ESCUELA DE SISTEMAS Y TECNOLOGÍAS

Transparencias de **ANALISTA DE SISTEMAS**
Edición 2023 - Materia: **Aplicaciones Móviles con Flutter**

TEMA: Persistencia

Agenda

- Shared Preferences
- Bases de Datos Locales
- Sintaxis Básica de SQLite
- El Paquete sqflite
  - Abrir o Crear la Base de Datos
  - Insertar Registros
  - Actualizar Registros
  - Eliminar Registros
  - Obtener Registros
  - Métodos "raw"
- Comprobar la BD en el Emulador

Shared Preferences (1)

- El paquete **shared_preferences** de *Flutter* facilita el almacenamiento persistente y la recuperación de datos en la aplicación.
- Se utiliza principalmente para guardar pequeñas cantidades de datos que deben persistir entre diferentes sesiones de la aplicación, como configuraciones de usuario, preferencias, ajustes, etc.
- La característica principal de *shared_preferences* es que utiliza el sistema de almacenamiento compartido del sistema operativo en el que se ejecuta la aplicación, lo que permite que los datos persistan incluso después de que la aplicación se cierre y se vuelva a abrir.

Shared Preferences (2)

- Los tipos de datos soportados que se pueden almacenar son: números enteros (*int*), números con decimales (*double*), booleanos (*bool*), cadenas de texto (*String*) y listas de cadenas de texto (*List<String>*).
- Si se necesita manejar grandes cantidades de datos o datos más complejos, puede ser más apropiado utilizar otras soluciones de almacenamiento como bases de datos *SQLite* o manejo de archivos.

Shared Preferences (3)

- Para utilizar este paquete, primero debe agregarse a la sección de dependencias del archivo *pubspec.yaml*:

  dependencies:
  ...
  shared_preferences: ^2.2.1

- Luego de lo cual podrá ser importado en el archivo de código *Dart* donde se vaya a utilizar:

  import 'package:shared_preferences/shared_preferences.dart';

Shared Preferences (4)

- Las operaciones para guardar y leer información se realizan mediante un objeto de tipo *SharedPreferences*.
- Este objeto se puede obtener con el método estático *getInstance()* de la clase **SharedPreferences**.
- El método *getInstance()* es asincrónico, por lo que si debe esperarse a que termine su ejecución se invoca con *await* (el método anfitrión tendrá que ser *async*).
- Luego se podrán guardar datos mediante los diferentes métodos *setInt(...)*, *setDouble(...)*, *setBool(...)*, *setString(...)*, y *setStringList(...)* que reciben un parámetro con el nombre de la preferencia, y otro con el valor a guardar.

Shared Preferences (5)

- Para leer los datos se utilizan los métodos *getInt(...)*, *getDouble(...)*, *getBool(...)*, *getString(...)* y *getStringList(...)* que devuelven el valor de la preferencia cuyo nombre se pasa por parámetro, o *null* si ésta no existe.
- Para eliminar un dato específico se utiliza el método *remove(...)* al que se le pasa el nombre de la preferencia a eliminar.
- Para eliminar todos los datos guardados por la aplicación mediante *Shared Preferences* se utiliza el método *clear()*.

Bases de Datos Locales (1)

- Las bases de datos locales desempeñan un papel fundamental en el desarrollo de aplicaciones *Flutter*, ya que permiten a los desarrolladores almacenar y administrar datos de manera eficiente en el dispositivo del usuario.
- Estas bases de datos son esenciales para aplicaciones que requieren almacenamiento de datos fuera de línea, como aplicaciones de lista de tareas, aplicaciones de notas, aplicaciones de mensajería, etc.
- En *Flutter*, hay varias opciones disponibles para implementar bases de datos locales, pero una de las más comunes y populares es *SQLite*.

Bases de Datos Locales (2)

- **SQLite** es un motor de base de datos relacional ligero y autónomo que se integra bien con *Flutter* y se utiliza para crear bases de datos locales en dispositivos móviles.
- *SQLite* es conocido por su rendimiento eficiente y su bajo consumo de recursos, lo que lo convierte en una excelente opción para aplicaciones móviles con necesidades de almacenamiento de datos locales.
- También ofrece mecanismos de seguridad como la encriptación de bases de datos, lo que permite proteger datos sensibles almacenados en la aplicación.

