Aunque podemos especificar tipos de datos para columnas en una sentencia CREATE TABLE, SQLite lo usará como un “hint” pero en una columna de cualquier tipo podemos poner el dato que querramos.
- ¿Se puede poner un String en una columna definida como Integer? → Si!! Y viceversa también!!
- Una de las características de SQLite es que no se asigna un tipo de datos a la columna, sino que se asigna a cada valor.

# SQLite

- Las siguientes son algunas clases que debemos conocer y entender para poder utilizar de manera efectiva la API de SQLite.
  - `android.database.sqlite.SQLiteDatabase`
  - `android.database.sqlite.SQLiteDatabase.CursorFactory`
  - `android.database.sqlite.SQLiteQueryBuilder`
  - `android.content.ContentValues`
  - `android.database.Cursor`
  - `android.database.SQLException`
  - `android.database.sqlite.SQLiteOpenHelper`

# SQLite

- Aunque definamos una columna como entero, podemos insertar un string.
- Tipos de datos:
  - NULL: el valor que va a contener la columna es NULL.
  - INTEGER: pueden almacenarse enteros, de 1,2,3,4,6, o 8 bytes con signo.
  - REAL: valores de punto flotante.
  - TEXT: se trata de un string.
  - BLOB: para valores BLOB( objetos binarios grandes) como imágenes, archivos multimedia, etc.

# SQLite

- No existen los boolean, como tal. Estos se almacenan como INTEGER con el valor 0(falso) o 1(verdadero).
- No existen campos fecha y hora.
- Se pueden almacenar como TEXT, REAL o INTEGER.
- En función del tipo elegido se almacenará en distintos formatos, por ejemplo, si es TEXT, el dato tendrá el formato "YYYY-MM-DD HH:MM:SS.SSS".

- Android no nos provee ninguna base de datos por defecto.
  - Si queremos usar SQLite, se necesitará crear una propia base de datos, y luego llenarla con nuestras tablas, índices y datos.
- Para crear y abrir una base de datos, la mejor opción es crear una subclase de **SQLiteOpenHelper**.
- Esta clase oculta o abstrae la lógica para crear y actualizar una base de datos.

# SQLite

- Si extendemos SQLiteOpenHelper disponemos de varios métodos:
  - En el constructor llamar al método padre indicando el nombre de la base de datos y un cursorFactory.
  - El método onCreate() será ejecutado automáticamente por nuestra clase cuando sea necesaria la creación de la base de datos, es decir, cuando aún no exista.
    - Las tareas típicas que deben hacerse en este método serán la creación de todas las tablas necesarias y la inserción de los datos iniciales si son necesarios.
  - Los métodos onUpgrade y onDowngrade, que ejecutan las sentencias necesarias para que la base de datos pase a la versión solicitada.

## Ejemplo

```
1. public class MyDatabaseHelper extends SQLiteOpenHelper {  
2.     private static final String DATABASE_NAME="db";  
3.     static final String TITLE="title";  
4.     static final String VALUE="value";  
5.     public DatabaseHelper(Context context) {  
6.         super(context, DATABASE_NAME, null, 1);  
7.     }  
8.     @Override  
9.     public void onCreate(SQLiteDatabase db) {  
10.         db.execSQL("create table producto (_id integer primary key autoincrement, title text,  
value real);");  
11.         ContentValues cv=new ContentValues();  
12.         cv.put("title", "abc");  
13.         cv.put("value", 99.99);  
14.         db.insert("constants", null, cv);  
15.     }
```



# Ejemplo

```
1. @Override
2. public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
3.     switch(oldVersion) {
4.         case 1:
5.             //upgrade logic from version 1 to 2
6.             db.execSQL("drop table if exists producto ");
7.             onCreate(db);
8.         case 2:
9.             //upgrade logic from version 2 to 3
10.        case 3:
11.            //upgrade logic from version 3 to 4
12.            break;
13.        default:
14.    }}
15. }
```

- En el método `onCreate`, recibimos una instancia de la base de datos creada y disponemos del método `execSQL()` que nos permite ejecutar las consultas para creación y manipulación de tablas.
  - Este método se limita a ejecutar directamente el código SQL que le pasemos como parámetro.
- El método `onUpgrade()` se lanzará automáticamente cuando sea necesaria una actualización de la estructura de la base de datos o una conversión de los datos. Si agregamos una columna por ejemplo.

