Instituto Politécnico Nacional

Escuela Superior de Cómputo

Desarrollo de Aplicaciones Móviles Nativas

“Ejemplo BD”

Alumno:

Malagón Baeza Alan Adrian

Profesor:
M. en C. José Asunción Enríquez Zárate

Grupo: 7CM1

Introducción

El presente reporte tiene como objetivo presentar los temas y tecnologías abordados en el desarrollo de una aplicación de gestión de eventos. A lo largo de esta experiencia, se han explorado diversos aspectos relacionados con el desarrollo de software y la implementación de funcionalidades clave.

En primer lugar, se ha profundizado en el desarrollo de aplicaciones móviles utilizando Kotlin, un lenguaje de programación moderno y altamente compatible con el entorno de Android. Con la ayuda del entorno de desarrollo integrado Android Studio, se ha logrado crear una aplicación nativa de Android con una interfaz de usuario interactiva y fluida.

En cuanto a la arquitectura de la aplicación, se ha utilizado el patrón Model-View-Controller (MVC), que permite una estructura organizada y modular del proyecto. Esta separación de responsabilidades ha facilitado el mantenimiento y la escalabilidad del código a medida que se han agregado nuevas funcionalidades.

Para el almacenamiento de datos, se ha empleado una base de datos SQLite utilizando la biblioteca Room. Room simplifica la gestión de la base de datos y proporciona una capa de abstracción sobre SQLite, permitiendo realizar operaciones eficientes como inserción, actualización y eliminación de registros.

Además, se ha hecho uso del patrón de diseño ViewHolder y la clase RecyclerView para la visualización eficiente de listas en la interfaz de usuario. Esto ha permitido mostrar de manera dinámica y optimizada los eventos y asistentes almacenados en la base de datos, brindando una experiencia fluida al usuario.

Durante el desarrollo de la aplicación, se han abordado conceptos fundamentales de programación, como la manipulación de objetos, el manejo de eventos de interfaz de usuario, la gestión de actividades y la navegación entre pantallas. Estas habilidades han sido fundamentales para el diseño y la implementación de una aplicación completa y funcional.

En resumen, este reporte presenta los temas y tecnologías utilizados en el desarrollo de una aplicación de gestión de eventos para Android. A través de esta experiencia, se ha adquirido un conocimiento práctico valioso en el desarrollo de aplicaciones móviles, lo que contribuye al aprendizaje y dominio de las habilidades necesarias en este campo.

Conceptos

SQLite

SQLite es un sistema de gestión de base de datos relacional (RDBMS) que se destaca por ser liviano, fácil de usar y de implementar. A diferencia de otros RDBMS como MySQL o PostgreSQL, SQLite no funciona como un servidor independiente, sino que se integra directamente en la aplicación que lo utiliza.

SQLite se utiliza ampliamente en aplicaciones móviles y de escritorio para almacenar datos localmente, especialmente en entornos donde se requiere una solución de base de datos ligera y autónoma. Es una opción popular para aplicaciones que necesitan un almacenamiento de datos sencillo sin tener que utilizar un servidor de base de datos completo.

Base de datos

Una base de datos SQLite es un archivo único que contiene todas las tablas, índices, vistas y otros objetos relacionados con la estructura y los datos almacenados. Este archivo se puede copiar y transferir fácilmente entre diferentes sistemas.

Tablas

Una base de datos SQLite está formada por una o más tablas, que son estructuras de datos organizadas en filas y columnas. Cada columna tiene un nombre y un tipo de datos asociados, como texto, número o fecha. Las filas representan los registros individuales de datos.

Consultas

SQLite utiliza el lenguaje SQL (Structured Query Language) para realizar consultas y manipulación de datos. Puedes utilizar sentencias SELECT para recuperar datos de la base de datos, INSERT para insertar nuevos registros, UPDATE para modificar registros existentes y DELETE para eliminar registros.

Acceso a datos

El acceso a datos con Room y SQLite se refiere a la forma en que se interactúa y se gestiona una base de datos SQLite utilizando la biblioteca Room en el desarrollo de aplicaciones de Android.

Room es una biblioteca de persistencia desarrollada por Google que proporciona una capa de abstracción sobre SQLite y simplifica el proceso de acceso a la base de

datos desde una aplicación Android. Room combina las características de SQLite, como su rendimiento y confiabilidad, con una sintaxis más sencilla y una gestión más eficiente de la base de datos.

El uso de Room simplifica la administración de la base de datos SQLite en una aplicación Android al proporcionar una capa de abstracción más alta y una sintaxis más intuitiva. Room se encarga de muchos aspectos relacionados con SQLite, como la creación y actualización de la base de datos, la generación de consultas SQL y la administración eficiente de los datos.

Entidades

Las entidades en Room son las clases que representan las tablas en la base de datos SQLite. Cada entidad se mapea a una tabla y cada instancia de la entidad se considera un registro en esa tabla. Las entidades se definen utilizando anotaciones, como `@Entity`, y cada entidad tiene una o más propiedades que corresponden a las columnas de la tabla.

DAO (Data Access Object

Los DAO en Room son interfaces o clases abstractas que definen los métodos para acceder a la base de datos. Los métodos en los DAO pueden incluir consultas SQL personalizadas o métodos predefinidos como `insert()`, `update()`, `delete()` para operaciones de CRUD (crear, leer, actualizar, eliminar) en la base de datos. Los DAO se anotan con `@Dao` en Room.

Base de datos

La clase de base de datos en Room es una clase abstracta que representa la base de datos SQLite. Esta clase se anota con `@Database` y define la lista de entidades que componen la base de datos, así como su versión. La clase de base de datos también puede proporcionar los DAO asociados y configuraciones adicionales, como migraciones de esquema.

Migraciones

Las migraciones en Room se utilizan para actualizar el esquema de la base de datos cuando se realizan cambios en las entidades o en la estructura de la base de datos. Una migración es una clase que implementa la interfaz `Migration` y define los cambios necesarios para pasar de una versión anterior a una nueva versión de la base de datos. Las migraciones permiten preservar los datos existentes durante las actualizaciones.