Bases de Datos Locales (3)

- *Flutter* ofrece una variedad de paquetes y bibliotecas para trabajar con bases de datos locales, como por ejemplo el paquete *sqflite*, que simplifican la interacción con *SQLite* y ofrecen herramientas útiles para administrar bases de datos locales.
- Dado que las operaciones de bases de datos pueden ser intensivas en términos de recursos, es importante realizar estas operaciones en un hilo separado para evitar bloquear la interfaz de usuario.
- Afortunadamente, *Flutter* proporciona herramientas para trabajar con operaciones de bases de datos asincrónicas y manejar eventos de manera eficiente.

Sintaxis Básica de SQLite (1)

- **SQLite** es un sistema de gestión de bases de datos relacionales que comparte muchas similitudes con *MySQL* en términos de sintaxis *SQL* básica, pero también tiene algunas diferencias.
- Para crear tablas en *SQLite*, se puede utilizar la declaración *CREATE TABLE*, similar a *MySQL*.
- *SQLite* admite los tipos de datos comunes *INTEGER*, *TEXT* (no se especifica longitud), *REAL* y *BLOB*.
- La clave primaria de una tabla se define utilizando la palabra clave *PRIMARY KEY*, como en *MySQL*.
- Se puede utilizar *AUTOINCREMENT* para crear columnas con valores autoincrementales, similar a *AUTO_INCREMENT* en *MySQL*.

Sintaxis Básica de SQLite (2)

- Se puede crear índices utilizando *CREATE INDEX*, al igual que en *MySQL*.
- Las consultas *SELECT* en *SQLite* son muy similares a *MySQL*. Se puede utilizar *SELECT*, *FROM*, *WHERE*, *GROUP BY*, *ORDER BY*, y otras cláusulas para recuperar datos.
- *SQLite* admite funciones agregadas como *SUM*, *AVG*, *COUNT*, *MAX*, y *MIN* para realizar cálculos en conjuntos de datos.
- *SQLite* admite *INNER JOIN*, *LEFT JOIN*, *RIGHT JOIN*, *FULL JOIN*, y *CROSS JOIN*, como *MySQL*.
- Se puede utilizar subconsultas dentro de otras consultas, similar a *MySQL*.

Sintaxis Básica de SQLite (3)

- *SQLite* tiene sus propias funciones para manipulación de cadenas y fechas.
- Para insertar datos en una tabla, se utiliza la sentencia *INSERT INTO*, como en *MySQL*.
- La actualización de datos se realiza mediante *UPDATE*, similar a *MySQL*.
- Se puede eliminar registros utilizando *DELETE FROM*, al igual que *MySQL*.
- *SQLite* admite transacciones utilizando *BEGIN*, *COMMIT*, y *ROLLBACK* para garantizar la integridad de los datos.
- Se puede agregar comentarios en las consultas *SQL* utilizando -- o /* */, como en *MySQL*.

Sintaxis Básica de SQLite (4)

- *SQLite* se centra en ser una base de datos ligera y de una sola conexión, por lo que carece de muchas de las características avanzadas que se encuentran en sistemas de gestión de bases de datos más grandes.
- No permite la creación de procedimientos almacenados (*stored procedures*), funciones definidas por el usuario (*stored functions*) o gatillos (*triggers*) como parte de su diseño.

El Paquete sqflite (1)

- El paquete **sqflite** es una biblioteca que facilita la interacción con bases de datos *SQLite* en aplicaciones *Flutter*.
- Simplifica la creación y administración de bases de datos *SQLite* en *Flutter*. Proporciona una interfaz simple y amigable para ejecutar consultas *SQL* y gestionar registros de bases de datos.
- Permite a las aplicaciones *Flutter* almacenar datos de manera persistente en el dispositivo del usuario, lo que significa que los datos permanecen disponibles incluso después de que la aplicación se cierre o el dispositivo se reinicie.

