



UNIVERSIDAD  
DE MÁLAGA



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA  
GRADO EN INGENIERÍA INFORMÁTICA

**Pruebas de rendimiento en operaciones de inserción  
sobre bases de datos SQLite en aplicaciones móviles  
Ionic**

**Realizado por:**












Enrique Ramírez García

# Índice

<b>1. Introducción</b>	<b>3</b>
<b>2. Vista Home</b>	<b>4</b>
<b>3. Vista Pruebas</b>	<b>5</b>
3.1. Método para crear base de datos . . . . .	7
3.2. Métodos para crear tabla con/sin autoincrement . . . . .	7
3.3. Método añadir datos tx . . . . .	8
3.4. Método añadir datos tx 2 . . . . .	9
3.5. Método añadir datos batch . . . . .	10
3.6. Método para mostrar prueba . . . . .	11
3.7. Método para borrar tabla . . . . .	12
<b>4. Pruebas de rendimiento</b>	<b>13</b>

# 1. Introducción

Para estas pruebas, seguiremos una serie de pasos que detallaremos a continuación. En primer lugar, para la realización de estas pruebas se ha utilizado la API online de **jsonplaceholder** [1]. En concreto, usaremos la tabla de datos photos [3], que nos brinda unos 5000 datos de base. Una vez hemos definido la API con la que trabajaremos, pasaremos a definir la estructura básica de la aplicación.

 home	19/04/2021 10:00	Carpeta de archivos	
 images	26/04/2021 10:44	Carpeta de archivos	
 pruebas	03/05/2021 12:08	Carpeta de archivos	
 app.component.html	12/04/2021 17:55	Chrome HTML Doc...	1 KB
 app.component.scss	12/04/2021 17:55	SCSS Style Sheet	0 KB
 app.component.spec.ts	12/04/2021 17:55	Archivo TS	1 KB
 app.component.ts	12/04/2021 17:55	Archivo TS	1 KB
 app.module.ts	19/04/2021 10:34	Archivo TS	1 KB
 app-routing.module.ts	03/05/2021 12:08	Archivo TS	1 KB
 database.service.spec.ts	19/04/2021 19:34	Archivo TS	1 KB
 database.service.ts	03/05/2021 11:44	Archivo TS	8 KB

Como vemos, hemos creado 3 carpetas para estas pruebas: La carpeta home, creada por defecto, que será el nexa de la aplicación y nos permitirá ir a la pagina con las pruebas; la carpeta images, que está pensada para ser usada en conjuncion con la clase albums de la API, pero que no usaremos en estas pruebas para hacerlas más simples; y la carpeta pruebas, donde crearemos la página con las pruebas que vamos a realizar. Además, tenemos el archivo **database.service**, que nos proporcionará el servicio de bases de datos que usaremos en la aplicación. Si echamos un vistazo a la vista de la aplicación, nada más abrirla, nos encontraremos en la página home.

## 2. Vista Home

La vista de la página home está definida como vemos a continuación:



Como podemos observar, esta página consta de varios botones, algunos que reutilizaremos en la página **pruebas**, pero el que nos interesa es el orientado en la esquina superior derecha de color verde, ya que los otros botones están orientados a funcionar con la tabla albums y, como hemos mencionado anteriormente, nos centraremos en la tabla photos.

### 3. Vista Pruebas

La vista de la página pruebas está definida como vemos a continuación:



Tras mostrar la vista, enumeraremos las funciones de los botones que vemos en ella:

- **Crear base de datos:** Con este botón creamos la base de datos. Esto podría evitarse haciendo que se inicie cada vez que cargue la aplicación, pero al estar haciendo pruebas, nos es más útil poder crearla cuando queramos.

- **Crear tabla con/sin autoincrement:** El parámetro *autoincrement* [2] se usa en columnas de tipo *INTEGER* para hacer que su rowid se incremente de forma automática al añadir un nuevo dato a esa columna que no haya 2 datos con el mismo rowid. Sin embargo, esto provoca un gasto extra de recursos y, por tanto, empeora el rendimiento. Por ello, se recomienda usarlo solo si es necesario. En nuestro caso, vamos a probar a insertar los datos en una tabla con y sin *autoincrement* para ver esta diferencia de rendimiento.
- **Añadir datos tx y tx 2:** Con estos botones añadiremos fotos a la base de datos de dos formas distintas mediante transacciones. Analizaremos el rendimiento de cada forma y sus pros y contras.
- **Añadir datos batch:** Con este botón añadimos, al igual que con los dos anteriores, fotos a la base de datos pero usando el método batch en vez de transacciones para ver que método es más eficiente.
- **Mostrar prueba:** Mediante este botón mostramos los datos insertados en la base de datos con los métodos anteriores. Debido al volumen de datos con el que vamos a trabajar ya que solo vamos a usarlo como comprobación, se mostrarán los 10 últimos datos (su id y la foto).
- **Borrar tabla images:** Con este botón borramos la tabla con todo su contenido.