Consultas y relaciones

Room ofrece un conjunto de anotaciones y características adicionales para definir consultas complejas y relaciones entre entidades. Puedes utilizar anotaciones como `@Query` para ejecutar consultas personalizadas y `@Relation` para definir relaciones entre entidades y facilitar la recuperación de datos relacionados.

Desarrollo

CRUD para Eventos

Entidad Evento

```
/**
 * Clase que representa la entidad 'Evento' en la base de datos.
 * Esta clase define la estructura y los atributos de un evento.
 *
 * @param idEvento El ID del evento (generado automáticamente).
 * @param nombreEvento El nombre del evento.
 * @param descripcionEvento La descripción del evento.
 * @param fechaEvento La fecha del evento.
 */
@Entity(tableName = "Evento")
data class Evento(
    @PrimaryKey(autoGenerate = true)
    val idEvento: Int,
    @ColumnInfo(name = "nombreEvento")
    var nombreEvento: String,
    @ColumnInfo(name = "descripcionEvento")
    var descripcionEvento: String,
    @ColumnInfo(name = "fechaEvento")
    var fechaEvento: Date
)
```

DAO Evento

```
/**
 * Objeto de Acceso a Datos (DAO, por sus siglas en inglés) para
la entidad 'Evento'.
 * Esta interfaz proporciona métodos para interactuar con la
tabla 'Evento' en la base de datos.
 */
@Dao
```

```kotlin
interface EventoDAO {
    /**
     * Inserta un nuevo registro de 'Evento' en la base de datos.
     * @param evento El objeto 'Evento' a insertar.
     */
    @Insert
    fun insert(evento: Evento)

    /**
     * Actualiza un registro existente de 'Evento' en la base de
datos.
     * @param evento El objeto 'Evento' actualizado.
     */
    @Update
    fun update(evento: Evento)

    /**
     * Elimina un registro de 'Evento' de la base de datos.
     * @param evento El objeto 'Evento' a eliminar.
     */
    @Delete
    fun delete(evento: Evento)

    /**
     * Recupera todos los registros de 'Evento' de la base de
datos.
     * @return Una lista de todos los registros de 'Evento'.
     */
    @Query("SELECT * FROM Evento")
    fun readAll(): List<Evento>

    /**
     * Recupera todos los registros de 'Evento' de la base de
datos como LiveData.
     * LiveData es una clase contenedora de datos que puede ser
observada para detectar cambios.
     * @return Un objeto LiveData que contiene una lista de todos
los registros de 'Evento'.
     */
```

```
    @Query("SELECT * FROM Evento")
    fun getAllUsersInDB(): LiveData<List<Evento>>

    /**
     * Recupera un registro de 'Evento' de la base de datos según
su ID.
     * @param idEvento El ID del registro de 'Evento' a
recuperar.
     * @return El registro de 'Evento' con el ID especificado.
     */
    @Query("SELECT * FROM Evento WHERE idEvento LIKE :idEvento")
    fun read(idEvento: Int): Evento
}
```

CRUD para Asistentes

Entidad Asistente

```
/**
 * Clase que representa la entidad 'Asistente' en la base de
datos.
 * Esta clase define la estructura y los atributos de un
asistente de un evento.
 *
 * @param idAsistente El ID del asistente (generado
automáticamente).
 * @param nombreAsistente El nombre del asistente.
 * @param paternoAsistente El apellido paterno del asistente.
 * @param maternoAsistente El apellido materno del asistente.
 * @param emailAsistente El correo electrónico del asistente.
 * @param idEvento El ID del evento al que pertenece el
asistente.
 */
@Entity(tableName = "Asistente")
data class Asistente(
    @PrimaryKey(autoGenerate = true)
    val idAsistente: Int,
```

```
    @ColumnInfo(name = "nombreAsistente")
    var nombreAsistente: String,
    @ColumnInfo(name = "paternoAsistente")
    var paternoAsistente: String,
    @ColumnInfo(name = "maternoAsistente")
    var maternoAsistente: String,
    @ColumnInfo(name = "emailAsistente")
    var emailAsistente: String,
    @ColumnInfo(name = "idEvento")
    val idEvento: Int
)
```

DAO Asistente

```
/**
 * Data Access Object (DAO) para la entidad 'Asistente'.
 * Esta interfaz proporciona métodos para interactuar con la
tabla 'Asistente' en la base de datos.
 */
@Dao
interface AsistenteDAO {
    /**
     * Inserta un nuevo 'Asistente' en la base de datos.
     * @param asistente El objeto 'Asistente' a insertar.
     */
    @Insert
    fun insert(asistente: Asistente)

    /**
     * Actualiza un 'Asistente' existente en la base de datos.
     * @param asistente El objeto 'Asistente' actualizado.
     */
    @Update
    fun update(asistente: Asistente)

    /**
     * Elimina un 'Asistente' de la base de datos.
     * @param asistente El objeto 'Asistente' a eliminar.
```

```
     */
    @Delete
    fun delete(asistente: Asistente)

    /**
     * Recupera todos los registros de 'Asistente' de la base de
datos.
     * @return Una lista de todos los registros de 'Asistente'.
     */
    @Query("SELECT * FROM Asistente")
    fun readAll(): List<Asistente>

    /**
     * Recupera todos los registros de 'Asistente' de la base de
datos como LiveData.
     * LiveData es una clase contenedora de datos que puede ser
observada para detectar cambios.
     * @return Un objeto LiveData que contiene una lista de todos
los registros de 'Asistente'.
     */
    @Query("SELECT * FROM Asistente")
    fun getAllUsersInDB(): LiveData<List<Asistente>>

    /**
     * Recupera los 'Asistentes' de un evento específico de la
base de datos.
     * @param eventoId El ID del evento.
     * @return Una lista de 'Asistentes' que pertenecen al evento
con el ID especificado.
     */
    @Query("SELECT * FROM Asistente WHERE idEvento = :eventoId")
    fun getAsistentesByEventoId(eventoId: Int): List<Asistente>

    /**
     * Recupera un registro de 'Asistente' de la base de datos
según su ID.
     * @param idAsistente El ID del registro de 'Asistente' a
recuperar.
     * @return El registro de 'Asistente' con el ID especificado.
```

```
     */
    @Query("SELECT * FROM Asistente WHERE idAsistente
LIKE :idAsistente")
    fun read(idAsistente: Int): Asistente
}
```