## Ejemplo

- El método `onUpgrade()` cuando intentemos abrir una versión concreta de la base de datos que aún no exista recibe, como parámetros la versión actual de la base de datos en el sistema, y la nueva versión a la que se quiere convertir.
- En función de esta pareja de datos necesitaremos realizar unas acciones u otras.
- Por ejemplo crear una tabla con 3 columnas, pasar los datos a la tabla nueva y luego borrar una tabla con 2 columnas. O directamente si es posible aplicar un `Alter Table` sobre la tabla con 2 columnas.

## Ejemplo

- Cuando en el constructor invocamos la creación de una base de datos, **el último parámetro que pasamos es la versión de la base de datos.**
- En el método `onUpgrade()`, para que el mismo funcione apropiadamente, debemos aumentar en 1 la versión.
- El método `onDowngrade()`, que fue introducido en Android 3.0, es invocado si el código requiere un esquema anterior al actualmente en uso.
- Es el método inverso al método `onUpgrade`.
- En general tampoco es necesario implementarlo, pero podríamos hacerlo para mantener compatibilidad hacia atrás.

## Usar la base de datos

- Para usar nuestra propia instancia de SQLiteOpenHelper, debemos simplemente crear una instancia y mantener un puntero a ella.
- Cuando necesitamos operar con la base de datos invocamos sobre esta instancia uno de los siguientes métodos:
  - Si vamos a consultarla `getReadableDatabase()`
  - Si vamos a modificarla `getWritableDatabase()`.
- Por ejemplo en un método de una actividad, podríamos realizar la siguiente operación
  - **`MyDatabaseHelper db = new MyDatabaseHelper(...);`**
  - `constantsCursor=db.getReadableDatabase()`
  - `.rawQuery("select _id, title, value from productos order by title", null);`
- Cuando terminamos de trabajar con la base de datos, o a más tardar cuando la actividad es cerrada, simplemente sobre la instancia de SQLiteOpenHelper invocamos al método `close()`;

## execSQL y DDL

- Para crear tablas e índices también es necesario pasar las consultas DDL como argumento al método `execSQL()`. Es un método sin retorno. Ejemplo
  - `db.execSQL("create table constants (_id integer primary key autoincrement, title text, value real);");`
  - *Crearé una table denominada "constants" con una clave primaria llamada \_id de tipo autoincrementeo y dos columnas titulo de tipo texto y valor de tipo real.*
- SQLite automáticamente creará un índice para la columna de clave primaria.
- Podemos agregar otros índices con sentencias `CREATE INDEX`.
- También con `execSQL` se puede invocar a `DROP INDEX` y `DROP TABLE` en caso de que sea necesario (en el momento de un upgrade).

# Insertar Datos

- Existen dos enfoques para insertar datos en una base de datos
  - Mediante `execSQL()` pasando como parámetro sentencias INSERT, UPDATE, y DELETE
  - Usar los métodos `insert()`, `update()`, y `delete()` del objeto `SQLiteDatabase` que permite simplificar la sintaxis y reducir la complejidad y “verbosity” de SQL

long	<a href="#"><code>insert(String table, String nullColumnHack, ContentValues values)</code></a>
Int	<a href="#"><code>delete(String table, String whereClause, String[] whereArgs)</code></a>
Int	<a href="#"><code>update(String table, ContentValues values, String whereClause, String[] whereArgs)</code></a>

## Content Values

- Implementa una interface de tipo Map, para trabajar con tipos de SQLite.
  - Tiene el método get(k) que retorna un valor por su clave, pero también podemos usar getAsInteger(), getAsFloat(), getAsString(), getAsByteArray() o getAsByte().
- <http://developer.android.com/reference/android/content/ContentResolver.html>



## Ejemplo de inserción

```
1. private void processAdd(Producto p) {  
2. ContentValues values=new ContentValues(2);  
3. values.put("title", p.getTitle());  
4. values.put("value", p.getValue());  
5. db.getWritableDatabase().insert("constants", "title", values);  
6. constantsCursor.requery();  
7. }
```