El Paquete sqflite (2)

- Para utilizar este paquete, primero debe agregarse a la sección de dependencias del archivo *pubspec.yaml*:

  dependencies:
  ...
  sqflite: ^2.3.0

- Luego de lo cual podrá ser importado en el archivo de código *Dart* donde se vaya a utilizar:

  import 'package:sqflite/sqflite.dart';

Abrir o Crear la Base de Datos (1)

- El primer paso para trabajar con una base de datos mediante *sqflite* es abrirla o crearla.
- El método **openDatabase(...)** se utiliza para abrir o crear (en caso de que no exista) una base de datos SQLite. Devuelve un *Future<Database>*.
- Parámetros comunes de *openDatabase(...)*:
  - **path**: *String* con la ruta absoluta o relativa en el sistema de archivos del dispositivo donde se buscará o almacenará la base de datos *SQLite*. Se puede utilizar el método *getDatabasesPath()* que devuelve un *Future<String>* para obtener la ruta del directorio de bases de datos de la aplicación.

Abrir o Crear la Base de Datos (2)

- Parámetros comunes de *openDatabase(...)* (cont.):
  - **version**: Entero que especifica la versión de la base de datos. Si la base de datos no existe o la versión especificada es diferente de la versión existente, se llamará a *onCreate* u *onUpgrade* respectivamente para configurar la base de datos. Es importante actualizar la versión de la base de datos cuando se realizan cambios en su estructura.
  - **onCreate**: *Function(Database, int)?* que se llama cuando se crea una nueva base de datos o cuando la versión de la base de datos cambia y no se encuentra una versión anterior. En ésta se define la estructura de la base de datos con sus tablas.

Abrir o Crear la Base de Datos (3)

- Parámetros comunes de *openDatabase(...)* (cont.):
  - **onUpgrade**: *Function(Database, int, int)?* que se llama cuando la versión de la base de datos existente es inferior a la especificada en el parámetro *version*. Se utiliza para realizar actualizaciones en la estructura para que la base de datos sea compatible con la nueva versión.
  - **onDowngrade**: Similar a *onUpgrade*, pero se llama cuando la versión de la base de datos existente es mayor que la especificada en el parámetro *version*. Se utiliza para manejar la degradación de la versión de la base de datos.

Abrir o Crear la Base de Datos (4)

- Parámetros comunes de *openDatabase(...)* (cont.):
  - **onOpen**: *Function(Database)?* que se llama cuando se abre la base de datos con éxito. Útil para realizar tareas adicionales de inicialización si es necesario.
  - **readOnly**: Indicador *bool* que determina si la base de datos se abrirá en modo de sólo lectura. Si se establece en *true*, no se podrán realizar operaciones de escritura en la base de datos.
  - **inMemory**: Indicador *bool* que especifica si la base de datos debe crearse en la memoria en lugar de en el almacenamiento persistente. Útil para bases de datos temporales.

Abrir o Crear la Base de Datos (5)

- Parámetros comunes de *openDatabase(...)* (cont.):
  - **singleInstance**: Indicador *bool* que controla si debe utilizarse una única instancia de la base de datos durante la vida de la aplicación. Esto puede ayudar a optimizar el rendimiento.
- Al finalizar, se recomienda cerrar la base de datos a través del objeto *Database* retornado por *openDatabase(...)* mediante el método *close()*.

Insertar Registros (1)

- El método **insert(...)** del objeto *Database* se utiliza para insertar un nuevo registro en una tabla de una base de datos *SQLite*.
- Parámetros comunes de *insert(...)*:
  - **table**: *String* que especifica la tabla en la que se desea insertar el registro.
  - **values**: *Map<String, Object?>* que contiene las columnas y sus valores correspondientes a insertar en la tabla. Las claves del mapa deben ser los nombres de las columnas de la tabla y los valores asociados son los datos que se desea insertar en esas columnas. Los valores deben ser de cualquier tipo compatible con *SQLite*.

Insertar Registros (2)

- Parámetros comunes de *insert(...)* (cont.):
  - **nullColumnHack**: String opcional que se utiliza para especificar el nombre de una columna a la que se le pasará un valor nulo si el mapa de valores está vacío. Esto puede ser útil cuando se desea insertar un registro únicamente con los valores predeterminados de las columnas, ya que la sentencia *INSERT* a generar debe incluir al menos una columna.