Base de Datos Eventos

```
/**
 * Clase abstracta que representa la base de datos principal de
la aplicación.
 * Esta clase extiende RoomDatabase y define los métodos de
acceso a los DAOs correspondientes.
 *
 * @property eventoDao El DAO para acceder a la tabla de eventos.
 * @property asistenteDao El DAO para acceder a la tabla de
asistentes.
 */
@Database(entities = [Evento::class, Asistente::class], version =
2)
@TypeConverters(Converters::class)
abstract class EventosDatabase : RoomDatabase() {
    abstract fun eventoDao(): EventoDAO
    abstract fun asistenteDao(): AsistenteDAO

    companion object {
        @Volatile
        private var INSTANCE: EventosDatabase? = null

        /**
         * Obtiene una instancia de la base de datos.
         * Si no existe una instancia previa, se crea una nueva.
         *
         * @param context El contexto de la aplicación.
         * @return La instancia de la base de datos.
         */
        fun getInstance(context: Context): EventosDatabase {
            synchronized(this) {
```

```
                var instance = INSTANCE
                if (instance == null) {
                    instance = Room.databaseBuilder(
                        context.applicationContext,
                        EventosDatabase::class.java,
                        "EventosAndroid"
                    )
                        .allowMainThreadQueries()
                        .fallbackToDestructiveMigration()
                        .build()
                }
                INSTANCE = instance
                return instance
            }
        }
    }
}
```

Presentación de la funcionalidad desarrollada

Agregar y Mostrar Eventos

```
/**
 * Clase principal de la actividad principal de la aplicación.
 * Esta clase extiende AppCompatActivity y gestiona la interfaz
de usuario y la lógica de la aplicación.
 */
class MainActivity : AppCompatActivity() {

    private lateinit var txtNombre: EditText
    private lateinit var txtDescripcion: EditText
    private lateinit var txtFecha: EditText
    private lateinit var btnCancelar: Button
    private lateinit var btnAgregar: Button

    private lateinit var recyclerViewEventos: RecyclerView
    private lateinit var eventosAdapter: EventosAdapter
```

```
    private lateinit var eventosDatabase: EventosDatabase

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Inicializar los elementos de la interfaz de usuario
        txtNombre = findViewById(R.id.txtNombre)
        txtDescripcion = findViewById(R.id.txtDescripcion)
        txtFecha = findViewById(R.id.txtFecha)
        btnCancelar = findViewById(R.id.btnCancelar)
        btnAgregar = findViewById(R.id.btnModificar)

        // Obtener una instancia de la base de datos
        eventosDatabase =
EventosDatabase.getInstance(applicationContext)

        // Configurar el botón de cancelar
        btnCancelar.setOnClickListener {
            borrarCampos()
        }

        // Configurar el botón de agregar
        btnAgregar.setOnClickListener {
            agregarEvento()
        }

        // Configurar el RecyclerView y el adaptador
        recyclerViewEventos =
findViewById(R.id.recyclerViewEventos)
        recyclerViewEventos.layoutManager =
LinearLayoutManager(this)

        eventosAdapter = EventosAdapter(this, eventosDatabase)
        recyclerViewEventos.adapter = eventosAdapter
    }

    /**
```

```
     * Borra los campos de texto en la interfaz de usuario.
     */
    private fun borrarCampos() {
        txtNombre.text.clear()
        txtDescripcion.text.clear()
        txtFecha.text.clear()
    }

    /**
     * Agrega un nuevo evento a la base de datos y actualiza el
adaptador del RecyclerView.
     */
    private fun agregarEvento() {
        // Obtener los valores ingresados en los campos de texto
        val nombre = txtNombre.text.toString()
        val descripcion = txtDescripcion.text.toString()
        val fechaString = txtFecha.text.toString()

        val dateFormat = SimpleDateFormat("yyyy-MM-dd",
Locale.getDefault())
        val fechaUtil: Date = dateFormat.parse(fechaString) as
Date
        val fecha = java.sql.Date(fechaUtil.time)

        val eventoDao = eventosDatabase.eventoDao()

        // Crear un objeto Evento con los datos ingresados
        val evento = Evento(0, nombre, descripcion, fecha)

        // Guardar el evento en la base de datos utilizando el
DAO
        eventoDao.insert(evento)

        // Actualizar el adaptador del RecyclerView con la nueva
lista de eventos
        eventosAdapter.actualizarEventos()
    }
}
```