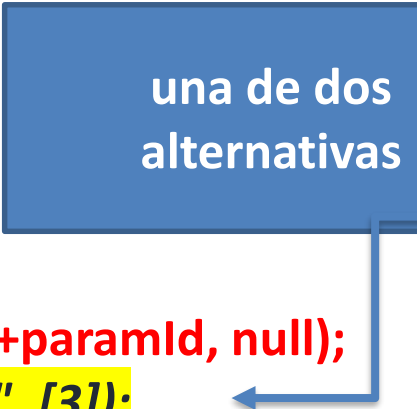
- El método insert, contiene 3 parámetros, el nombre de la tabla, el nombre de al menos una columna (nullColumnHack) que puede ser null, y la colección de datos a insertar (ContentValues ).
- **“NullColumnHack”**
  - En algunas bases de datos esto puede ser válido
    - INSERT INTO foo;
  - No así en SQLite, donde nos tenemos que asegurar que las sentencias que se generen tengan al menos una columna especificada
    - INSERT INTO foo (somecol) VALUES (NULL);

# Update

- El método `update()` tiene la siguiente sintaxis
  - `update(String table, ContentValues values, String whereClause, String[] whereArgs)`
- Toma como argumento la table a actualizar, un `ContentValues` que representa las columnas y los reemplazos a realizar y una cláusula `WHERE` opcional donde podemos definir a que filas aplicarle la actualización y un arreglo de parámetros opcional para rellenar la cláusula `where` usando como parámetros a reemplazar los signos de pregunta (?). los cuales son reemplazados de manera posicional.

## Ejemplo de update

```
1. conn= db.getWritableDatabase
2. ContentValues registro=new ContentValues();
3. registro.put("nombre","Producto 1");
4. registro.put("precio",10.5);
5. registro.put("stock",100);
6. int paramId=3;
7. int cant = conn.update("productos", registro, "id="+paramId, null);
8. int cant = conn.update("productos", registro, "id=?", [3]);
9. int cant = conn.update("productos", registro, "stock>? AND precio>?", [3,110.0]);
1. conn.close();
```



una de dos alternativas

## Borrar

- El método delete funciona de manera similar, salvo que no debemos especificar ContentsValues.
  - `public int delete (String table, String whereClause, String[] whereArgs)`
- Simplemente el nombre de la table si queremos borrar todos los registros.
- O una clausula where si teneos alguna condición.

```
private void processDelete(long rowId) {  
    String[] args={String.valueOf(rowId)};  
    db.getWritableDatabase().delete("constants", "_ID=?", args);  
    constantsCursor.requery();  
}
```

## Consultas con Raw Queries

- La solución más simple para consultar datos, en términos de la facilidad de uso del API es `is rawQuery()`.
- Simplemente se la invoca pasando el `select` como parámetro.
- Puede incluir parametros posicionales que forman el Segundo argumento de "`rawQuery()`".
- `constantsCursor=conn.getReadableDatabase()`
- `.rawQuery("SELECT _ID, title, value "+`
- `"FROM constants ORDER BY title",null);`
- Retorna un cursor que permite iterar sobre los resultados.

## El método query

- Si las consultas son en parte dinámicas el enfoque anterior se puede volver complicado en determinadas situaciones incluso con el tratamiento de parámetros dinámicos
- También es costoso el hecho de no conocer en tiempo de compilación las columnas que puede ser necesario recuperar.
- El método query() toma las piezas de una sentencia SELECT y las compone.

```
public Cursor query (String table, String\[\] columns, String selection, String\[\] selection  
Args, String groupBy, String having, String orderBy, String limit)
```

# El método query

- Las piezas que compone en el orden en que aparecen como parámetros son
  - La tabla a consultar
  - La lista de columnas a recuperar
  - La clausula where, la cual opcionalmente puede incluir un parámetro posicional.
  - La lista de parametros de sustitución del where.
  - Cláusula GROUP BY
  - Cláusula HAVING
  - Cláusula ORDER BY

## El método query

- Todos los parámetros excepto el nombre de la tabla son opcionales:
  - `String[] columns={"ID", "inventory"};`
  - `String[] parms={"snicklefritz"};`
  - `Cursor result=db.query("widgets", columns, "name=?",parms, null, null, null);`