Tras haber visto por encima los métodos que utilizaremos, mostraremos ahora su correspondiente funcionamiento interno. Para ello, haremos uso de la clase *database.service*, donde definiremos la lógica de estos métodos:

### 3.1. Método para crear base de datos

```
async getData() {
  this.http.get(this.url).subscribe(res => {
    this.items = res;
  });
}

async crearDatabasePrueba() {
  this.getData();
  await this.sqlite.create({
    name: "db",
    location: "default",
  }).then((db: SQLiteObject) => {
    this.storage = db;
  }).catch((e) => {
    alert("error on creating database " + JSON.stringify(e));
  });
}
```

Como podemos ver, con el método *getData()* hacemos una llamada a la url con la API Rest mencionada anteriormente. Tras esto, mediante el plugin de capacitor de sqlite, abrimos la base de datos.

### 3.2. Métodos para crear tabla con/sin autoincrement

```
async crearImagesSin() {
  await this.storage.transaction(function (tx) {
    console.time();
    tx.executeSql(
      'CREATE TABLE IF NOT EXISTS images (albumId INTEGER, id INTEGER PRIMARY KEY, title VARCHAR(255), url VARCHAR(255), thumbnailUrl VARCHAR(255))',
      []
    );
    console.timeEnd();
  }).catch(e => console.log(e));
}

async crearImagesCon() {
  await this.storage.transaction(function (tx) {
    console.time();
    tx.executeSql(
      'CREATE TABLE IF NOT EXISTS images (albumId INTEGER, id INTEGER PRIMARY KEY AUTOINCREMENT, title VARCHAR(255), url VARCHAR(255), thumbnailUrl VARCHAR(255))',
      []
    );
    console.timeEnd();
  }).catch(e => console.log(e));
}
```

Estos dos métodos son prácticamente iguales, a excepción del parámetro *autoincrement* que añadimos en el segundo método. En ellos llamamos a la

base de datos sqlite y creamos la tabla Images en la que almacenaremos los parámetros albumId, id (clave primaria), title, url (la foto), thumbnailUrl (miniatura de la foto).

### 3.3. Método añadir datos tx

```
async aniadirDatos() {
  let datos = this.items;
  const loading = await this.loadingCtrl.create({ message: 'Insertando datos', spinner: "circles" });
  loading.present();
  var t0 = performance.now();
  await this.storage.transaction(function (tx) {
    for (var cont = 0; cont < 30; cont++) {
      for (var i = 0; i < datos.length; i++) {
        tx.executeSql('INSERT INTO images (albumId, id, title, url, thumbnailUrl) VALUES (?, ?, ?, ?, ?)',
          [datos[i].albumId, datos[i].id + (cont * datos.length), datos[i].title, datos[i].url, datos[i].thumbnailUrl]);
      }
    }
    loading.dismiss();
    var t1 = performance.now();
    var time = t1 - t0;
    console.log("Tiempo total acumulado: " + time + " ms");
  }).then()
  .catch(e => console.log(e));
}
```

Con este método, insertamos en la base de datos todos los datos que pasemos. Utilizaremos dos bucles for, el primero para obtener los 5000 datos de la tabla images con las fotos, llamémoslo *slot*, y el segundo para añadir un slot adicional para cada iteración. Como el id debe ser único, lo modificamos con la operación que mostramos. En este caso, con 30 iteraciones por 5000 datos en cada una, tenemos un total de 150.000 datos.



### 3.4. Método añadir datos tx 2

```
async aniadirDatosAmpliado(lim) {
  let datos = this.items;
  let timer = 0;
  const loading = await this.loadingCtrl.create({ message: 'Insertando stack ' + (lim+1) + ' de datos', spinner: "circles" });
  loading.present();
  var t0 = performance.now();
  await this.storage.transaction(function (tx) {
    for (var i = 0; i < datos.length; i++) {
      tx.executeSql('INSERT INTO images (albumId, id, title, url, thumbnailUrl) VALUES (?, ?, ?, ?, ?)',
        [datos[i].albumId, datos[i].id + (lim * datos.length), datos[i].title, datos[i].url, datos[i].thumbnailUrl]);
    }
  });
  loading.dismiss();
  var t1 = performance.now();
  var time2 = t1 - t0;
  timer += time2;
}).then().catch(e => console.log(e));
this.con = timer;
}

async aniadirDatosTransaccionAmpl(){
  let c = 0;
  for(var i=0;i<30;i++){
    await this.db.aniadirDatosAmpliado(i);
    c += this.db.con;
  }
  console.log("Tiempo total acumulado: " + c + " ms");
}
```