Eliminar Evento

```
/**
 * Clase adaptador para el RecyclerView que muestra la lista de
eventos.
 * Esta clase se encarga de gestionar la vista de cada elemento
de la lista y de manejar las interacciones con los botones.
 *
 * @param context El contexto de la aplicación.
 * @param eventosDatabase La instancia de la base de datos de
eventos.
 */
class EventosAdapter(private val context: Context, private val
eventosDatabase: EventosDatabase) :
    RecyclerView.Adapter<EventosAdapter.EventoViewHolder>() {

    private val eventos: MutableList<Evento> =
eventosDatabase.eventoDao().readAll() as MutableList<Evento>

    override fun onCreateViewHolder(parent: ViewGroup, viewType:
Int): EventoViewHolder {
        val view =
LayoutInflater.from(parent.context).inflate(R.layout.evento_item,
parent, false)
        return EventoViewHolder(view)
    }

    override fun onBindViewHolder(holder: EventoViewHolder,
position: Int) {
        val evento = eventos[position]

        holder.bind(evento)

        holder.buttonEliminar.setOnClickListener {
            Toast.makeText(context, "ID: ${evento.idEvento}",
Toast.LENGTH_LONG).show()

            // Realizar la lógica para eliminar el evento de la
base de datos
            eventosDatabase.eventoDao().delete(evento)
```

```
            // Actualizar el contenido del adaptador eliminando
el evento de la lista
            eventos.removeAt(position)
            notifyItemRemoved(position)
            notifyItemRangeChanged(position, eventos.size)
            actualizarEventos()
        }

        holder.buttonModificar.setOnClickListener {
            Toast.makeText(context, "ID: ${evento.idEvento}",
Toast.LENGTH_LONG).show()

            // Abrir la actividad de modificación de eventos
            val intent = Intent(context,
EventoActivity::class.java)
            intent.putExtra("evento_id", evento.idEvento)
            context.startActivity(intent)
        }

        holder.buttonAsistentes.setOnClickListener {
            Toast.makeText(context, "ID: ${evento.idEvento}",
Toast.LENGTH_LONG).show()

            // Abrir la actividad de asistentes para el evento
            val intent = Intent(context,
AsistentesActivity::class.java)
            intent.putExtra("evento_id", evento.idEvento)
            context.startActivity(intent)
        }
    }

    override fun getItemCount(): Int {
        return eventos.size
    }

    inner class EventoViewHolder(itemView: View) :
RecyclerView.ViewHolder(itemView) {
        val textViewNombre: TextView =
```

```
itemView.findViewById(R.id.textViewNombre)
        val buttonEliminar: Button =
itemView.findViewById(R.id.buttonEliminar)
        val buttonModificar: Button =
itemView.findViewById(R.id.buttonModificar)
        val buttonAsistentes: Button =
itemView.findViewById(R.id.buttonAsistentes)

        /**
         * Vincula los datos del evento a la vista del
ViewHolder.
         */
        fun bind(evento: Evento) {
            textViewNombre.text = evento.nombreEvento
        }
    }

    /**
     * Actualiza la lista de eventos en el adaptador.
     * Esta función se utiliza después de realizar cambios en la
base de datos para reflejar los cambios en la vista.
     */
    fun actualizarEventos() {
        eventos.clear()
        eventos.addAll(eventosDatabase.eventoDao().readAll() as
MutableList<Evento>)
        notifyDataSetChanged()
    }
}
```

Modificar Evento

```
/**
 * Actividad para mostrar y modificar los detalles de un evento.
 * Permite al usuario editar el nombre, descripción y fecha del
evento.
 */
class EventoActivity : AppCompatActivity() {
```

```
    lateinit var txtNombre: EditText
    lateinit var txtDescripcion: EditText
    lateinit var txtFecha: EditText
    lateinit var btnCancelar: Button
    lateinit var btnModificar: Button
    lateinit var btnRegresar: Button

    private lateinit var eventosDatabase: EventosDatabase
    private lateinit var evento: Evento

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_evento)

        // Obtener el ID del evento de los extras del intent
        val eventoId = intent.getIntExtra("evento_id", -1)

        if (eventoId == -1) {
            finish()
        }

        eventosDatabase =
EventosDatabase.getInstance(applicationContext)
        evento = eventosDatabase.eventoDao().read(eventoId)

        txtNombre = findViewById(R.id.txtNombre)
        txtDescripcion = findViewById(R.id.txtDescripcion)
        txtFecha = findViewById(R.id.txtFecha)
        btnCancelar = findViewById(R.id.btnCancelar)
        btnModificar = findViewById(R.id.btnModificar)
        btnRegresar = findViewById(R.id.btnRegresar)

        actualizarCampos()

        btnCancelar.setOnClickListener {
            borrarCampos()
        }
```

```
        btnRegresar.setOnClickListener {
            finish()
        }

        btnModificar.setOnClickListener {
            val nombre = txtNombre.text.toString()
            val descripcion = txtDescripcion.text.toString()
            val fechaString = txtFecha.text.toString()

            val dateFormat = SimpleDateFormat("yyyy-MM-dd",
Locale.getDefault())
            val fechaUtil: Date = dateFormat.parse(fechaString)
as Date
            val fecha = java.sql.Date(fechaUtil.time)

            // Actualizar los datos del evento
            evento.nombreEvento = nombre
            evento.descripcionEvento = descripcion
            evento.fechaEvento = fecha

            // Guardar el evento actualizado en la base de datos
            eventosDatabase.eventoDao().update(evento)

            // Mostrar un mensaje de toast u realizar otras
acciones después de actualizar el evento
            Toast.makeText(this, "¡Evento actualizado!",
Toast.LENGTH_LONG).show()

            actualizarCampos()
        }
    }

    /**
     * Borra los campos de texto.
     */
    private fun borrarCampos() {
        txtNombre.text.clear()
        txtDescripcion.text.clear()
        txtFecha.text.clear()
```

```
    }

    /**
     * Actualiza los campos de texto con los datos del evento
actual.
     */
    private fun actualizarCampos() {
        txtNombre.setText(evento.nombreEvento)
        txtDescripcion.setText(evento.descripcionEvento)
        txtFecha.setText(evento.fechaEvento.toString())
    }
}
```