## Cursores

- Sea cual fuere el método que usamos para ejecutar consultas lo que recibimos como resultado es un Cursor.
- Con un cursor podemos:
  - Encontrar cuantas filas fueron retornadas a través del método `getCount()`
    - Este método puede ser lento, porque implica recuperar todas las filas de un resultado.
  - Iterar sobre las filas, a través de los métodos `moveToFirst()`, `moveToNext()`, `isBeforeFirst()` y `isAfterLast()`.
  - Conocer los nombres de las columnas a través del método `getColumnNames()`, o conocer la posición de dicho índice con `getColumnIndex()`

- Con un cursor podemos
  - Consultar el tipo de una columna con `getType()`
  - Obtener los valores de la fila actual, con métodos como `getString()`, `getInt()`, `getFloat()`, `getBlob()`
  - Re ejecutar la consulta que creo el cursor, via `requery()`
  - Liberar el cursor a través de `close()`

# Ejemplo

```
1. Cursor result=
2. db.rawQuery("select id, name, inventory from widgets", null);
3. while (result.moveToNext()) {
4.   int id=result.getInt(0);
5.   String name=result.getString(1);
6.   int inventory=result.getInt(2);
7.   // logica
8. }
9. result.close();
```

## Cerrar la conexión

- Dado que los métodos `getWritableDatabase()` y `getReadableDatabase()` son costosos cuando deben ejecutar y la base de datos está cerrada, se recomienda dejar la conexión abierta tanto como sea posible y probable que necesitemos acceder a la base de datos.
- Típicamente lo óptimo es cerrar la base de datos en la el método de ciclo de vida `onDestroy()` de una actividad o servicio.

```
@Override  
protected void onDestroy() {  
    mDbHelper.close();  
    super.onDestroy();  
}
```

# ROOM

- Actualmente el sitio de desarrollo de Android recomienda emplear alternativas de más alto nivel para gestionar la base de datos

## Save data using SQLite



Saving data to a database is ideal for repeating or structured data, such as contact information. This page assumes that you are familiar with SQL databases in general and helps you get started with SQLite databases on Android. The APIs you'll need to use a database on Android are available in the `android.database.sqlite` package.

- !** **Caution:** Although these APIs are powerful, they are fairly low-level and require a great deal of time and effort to use:
- There is no compile-time verification of raw SQL queries. As your data graph changes, you need to update the affected SQL queries manually. This process can be time consuming and error prone.
  - You need to use lots of boilerplate code to convert between SQL queries and data objects.

For these reasons, we **highly recommended** using the [Room Persistence Library](#) as an abstraction layer for accessing information in your app's SQLite databases.

# Room

- La biblioteca/librería de persistencia Room provee una capa de abstracción sobre SQLite para permitir un acceso a la base de datos robusto y flexible
- Permite guardar datos en caché, y mostrarle al usuario una vista consistente de los datos almacenados.
- Ha sido introducida por Android en la I/O 2017 junto con otros componentes arquitecturales.

## Configurar Room

- En el archivo "gradle.build" de nivel de proyecto agregar "google()" como repositorio.

```
allprojects {  
    repositories {  
        jcenter()  
        google()  
    }  
}
```

- En "gradle.build" de nivel de módulo agregar las dependencias

```
def room_version = "1.1.1"
```

```
implementation "android.arch.persistence.room:runtime:$room_version"
```

```
annotationProcessor "android.arch.persistence.room:compiler:$room_version" // use kapt for Kotlin
```

```
// optional - RxJava support for Room
```

```
implementation "android.arch.persistence.room:rxjava2:$room_version"
```