Insertar Registros (3)

- Parámetros comunes de *insert(...)* (cont.):
  - **conflictAlgorithm**: Parámetro opcional que determina cómo se debe manejar un conflicto si se intenta insertar un registro que tiene el mismo valor en una columna única que ya existe en la tabla. Los valores posibles son (*ConflictAlgorithm*): *.rollback*: Revertirá la transacción si hay un conflicto; *.abort*: Cancelará la inserción si hay un conflicto; *.fail*: Fallará la inserción si hay un conflicto; *.ignore*: Ignorará la inserción si hay un conflicto; *.replace*: Reemplazará el registro existente por el nuevo registro en caso de conflicto.

Insertar Registros (4)

- El método *insert(...)* devuelve un valor entero que representa el *ID* del nuevo registro insertado, o *-1* en caso de error. El *ID* suele ser una clave primaria autoincremental si se ha configurado así en la tabla.

Actualizar Registros (1)

- El método **update(...)** del objeto *Database* se utiliza para actualizar registros existentes en una tabla de una base de datos SQLite.
- Parámetros comunes de *update(...)*:
  - **table**: *String* que especifica la tabla en la que se desea actualizar registros.
  - **values**: *Map<String, Object?>* que contiene las columnas y sus valores correspondientes a actualizar en la tabla. Las claves del mapa deben ser los nombres de las columnas de la tabla y los valores asociados son los datos que se desea actualizar en esas columnas. Los valores deben ser de cualquier tipo compatible con *SQLite*.

Actualizar Registros (2)

- Parámetros comunes de *update(...)* (cont.):
  - **where**: *String* opcional que se utiliza para especificar una cláusula *WHERE* que determina qué registros se deben actualizar. Se pueden utilizar marcadores de posición (*?*) y proporcionar los valores correspondientes en *whereArgs*.
  - **whereArgs**: *List<Object?>* opcional que se utiliza junto con *where* para proporcionar los valores de marcadores de posición que se sustituirán en la cláusula *WHERE*.

Actualizar Registros (3)

- Parámetros comunes de *update(...)* (cont.):
  - **conflictAlgorithm**: Parámetro opcional que determina cómo se debe manejar un conflicto si se intenta realizar una actualización que produce un conflicto en una columna única. Los valores posibles son los mismos que se mencionaron para el método *insert(...)*.
- El método *update(...)* devuelve un valor entero que representa el número de registros que se han actualizado con éxito. Si no se actualiza ningún registro o se produce un error, el valor devuelto será *0*.

Eliminar Registros (1)

- El método **delete(...)** del objeto *Database* se utiliza para eliminar registros de una tabla en una base de datos SQLite.
- Parámetros comunes de *delete(...)*:
  - **table**: *String* que especifica la tabla de la que se desea eliminar registros.
  - **where**: *String* opcional que se utiliza para especificar una cláusula *WHERE* que determina qué registros se deben eliminar. Se pueden utilizar marcadores de posición (*?*) y proporcionar los valores correspondientes en *whereArgs*.

Eliminar Registros (2)

- Parámetros comunes de *delete(...)* (cont.):
  - **whereArgs**: *List<Object?>* opcional que se utiliza junto con *where* para proporcionar los valores de marcadores de posición que se sustituirán en la cláusula *WHERE*.
- El método *delete(...)* devuelve un valor entero que representa el número de registros que se han eliminado con éxito. Si no se elimina ningún registro o se produce un error, el valor devuelto será *0*.

Obtener Registros (1)

- El método **query(...)** del objeto *Database* se utiliza para realizar consultas *SELECT* en una base de datos *SQLite*. Permite recuperar datos de una o varias tablas de la base de datos.
- Parámetros comunes de *query(...)*:
  - **table**: *String* que especifica la(s) tabla(s) de la(s) que se desea recuperar datos. Se puede incluir múltiples tablas separadas por comas si se desea realizar una consulta *JOIN*.
  - **distinct**: Indicador *bool* que determina si la consulta debe devolver resultados distintos (sin duplicados) o no. Establecerlo en *true* elimina las filas duplicadas de los resultados.

Obtener Registros (2)

- Parámetros comunes de *query(...)* (cont.):
  - **columns**: *List<String>* que especifica las columnas que se desea recuperar de la tabla. Si no se especifica, se seleccionarán todas las columnas.
  - **where**: *String* opcional que se utiliza para especificar una cláusula *WHERE* que determina qué registros se deben seleccionar. Se pueden utilizar marcadores de posición (*?*) y proporcionar los valores correspondientes en *whereArgs*.
  - **whereArgs**: *List<Object?>* opcional que se utiliza junto con *where* para proporcionar los valores de marcadores de posición que se sustituirán en la cláusula *WHERE*.