Agregar y Mostrar Asistentes

```
/**
 * Actividad para mostrar y agregar asistentes a un evento.
 * Permite al usuario ingresar el nombre, apellido paterno,
apellido materno y correo electrónico del asistente.
 */
class AsistentesActivity : AppCompatActivity() {

    lateinit var txtNombre: EditText
    lateinit var txtPaterno: EditText
    lateinit var txtMaterno: EditText
    lateinit var txtEmail: EditText
    lateinit var btnCancelar: Button
    lateinit var btnAgregar: Button

    private lateinit var recyclerViewAsistentes: RecyclerView
    private lateinit var asistentesAdapter: AsistentesAdapter

    private lateinit var eventosDatabase: EventosDatabase

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_asistentes)
```

```
        val idEvento: Int = intent.getIntExtra("evento_id", -1)

        if (idEvento == -1) {
            finish()
        }

        txtNombre = findViewById(R.id.txtNombre)
        txtPaterno = findViewById(R.id.txtPaterno)
        txtMaterno = findViewById(R.id.txtMaterno)
        txtEmail = findViewById(R.id.txtEmail)
        btnCancelar = findViewById(R.id.btnCancelar)
        btnAgregar = findViewById(R.id.btnModificar)

        eventosDatabase =
EventosDatabase.getInstance(applicationContext)

        btnCancelar.setOnClickListener {
            borrarCampos()
        }

        btnAgregar.setOnClickListener {
            agregarAsistente(idEvento)
        }

        recyclerViewAsistentes =
findViewById(R.id.recyclerViewAsistentes)
        recyclerViewAsistentes.layoutManager =
LinearLayoutManager(this)

        asistentesAdapter = AsistentesAdapter(this,
eventosDatabase, idEvento)
        recyclerViewAsistentes.adapter = asistentesAdapter
    }

    /**
     * Borra los campos de texto.
     */
    private fun borrarCampos() {
        txtNombre.text.clear()
```

```
        txtPaterno.text.clear()
        txtMaterno.text.clear()
        txtEmail.text.clear()
    }

    /**
     * Agrega un nuevo asistente al evento.
     */
    private fun agregarAsistente(idEvento: Int) {
        // Obtener los valores ingresados en los campos de texto
        val nombre = txtNombre.text.toString()
        val paterno = txtMaterno.text.toString()
        val materno = txtPaterno.text.toString()
        val email = txtEmail.text.toString()

        val asistenteDao = eventosDatabase.asistenteDao()

        // Crear un objeto Asistente con los datos ingresados
        val asistente = Asistente(0, nombre, paterno, materno,
email, idEvento)

        // Guardar el asistente en la base de datos utilizando el
DAO
        asistenteDao.insert(asistente)

        // Actualizar el adaptador del RecyclerView con la nueva
lista de asistentes
        asistentesAdapter.actualizarAsistentes()
    }
}
```

Eliminar Asistente

```
/**
 * Adaptador para mostrar y gestionar la lista de asistentes de
un evento.
 * Permite al usuario eliminar asistentes y abrir la actividad de
modificación de asistentes.
```

```
 */
class AsistentesAdapter(
    private val context: Context,
    private val eventosDatabase: EventosDatabase,
    private val idEvento: Int
) : RecyclerView.Adapter<AsistentesAdapter.AsistenteViewHolder>()
{

    private val asistentes: MutableList<Asistente> =

eventosDatabase.asistenteDao().getAsistentesByEventoId(idEvento)
as MutableList<Asistente>

    override fun onCreateViewHolder(parent: ViewGroup, viewType:
Int): AsistenteViewHolder {
        val view =
LayoutInflater.from(parent.context).inflate(R.layout.asistente_it
em, parent, false)
        return AsistenteViewHolder(view)
    }

    override fun onBindViewHolder(holder: AsistenteViewHolder,
position: Int) {
        val asistente = asistentes[position]

        holder.bind(asistente)

        holder.buttonEliminar.setOnClickListener {
            Toast.makeText(context, "ID: ${asistente.idEvento}",
Toast.LENGTH_LONG).show()

            // Realizar la lógica para eliminar el asistente de
la base de datos
            eventosDatabase.asistenteDao().delete(asistente)

            // Actualizar el contenido del adaptador eliminando
el asistente de la lista
            asistentes.removeAt(position)
            notifyItemRemoved(position)
```

```
            notifyItemRangeChanged(position, asistentes.size)
            actualizarAsistentes()
        }

        holder.buttonModificar.setOnClickListener {
            Toast.makeText(context, "ID:
${asistente.idAsistente}", Toast.LENGTH_LONG).show()
            val intent = Intent(context,
AsistenteActivity::class.java)
            intent.putExtra("asistente_id",
asistente.idAsistente)
            context.startActivity(intent)
        }
    }

    override fun getItemCount(): Int {
        return asistentes.size
    }

    inner class AsistenteViewHolder(itemView: View) :
RecyclerView.ViewHolder(itemView) {
        val textViewNombre: TextView =
itemView.findViewById(R.id.textViewNombre)
        val buttonEliminar: Button =
itemView.findViewById(R.id.buttonEliminar)
        val buttonModificar: Button =
itemView.findViewById(R.id.buttonModificar)

        fun bind(asistente: Asistente) {
            textViewNombre.text = "${asistente.nombreAsistente}
${asistente.paternoAsistente} ${asistente.maternoAsistente}"
        }
    }

    /**
     * Actualiza la lista de asistentes del adaptador.
     */
    fun actualizarAsistentes() {
        asistentes.clear()
```

```
asistentes.addAll(eventosDatabase.asistenteDao().getAsistentesByE
ventoId(idEvento) as MutableList<Asistente>)
        notifyDataSetChanged()
    }
}
```