En este caso, aunque la función es la misma que en el método anterior, llamamos al segundo bucle for en el método de la propia pagina de pruebas (segunda captura) en vez de llamarlo en el del servicio, haciendo la llamada al método del servicio dentro de este bucle. Para contar el tiempo, usaremos la variable c para ir añadiendo cada iteración dicho tiempo.

### 3.5. Método añadir datos batch

```
async aniadirDatosBatch(lim) {
  let datos = this.items;
  let sql = [];
  let timer = 0;
  const loading = await this.loadingCtrl.create({ message: 'Insertando stack ' + (lim+1) + ' de datos', spinner: "circles" });
  loading.present();
  var t0 = performance.now();
  {
    datos.forEach(item => {
      sql.push(["INSERT INTO images (albumId, id, title, url, thumbnailUrl) VALUES (?, ?, ?, ?, ?)",
        [item.albumId, item.id + (lim * datos.length), item.title, item.url, item.thumbnailUrl]]);
    });
  }
  await this.storage.sqlBatch(sql).then((res) => {
    loading.dismiss();
    var t1 = performance.now();
    var time = t1 - t0;
    timer+=time;
  }).catch(e => console.log(e));
  this.con = timer;
}

async aniadirDatosBatch(){
  let c = 0;
  for (var cont = 0; cont < 30; cont++){
    await this.db.aniadirDatosBatch(cont);
    c += this.db.con;
  }
  console.log("Datos insertados con batch");
  console.log("Tiempo total acumulado: " + c + " ms");
  this.db.con = 0;
}
```

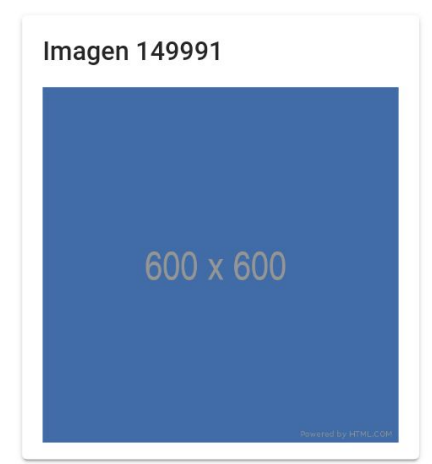
Aunque la función sigue siendo la misma que en los metodos de adición anteriores, en este caso, al usar un batch, operamos de una forma algo distinta. Recorremos los datos con un `forEach` y después los introducimos en un array vacío para, finalmente, llamar al método `batch` sobre el array que hemos rellenado previamente. El resto del método es igual que el anterior sobre transacciones.

### 3.6. Método para mostrar prueba

```
async mostrarDatosPrueba() {
  return this.storage.executeSql(`SELECT * FROM images WHERE images.id > 149990`, []).then(res => {
    return res;
  }).catch(e => {
    return "error on getting albums " + JSON.stringify(e);
  })
}

async mostrarDatos(){
  await this.db.mostrarDatosPrueba().then((data) => {
    this.photos = [];
    for (var i = 0; i < data.rows.length; i++) {
      this.photos.push(data.rows.item(i));
    }
  });
}
```

Para hacer una pequeña comprobación más visual de que los datos han sido insertados correctamente, mediante la sentencia que vemos en la primera captura, hacemos que se filtren esos datos y se muestren en pantalla de la siguiente forma:



donde se muestra la imagen con su id encima. Como dijimos anteriormente, al solo usarlo como comprobante, imprimirems únicamente 10 datos (como podemos ver en la sentencia select). Al haber 150.000 datos, cogeremos los 10 últimos, es decir, desde el id 149.991 al 150.000.

### 3.7. Método para borrar tabla

```
async borrarImages() {
  await this.storage.transaction(function (tx) {
    tx.executeSql(
      `DROP TABLE images`,
      []
    );
    console.log("Tabla Images borrada");
  }).catch(e => console.log(e));
}

borrarTabla(){
  this.db.borrarImages();
  this.photos = [];
}
```

Para borrar una tabla con sus datos correspondientes, usamos la sentencia *drop table* con dicha tabla. Una vez hecho esto, en la pagina de pruebas ponemos el array **photos** vacío para quitar los datos de la pantalla en caso de que estuvieran mostrándose (segunda captura).