Obtener Registros (3)

- Parámetros comunes de *query(...)* (cont.):
  - **groupBy**: *String* con una cláusula *GROUP BY* que agrupa los resultados por una o varias columnas. Esto se utiliza en combinación con funciones de agregación como *SUM*, *COUNT*, etc.
  - **having**: *String* con una cláusula *HAVING* que permite filtrar los resultados de una consulta *GROUP BY*. Funciona de manera similar a *WHERE*, pero se aplica después del agrupamiento.

Obtener Registros (4)

- Parámetros comunes de *query(...)* (cont.):
  - **orderBy**: *String* con una cláusula *ORDER BY* que determina el orden en el que se devolverán los resultados. Se puede especificar una o varias columnas y el orden (*ASC* para ascendente, *DESC* para descendente).
  - **limit**: Valor *int* que limita la cantidad máxima de filas que se devolverán en los resultados. Es útil para la paginación o para limitar la cantidad de datos recuperados.

Obtener Registros (5)

- Parámetros comunes de *query(...)* (cont.):
  - **offset**: Valor *int* que especifica el número de filas que se deben omitir antes de comenzar a devolver resultados. Se utiliza en combinación con *limit* para la paginación.
- El método *query(...)* devuelve una lista de mapas *List<Map<String, Object?>>*, donde cada mapa representa una fila de resultados de la consulta. Los nombres de las columnas son las claves del mapa y los valores son los datos de esas columnas para cada fila.

Métodos "raw" (1)

- Además de los métodos vistos anteriormente, el paquete *sqflite* proporciona métodos "*raw*" como *rawInsert(...)*, *rawUpdate(...)*, *rawDelete(...)*, y *rawQuery(...)*, que permiten ejecutar consultas *SQL* personalizadas en una base de datos *SQLite*.
- A diferencia de los métodos *insert(...)*, *update(...)*, *delete(...)*, y *query(...)*, que proporcionan una interfaz más orientada a objetos, los métodos "*raw*" dan más flexibilidad para ejecutar consultas *SQL* directas.
- Al utilizar métodos "*raw*", se escriben manualmente las consultas *SQL*, lo que proporciona un alto grado de control pero también requiere cuidado para evitar problemas de seguridad como la inyección de *SQL*.

Métodos "raw" (2)

- Se recomienda validar y escapar adecuadamente los valores que serán incorporados en las consultas *SQL* para prevenir ataques de seguridad.
- Cada uno de estos métodos "*raw*" recibe por parámetro la consulta *SQL* a ejecutar, en la que se pueden utilizar marcadores de posición (*?*), y también un *List<Objetct?>* para proporcionar los valores de dichos marcadores de posición.
- Los valores de retorno de los métodos "*raw*" son análogos a los de los métodos *insert(...)*, *update(...)*, *delete(...)* y *query(...)*.

Comprobar la BD en el Emulador (1)

- Para comprobar la creación de la base de datos en el emulador *Android* se puede utilizar desde la línea de comandos la herramienta **adb** (**Android Debug Bridge**), incluida en el paquete de herramientas de la plataforma (*platform-tools*) del *SDK* de *Android*:
  - **adb devices**: Muestra los dispositivos conectados, incluyendo su identificador serial.
  - **adb shell**: Abre una terminal en el único dispositivo conectado. Para salir, ingresar *exit*.
  - **adb -s <IDENTIFICADOR> shell**: Abre una terminal en el dispositivo correspondiente al número de serie indicado.

Comprobar la BD en el Emulador (2)

- Una vez que se ha ingresado a la terminal del dispositivo se puede localizar la base de datos en el directorio */data/data/<PAQUETE>/databases/*
- Se puede conectar con la base de datos mediante el comando **sqlite3**, indicando el archivo de la misma.
- La herramienta *sqlite3* permite enviar consultas a la base de datos, y dispone de algunos comandos útiles:
  - **.tables**: Muestra las tablas de la base de datos.
  - **.schema**: Muestra el esquema de las tablas. Se puede indicar el nombre de una tabla concreta para mostrar únicamente el esquema de ésta.
  - **.exit** o **.quit**: Salir y volver a la terminal del dispositivo.