Modificar Asistente

```
/**
 * Actividad para mostrar y modificar los detalles de un
asistente.
 * Permite al usuario editar los campos del asistente y guardar
los cambios en la base de datos.
 */
class AsistenteActivity : AppCompatActivity() {

    lateinit var txtNombre: EditText
    lateinit var txtPaterno: EditText
    lateinit var txtMaterno: EditText
    lateinit var txtEmail: EditText
    lateinit var btnCancelar: Button
    lateinit var btnModificar: Button
    lateinit var btnRegresar: Button

    private lateinit var eventosDatabase: EventosDatabase
    private lateinit var asistente: Asistente

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_asistente)

        // Obtener los extras del intent
        val asistenteId = intent.getIntExtra("asistente_id", -1)

        if (asistenteId == -1) finish()

        eventosDatabase =
```

```
EventosDatabase.getInstance(applicationContext)
        asistente =
eventosDatabase.asistenteDao().read(asistenteId)

        txtNombre = findViewById(R.id.txtNombre)
        txtPaterno = findViewById(R.id.txtPaterno)
        txtMaterno = findViewById(R.id.txtMaterno)
        txtEmail = findViewById(R.id.txtEmail)

        btnCancelar = findViewById(R.id.btnCancelar)
        btnModificar = findViewById(R.id.btnModificar)
        btnRegresar = findViewById(R.id.btnRegresar)

        actualizarCampos()

        btnCancelar.setOnClickListener {
            borrarCampos()
        }

        btnRegresar.setOnClickListener {
            finish()
        }

        btnModificar.setOnClickListener {
            val nombre = txtNombre.text.toString()
            val paterno = txtMaterno.text.toString()
            val materno = txtPaterno.text.toString()
            val email = txtEmail.text.toString()

            // Actualizar los datos del asistente
            asistente.nombreAsistente = nombre
            asistente.paternoAsistente = paterno
            asistente.maternoAsistente = materno
            asistente.emailAsistente = email

            // Guardar el asistente actualizado en la base de
datos
            eventosDatabase.asistenteDao().update(asistente)
```

```
            // Mostrar un mensaje toast u realizar cualquier otra
acción después de actualizar el asistente
            Toast.makeText(this, "Asistente Actualizado!",
Toast.LENGTH_LONG).show()

            actualizarCampos()
        }
    }

    private fun borrarCampos() {
        txtNombre.text.clear()
        txtPaterno.text.clear()
        txtMaterno.text.clear()
        txtEmail.text.clear()
    }

    private fun actualizarCampos() {
        txtNombre.setText(asistente.nombreAsistente)
        txtPaterno.setText(asistente.paternoAsistente)
        txtMaterno.setText(asistente.maternoAsistente)
        txtEmail.setText(asistente.emailAsistente)
    }
}
```