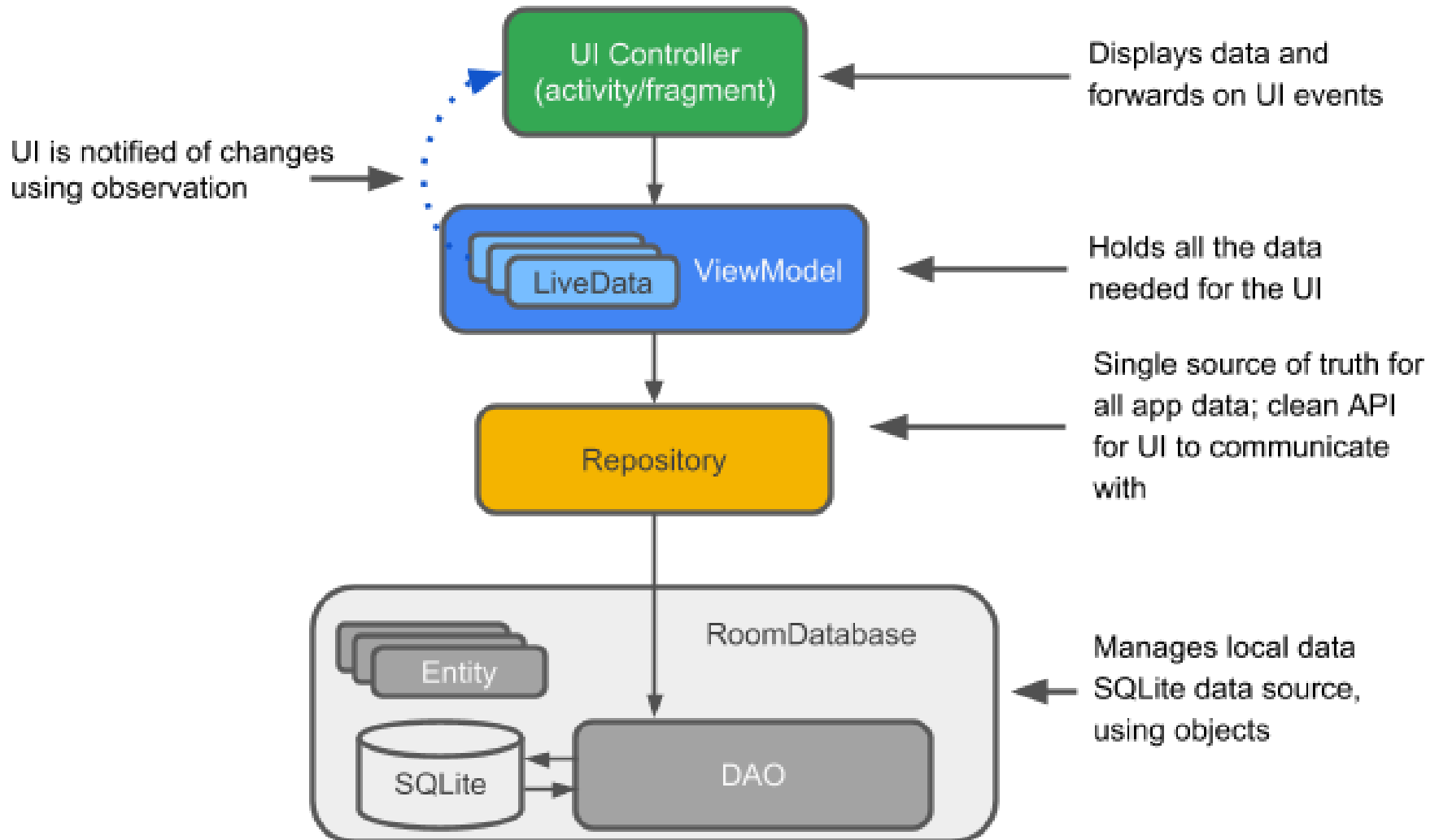
```
// optional - Guava support for Room, including Optional and ListenableFuture
```

```
implementation "android.arch.persistence.room:guava:$room_version"
```

```
// Test helpers
```

```
testImplementation "android.arch.persistence.room:testing:$room_version"
```