## 4. Pruebas de rendimiento

A continuación, pasaremos a realizar las pruebas de rendimientos sobre los métodos anteriormente mostrados. Usaremos la herramienta **devTools** de chrome para ello. Procedemos primero a crear la base de datos, para después crear la tabla sin autoincrement e insertar los datos con el primer tipo de transacción para luego, compararlo con la tabla creada con autoincrement. Estos son los resultados:

Tabla creada <u>con autoincrement</u>	<a href="#">pruebas.page.ts:25</a>
default: 0.06201171875 ms	<a href="#">database.service.ts:93</a>
Tiempo total acumulado: 380.6999999942491 ms	<a href="#">database.service.ts:131</a>
Datos insertados con transacciones	<a href="#">pruebas.page.ts:35</a>
Tabla Images borrada	<a href="#">database.service.ts:213</a>
Tabla creada <u>sin autoincrement</u>	<a href="#">pruebas.page.ts:30</a>
default: 0.029052734375 ms	<a href="#">database.service.ts:82</a>
Tiempo total acumulado: 215.59999999590218 ms	<a href="#">database.service.ts:131</a>
Datos insertados con transacciones	<a href="#">pruebas.page.ts:35</a>

Como vemos, el tiempo total es mayor cuando usamos el autoincrement para el primer tipo de transacciones. A continuación, veamos los resultados usando el segundo tipo de transacciones:

Tabla creada <u>con autoincrement</u>	<a href="#">pruebas.page.ts:25</a>
default: 0.037841796875 ms	<a href="#">database.service.ts:93</a>
Datos insertados con transacciones 2	<a href="#">pruebas.page.ts:46</a>
Tiempo total acumulado: 566.2000000011176 ms	<a href="#">pruebas.page.ts:47</a>
Tabla Images borrada	<a href="#">database.service.ts:213</a>
Tabla creada <u>sin autoincrement</u>	<a href="#">pruebas.page.ts:30</a>
default: 0.05517578125 ms	<a href="#">database.service.ts:82</a>
Datos insertados con transacciones 2	<a href="#">pruebas.page.ts:46</a>
Tiempo total acumulado: 561.5000000252621 ms	<a href="#">pruebas.page.ts:47</a>

Como vemos, el tiempo sigue siendo superior en la tabla con autoincrement, aunque en este caso la diferencia es menor. En cuanto a la comparación entre ambos tipos de transacciones, vemos que este segundo tiene un tiempo mayor que el primer tipo de transacción en ambos casos. Pasemos, por último, a ver los resultados con la operación batch:

Tabla creada con <u>autoincrement</u>	pruebas.page.ts:25
default: 0.037109375 ms	database.service.ts:93
Elementos insertados con batch	database.service.ts:168
Tiempo total acumulado: 15338.499999998137 ms	database.service.ts:171
Tabla Images borrada	database.service.ts:213
Tabla creada <u>sin autoincrement</u>	pruebas.page.ts:30
default: 0.0830078125 ms	database.service.ts:82
Elementos insertados con batch	database.service.ts:168
Tiempo total acumulado: 14780.100000003353 ms	database.service.ts:171

Podemos observar que se sigue cumpliendo que los tiempos son mayores en las tablas con autoincrement, si bien, la diferencia con los tiempos de las transacciones es mucho mayor en este caso.

Por tanto, podríamos concluir que el método más rápido para insertar grandes conjuntos de datos en una base de datos SQLite es el primer tipo de transacción, pero ocurre algo cuando aumentamos el número de datos al propuesto en este caso. Si probamos a aumentar este número (recordemos, teníamos 150.000 datos) a, por ejemplo, 250000; veamos que es lo que ocurre:

OPEN database: db	capacitor-runtime.js:2813
OPEN database: db - OK	capacitor-runtime.js:2817
Tabla Images borrada	database.service.ts:213
Tabla creada con autoincrement	pruebas.page.ts:25
default: 0.06201171875 ms	database.service.ts:93
Tiempo total acumulado: 435.7000000018161 ms	database.service.ts:131
Error: Java exception was raised during method invocation at capacitorExec (capacitor-runtime.js:2001) at Object.execProxy [as exec] (capacitor-runtime.js:2111) at SQLitePluginTransaction.run (capacitor-runtime.js:3108) at SQLitePluginTransaction.start (capacitor-runtime.js:2970) at capacitor-runtime.js:2783 at ZoneDelegate.invokeTask (zone-evergreen.js:399) at Object.onInvokeTask (core.js:28540) at ZoneDelegate.invokeTask (zone-evergreen.js:398) at Zone.runTask (zone-evergreen.js:167) at invokeTask (zone-evergreen.js:480)	database.service.ts:133
Datos insertados con transacciones	pruebas.page.ts:35
Tabla Images borrada	database.service.ts:213
Tabla creada sin autoincrement	pruebas.page.ts:30
default: 0.5380859375 ms	database.service.ts:82
Tiempo total acumulado: 472.3999999987427 ms	database.service.ts:131
Error: Java exception was raised during method invocation at capacitorExec (capacitor-runtime.js:2001) at Object.execProxy [as exec] (capacitor-runtime.js:2111) at SQLitePluginTransaction.run (capacitor-runtime.js:3108) at SQLitePluginTransaction.start (capacitor-runtime.js:2970) at capacitor-runtime.js:2783 at ZoneDelegate.invokeTask (zone-evergreen.js:399) at Object.onInvokeTask (core.js:28540) at ZoneDelegate.invokeTask (zone-evergreen.js:398) at Zone.runTask (zone-evergreen.js:167) at invokeTask (zone-evergreen.js:480)	database.service.ts:133
Datos insertados con transacciones	pruebas.page.ts:35

Podemos comprobar que, pese a que parece que puede ejecutarlo, en última instancia se produce el error que hemos señalado, produciéndose un *rollback* y quedando anulada la transacción. Si comprobamos que ocurre tanto con el otro tipo de transacción que tenemos como con el batch, observamos lo siguiente:

OPEN database: db	capacitor-runtime.js:2813
OPEN database: db - OK	capacitor-runtime.js:2817
Tabla creada <u>con autoincrement</u>	pruebas.page.ts:25
default: 0.322998046875 ms	database.service.ts:93
Datos insertados con transacciones 2	pruebas.page.ts:46
Tiempo total acumulado: 991.00000067288 ms	pruebas.page.ts:47
Tabla Images borrada	database.service.ts:213
Tabla creada <u>sin autoincrement</u>	pruebas.page.ts:30
default: 0.05224609375 ms	database.service.ts:82
Datos insertados con transacciones 2	pruebas.page.ts:46
Tiempo total acumulado: 890.1999999652617 ms	pruebas.page.ts:47

OPEN database: db	capacitor-runtime.js:2813
OPEN database: db - OK	capacitor-runtime.js:2817
Tabla creada <u>con autoincrement</u>	pruebas.page.ts:25
default: 0.172119140625 ms	database.service.ts:94
Datos insertados con batch	pruebas.page.ts:62
Tiempo total acumulado: 30230.19999996177 ms	pruebas.page.ts:63
Tabla Images borrada	database.service.ts:236
Tabla creada <u>sin autoincrement</u>	pruebas.page.ts:30
default: 0.028076171875 ms	database.service.ts:83
Datos insertados con batch	pruebas.page.ts:62
Tiempo total acumulado: 32141.600000148173 ms	pruebas.page.ts:63

Como vemos, en este caso si se ejecuta correctamente la inserción (a pesar de que, evidentemente, el tiempo es mayor por la mayor cantidad de datos). Por lo tanto, para volúmenes de datos muy grandes, no es recomendable usar este primer método. También cabe recalcar que en este primer método existe un cierto *delay* tras acabar la operación de inserción (cuando acaba la pantalla de carga) y el mensaje por consola que nos informa de la finalización de dicho proceso.

Por ello, concluimos que lo mejor es usar tanto el método batch como, sobre todo, el segundo método de transacciones antes que este primero. Si tenemos que elegir entre el batch o el segundo tipo de transacción, vemos que la transacción es bastante más rápida. También conviene recordar que los datos

utilizados para esta prueba son datos ficticios, por lo que muy probablemente, al usar datos reales con más parámetros, los tiempos en general sean más elevados. Adicionalmente, dejaremos un enlace a github [4] para descargar el proyecto completo y que así, pueda editarlo y/o probarlo, además de otros enlaces que hablan un poco más de este tema [5,6].

## Referencias

- [1] <https://jsonplaceholder.typicode.com/>
- [2] <https://www.sqlite.org/autoinc.html>
- [3] <https://jsonplaceholder.typicode.com/photos>
- [4] <https://github.com/quiquesk8/pruebasInsercionSQLite>
- [5] <https://sqlite.org/speed.html>
- [6] <https://www.sqlite.org/faq.html#q19>