Resultados

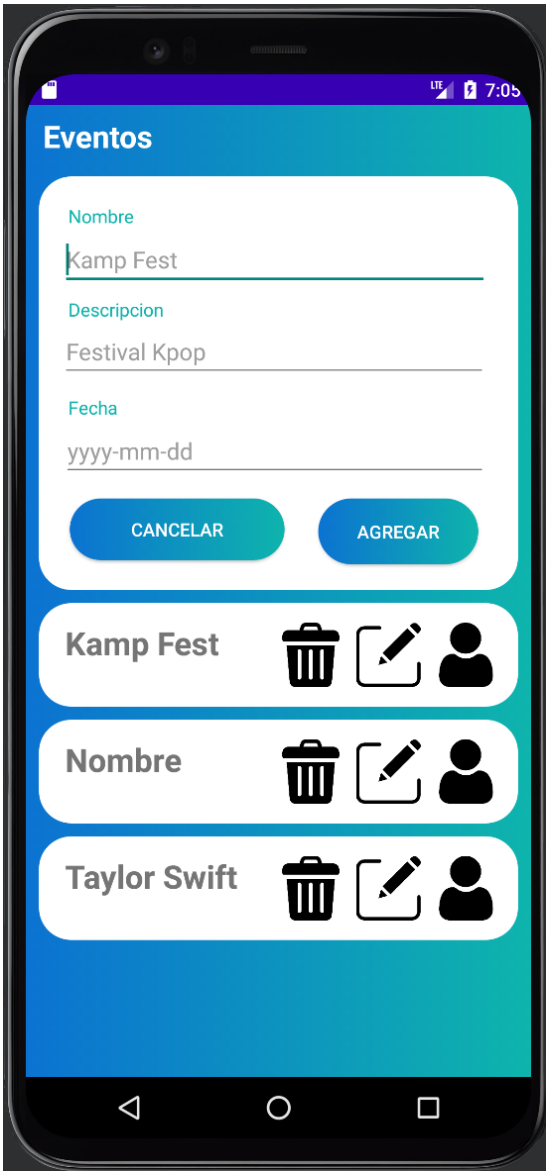

Imagen 1. Agregando Eventos

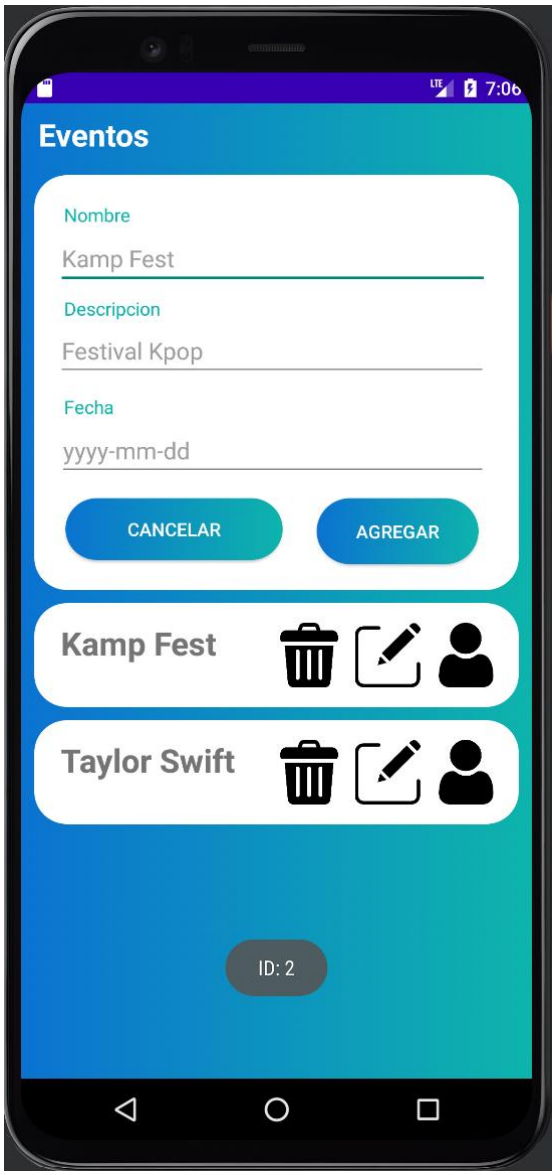

Imagen 2. Eliminando Evento

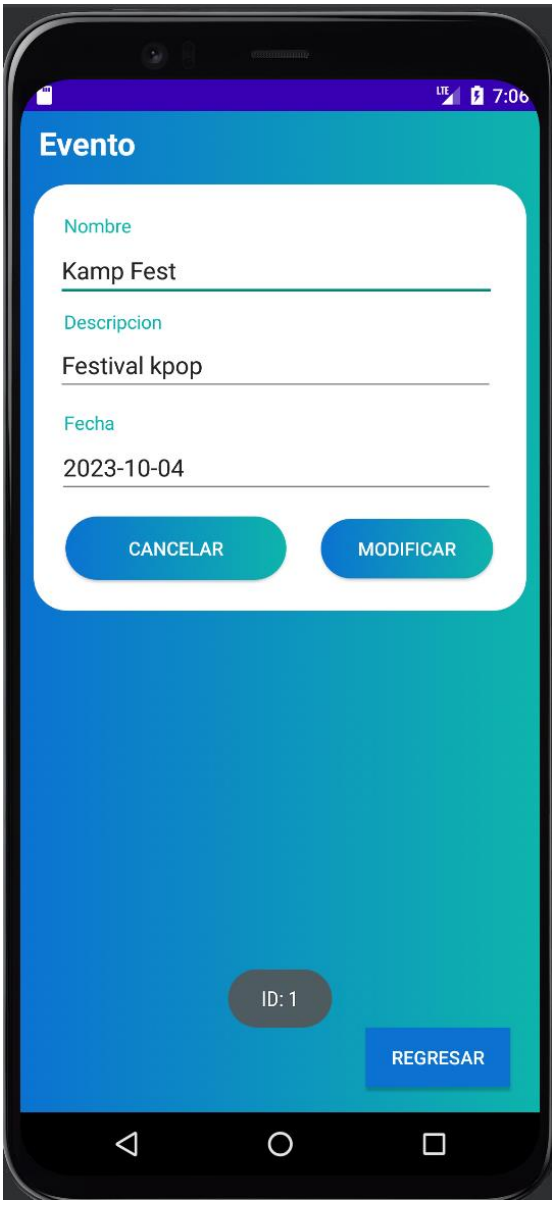

Imagen 3. Editando Evento

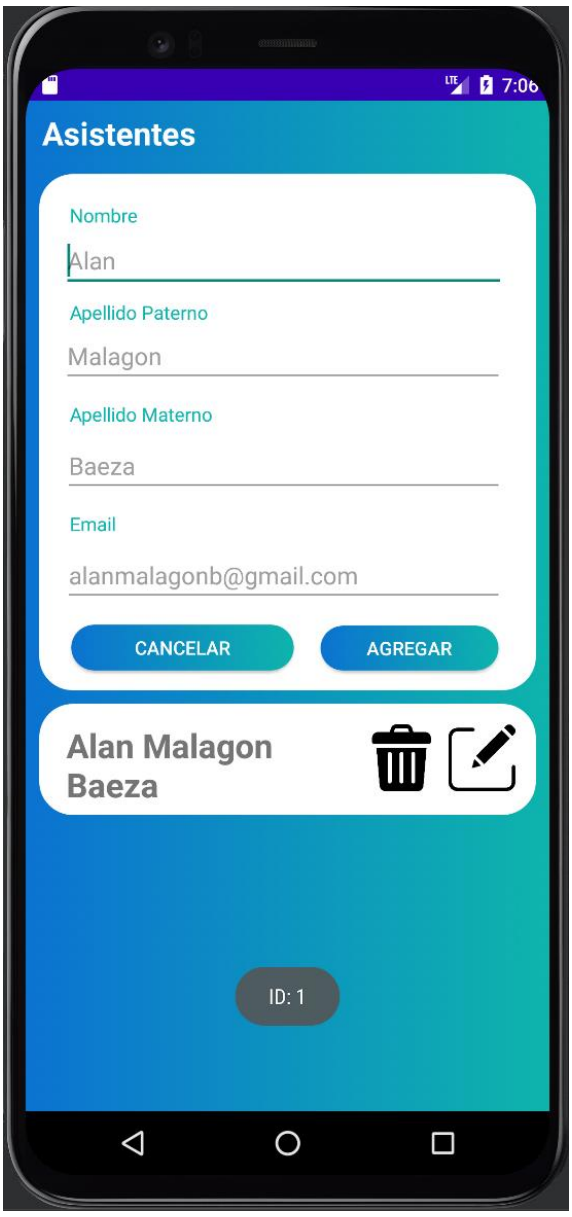

Imagen 4. Agregando Asistentes

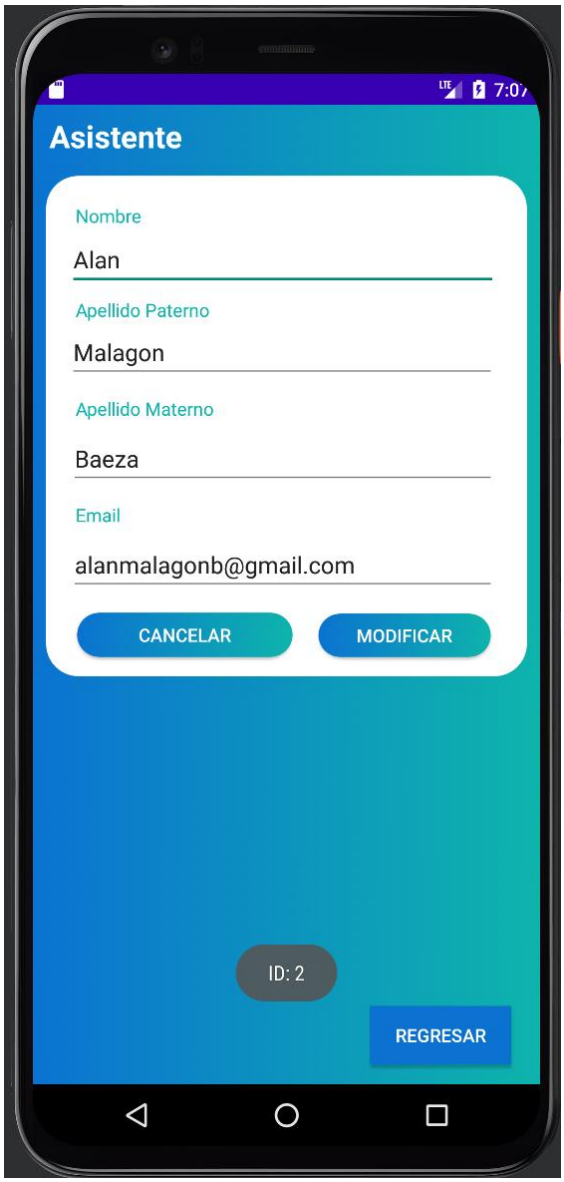

Imagen 5. Editando Asistente

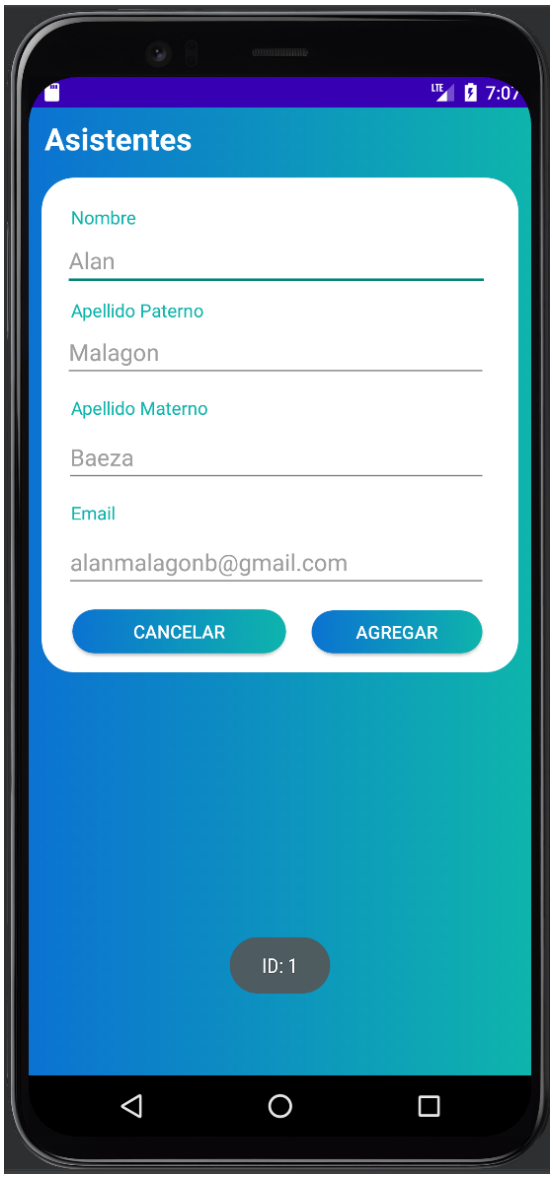

Imagen 6. Eliminando Asistente

Conclusiones

En conclusión, el desarrollo de la aplicación de gestión de eventos ha sido una experiencia enriquecedora que ha permitido adentrarse en los aspectos clave del desarrollo de software para dispositivos móviles. A lo largo de este proyecto, se ha explorado y aplicado un conjunto de tecnologías y conceptos fundamentales que son esenciales para la creación de aplicaciones nativas de Android.

Mediante el uso de Kotlin como lenguaje de programación, se ha demostrado su versatilidad y potencial para el desarrollo ágil y eficiente de aplicaciones móviles. La integración de Android Studio como entorno de desarrollo ha brindado las herramientas necesarias para diseñar una interfaz de usuario intuitiva y atractiva, así como para realizar pruebas y depuración de manera efectiva.

La implementación de la arquitectura MVC ha permitido una organización estructurada del código, facilitando su mantenimiento y extensibilidad a medida que se han añadido nuevas funcionalidades. Además, el uso de la biblioteca Room ha simplificado la gestión de la base de datos SQLite, proporcionando operaciones de almacenamiento y consulta eficientes.

La utilización del patrón ViewHolder y la clase RecyclerView ha mejorado significativamente la experiencia de usuario al optimizar la visualización de listas de eventos y asistentes, garantizando un rendimiento suave y eficiente.