# Arquitectura

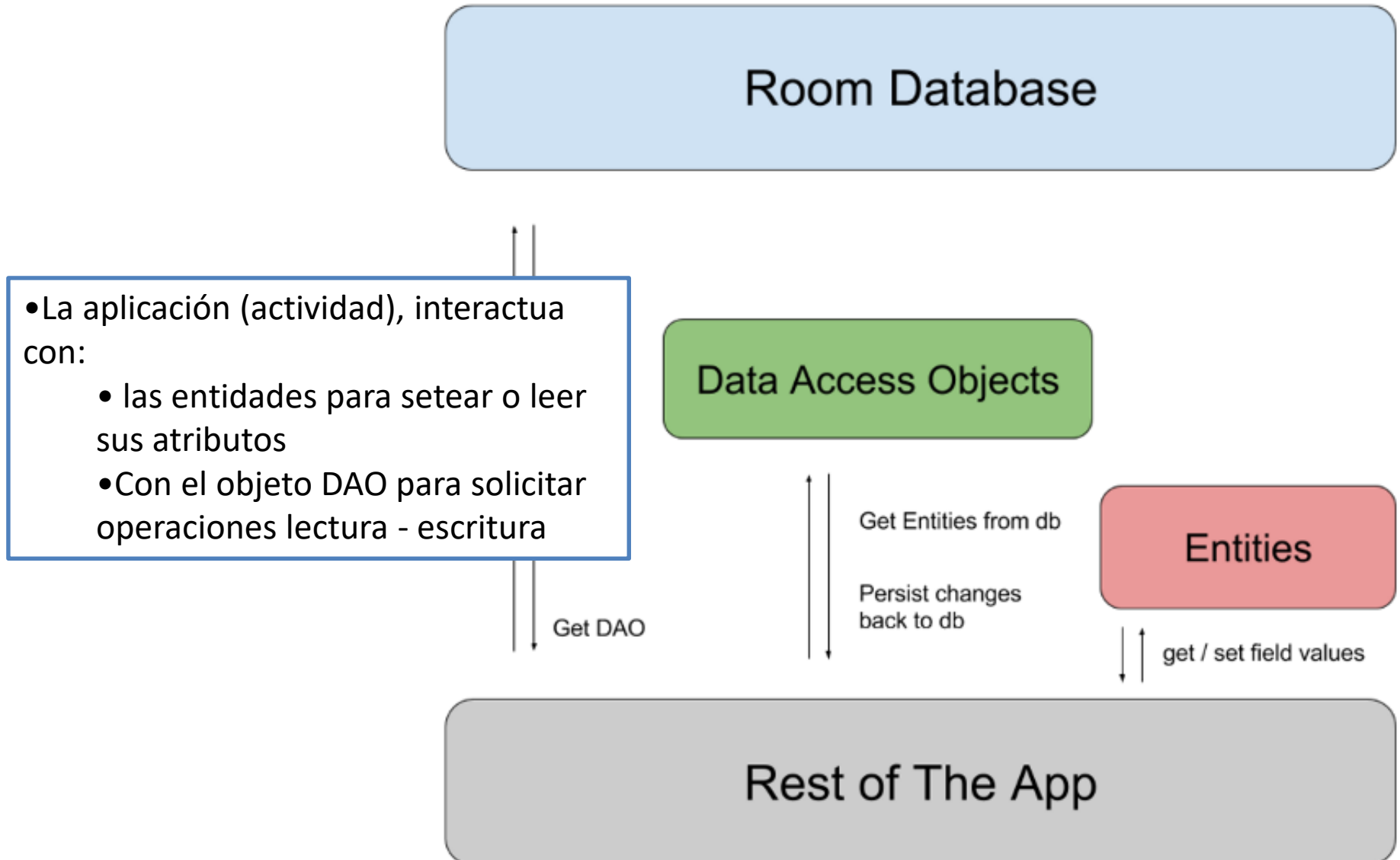




## Conceptos

- *Entity*: Es una clase java, "plana" (POJO), que posee anotaciones y que describe como se vincula dicha clase con una tabla en la base de datos y cada atributo con una columna de dicha tabla
- *SQLite database*: la base de datos donde Room almacena nuestros datos.
- *DAO*: acrónimo de *data access object* . Es una clase que contiene el mapeo de consultas SQL con métodos, de forma tal que en la ejecución de dichos métodos se ejecuten las setencias SQL asociadas.
- *Room database*: es una capa de bases de datos por sobre SQLite, que realiza todas las tareas de gestión de datos que como desarrollador evitamos al configurar las anotaciones.
- *Repository*: una clase que creamos para gestionar los múltiples orígenes de datos (Room, un API Rest, etc).

# Organización



# Entidades

- Cualquier clase java que represente una entidad (una entidad es una instancia de un objeto en memoria, cuyos atributos coinciden con los datos de una fila en la base de datos) debe ser marcada con la anotación [@Entity](#) para que Room la gestione.
  - posee el atributo "tableName" para indicar cual es la tabla en la base de datos que almacena la información.
- Cada entidad debe tener al menos 1 campo definido con la anotación [@PrimaryKey](#), que representa la clave primaria.
  - posee el atributo "[autoGenerate\(\)](#)" que admite un valor boolean si deseamos que SQLite genere un autonumérico para la clave

## Entidades

- Cada entidad debe tener o un constructor sin argumentos, o un constructor con todos los atributos que no son publicos o no tienen métodos setters.
- Las claves primarias deben anotarse con `android.support.annotation.NonNull` (`@NonNull`)
- Si un atributo es usado solo en memoria pero no debe sincronizarse con la base de datos marcarlo como [@Ignore](#).
- La anotación "`@ColumnInfo`" permite definir para Room cual será el nombre de la columna de la base de datos asociada a un atributo de la clase.