A lo largo de este proyecto, se han adquirido habilidades fundamentales de programación y se ha profundizado en los conceptos clave de desarrollo de aplicaciones móviles, como la manipulación de objetos, la gestión de eventos, la navegación entre pantallas y la interacción con bases de datos.

En definitiva, este proyecto ha proporcionado una base sólida para continuar explorando y desarrollando habilidades en el ámbito de la programación móvil. La aplicación de gestión de eventos creada es un ejemplo tangible de las capacidades y conocimientos adquiridos, y representa un paso importante en la comprensión y aplicación de tecnologías actuales para crear soluciones móviles innovadoras y funcionales.

Referencias Bibliográficas

- Android Developers. (s.f.). Almacenamiento de datos en Android con Room. Recuperado de https://developer.android.com/training/data-storage/room?hl=es-419#groovy
- Android Developers. (s.f.). RecyclerView. Recuperado de https://developer.android.com/guide/topics/ui/layout/recyclerview?hl=es-419#:~:text=RecyclerView%20es%20el%20ViewGroup%20que,un%20objeto%20contenedor%20de%20vistas.
- Android Developers. (s.f.). Room. Recuperado de https://developer.android.com/jetpack/androidx/releases/room?hl=es-419.
- Mozilla. (s.f.). Modelo-Vista-Controlador (MVC). Recuperado de https://developer.mozilla.org/es/docs/Glossary/MVC.
- Android Developers. (s.f.). RecyclerView. Recuperado de https://developer.android.com/guide/topics/ui/layout/recyclerview?hl=es-419.
- Android Developers. (s.f.). Intents y filtros de intención. Recuperado de https://developer.android.com/guide/components/intents-filters?hl=es-419.
- Android Developers. (s.f.). Diseño de cuadrícula. Recuperado de https://developer.android.com/guide/topics/ui/layout/grid?hl=es-419.
- Android Developers. (s.f.). Diseño relativo. Recuperado de https://developer.android.com/guide/topics/ui/layout/relative?hl=es-419.