# Entidades

```
@Entity(tableName = "APP_PROYECTO")
public class Proyecto {
    @PrimaryKey
    @NonNull
    @ColumnInfo(name = "ID_PROYECTO")
    private String id;

    public Proyecto(){
        this.id = UUID.randomUUID().toString();
    }

    private String nombre;
    private Double presupuesto;
    private Integer horas;
```

## DAO

- Al marcar una clase con `@Entity`, le indicamos a Room que deberá gestionar la persistencia de las instancias de dicha clase. (SELECT – INSERT – DELETE – UPDATE )
- Para personalizar los métodos de estas operaciones debemos crear una interface marcada como `@Dao`, donde indicaremos a Room cuales son los métodos de gestión de datos que ofrece.
- Internamente, en tiempo de ejecución, Room creará una clase que implemente automáticamente todas estas acciones.
- Cada objeto Dao gestionará la persistencia de una entidad.

# DAO

- Deberemos definir un método y anotarlo con `@Insert` para darle la semántica de crear un nuevo dato en una única transacción.
- El método creará un insert para el único parámetro, que debe ser un puntero a un objeto de tipo `@Entity`, o una lista de los mismos.
- Tiene un atributo que permite determinar que hacer en caso de conflicto (los ID coinciden)

int	<a href="#"><u>ABORT</u></a> OnConflict strategy constant to abort the transaction.
int	<a href="#"><u>FAIL</u></a> OnConflict strategy constant to fail the transaction.
int	<a href="#"><u>IGNORE</u></a> OnConflict strategy constant to ignore the conflict.
int	<a href="#"><u>REPLACE</u></a> to replace the old data and continue the transaction.
int	<a href="#"><u>ROLLBACK</u></a> OnConflict strategy constant to rollback the transaction.

## DAO - Insert

- Ejemplo de métodos anotados con insert

```
@Dao
public interface MyDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    public void insertUsers(User... users);

    @Insert
    public void insertBothUsers(User user1, User user2);

    @Insert
    public void insertUsersAndFriends(User user, List<User> friends);
}
```



## Update y Delete

- Del mismo modo que podemos anotar un método con `@Insert`, lo podemos anotar con `@Update` o `@Delete`
- La semántica es similar, al argumento de tipo entidad que reciba como parámetro le aplicará una actualización o borrado, basado en el campo definido como clave primaria.
- El método marcado como `@Update` permite también definir que hacer en caso de conflicto

```
@Dao
public interface MyDao {
    @Update
    public void updateUser(User... users);
    @Delete
    public void deleteUser(User... users);
}
```

## Consultas

- La anotación [@Query](#) es la principal anotación para consulta.
- Se verifica en tiempo de compilación por lo que si el SQL es incorrecto no podemos generar el APK.
- Esta anotación recibe como argumento una sentencia SELECT

```
@Dao
public interface MyDao {
    @Query("SELECT * FROM user")
    public User[] loadAllUsers();
}
```

## Consultas

- La consulta puede parametrizarse
  - En la consulta definir un atributo con el prefijo ":"
  - El nombre de dicho atributo debe ser igual al nombre del argumento del método.

```
@Dao
public interface MyDao {
    @Query("SELECT * FROM user WHERE age > :minAge")
    public User[] loadAllUsersOlderThan(int minAge);
}
```

## Consultas

- Podemos retornar un subconjunto de columnas, si definimos una clase que tenga los atributos que la consulta retornará mapeados con la anotación `@ColumnInfo`

```
public class NameTuple {  
    @ColumnInfo(name="first_name")  
    public String firstName;  
  
    @ColumnInfo(name="last_name")  
    public String lastName;  
}
```

```
@Dao  
public interface MyDao {  
    @Query("SELECT first_name, last_name FROM user")  
    public List<NameTuple> loadFullName();  
}
```

## Consultas con multiples argumentos

- Algunas consultas pueden requerir un numero variable de parámetros (generalmente las que son de la forma IN(...))
- Room traduce automáticamente cuando un parámetro representa una colección y automáticamente expande el numero de parámetros pasados.

```
@Dao
public interface MyDao {
    @Query("SELECT first_name, last_name FROM user WHERE region IN (:regions)")
    public List<NameTuple> loadUsersFromRegions(List<String> regions);
}
```

## Consultas que realizan JOIN

- EN las consultas es posible usar joins

```
@Dao
public interface MyDao {
    @Query("SELECT * FROM book "
        + "INNER JOIN loan ON loan.book_id = book.id "
        + "INNER JOIN user ON user.id = loan.user_id "
        + "WHERE user.name LIKE :userName")
    public List<Book> findBooksBorrowedByNameSync(String
        userName);
}
```

## Definir el gestor de bases de datos

- Para operar con el Objeto DAO, necesitamos obtener una implementación del mismo.
- La misma la genera Room. Por lo tanto debemos invocar a un método que realice todas las tareas necesarias para la generación. Para ello creamos una clase "abstract" que extiende de RoomDatabase y luego mediante un método de clase solicitamos que se nos retorne una instancia implementada por Room de dicha clase

# Entidad

```
1. @Entity(tableName = "APP_PROYECTO")
2. public class Proyecto {
3.     @PrimaryKey
4.     @NonNull
5.     @ColumnInfo(name = "ID_PROYECTO")
6.     private String id;
7.
8.     public Proyecto(){
9.         this.id = UUID.randomUUID().toString();
10.    }

11. private String nombre;
12. private Double presupuesto;
13. private Integer horas;

14. //getters y setters
```



# Dao

@Dao

```
public interface ProyectoDao {  
    @Insert void crearProyecto(Proyecto pry);  
    @Insert void crearProyectos(List<Proyecto> pryList);  
    @Update void actualizar (Proyecto pry);  
    @Delete void deleteProyecto(Proyecto pry);  
  
    @Query("SELECT * FROM APP_PROYECTO WHERE ID_PROYECTO = :pryId")  
    Proyecto buscarPorId(String pryId);  
  
    @Query("SELECT * FROM APP_PROYECTO")  
    List<Proyecto> buscarTodos();  
}
```

```
@Database(entities = {Proyecto.class}, version = 3)  
public abstract class MyDatabase extends RoomDatabase {  
    public abstract ProyectoDao proyectoDao();  
}
```

# Repositorio (Singleton recomendado)

```
public class ProyectoRepository {  
  
    private static ProyectoRepository _REPO= null;  
    private ProyectoDao proyectoDao;  
  
    private ProyectoRepository(Context ctx){  
        MyDatabase db = Room.databaseBuilder(ctx,  
        MyDatabase.class, "dam-pry-2018").fallbackToDestructiveMigration()  
        .build();  
        proyectoDao = db.proyectoDao();  
    }  
  
    public static ProyectoRepository getInstance(Context ctx){  
        if(_REPO==null) _REPO = new ProyectoRepository(ctx);  
        return _REPO;  
    }  
}
```

## Repositorio (continuación métodos de dao)

```
public void crearProyecto(Proyecto p){  
    proyectoDao.crearProyecto(p);  
}
```

```
public void actualizarProyecto(Proyecto p){  
    proyectoDao.actualizar(p);  
}
```

```
public List<Proyecto> getAll(){  
    return proyectoDao.buscarTodos();  
}
```

```
public Proyecto buscarPorId(String id){  
    return proyectoDao.buscarPorId(id);  
}  
}
```

## Relaciones entre Entidades

- Sqlite es una base de datos relacional por lo que es posible establecer de que manera las claves foráneas son exportadas en los mapeos de zoom.
- A diferencias de mapeadores objetos relacional mas conocidos como Hibernate, en el caso de las relaciones, Room no permite que una entidad referencie a otra entidad, solo a la columna que representa su clave foránea.
- Para ello define la anotación `@ForeignKey`

## Definir claves foráneas

```
@Entity(foreignKeys = @ForeignKey(entity = User.class,  
    parentColumns = "id",  
    childColumns = "user_id"))  
  
public class Book {  
    @PrimaryKey  
    public int bookId;  
  
    public String title;  
  
    @ColumnInfo(name = "user_id")  
    public int userId;  
}
```