PROGRAMACIÓN DE DISPOSITIVOS MÓVILES

ESCUELA PLATÓN – Abril 2026 |JCMF|

Tema 3

Temas Avanzados de Android

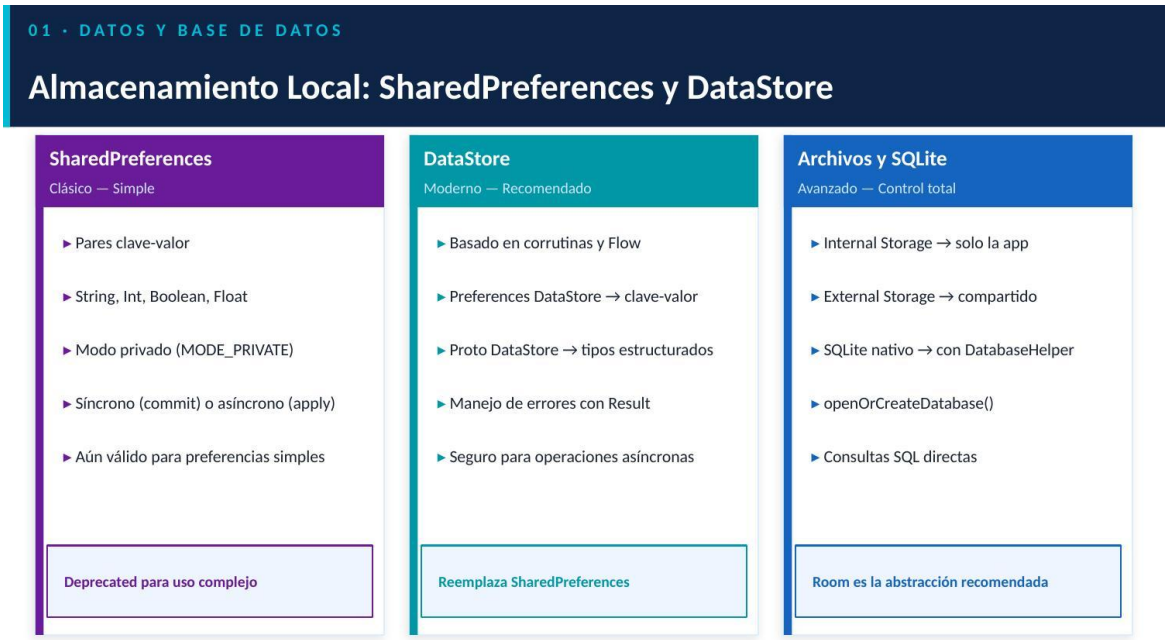
> *Estos apuntes cubren las capacidades avanzadas del desarrollo Android: almacenamiento de datos local y en la nube, integración de mapas y GPS, uso de sensores del dispositivo, reproducción y captura multimedia, creación de widgets para el escritorio, consumo de APIs REST y comunicación Bluetooth.*

Contenidos

1. Datos y Base de Datos 4
1.1. Almacenamiento Local 4
SharedPreferences 4
DataStore (Recomendado — Reemplaza SharedPreferences) 5
1.2. SQLite y Room Database 5
Entidad (Entity) 6
DAO (Data Access Object) 6
Database y Repository 6
1.3. Firebase: Base de Datos en la Nube 7
Firebase Realtime Database 7
Cloud Firestore 8
2. Mapas y GPS 9
2.1. Google Maps SDK 9
Dependencias y configuración 9
Funcionalidades principales del mapa 9
Implementación básica 10
2.2. Geolocalización 10
Geofencing 11
3. Sensores 12
3.1. Tipos de Sensores Disponibles 12
3.2. Acceso y Gestión de Sensores 12
Velocidades de muestreo (SENSOR_DELAY) 14
Casos de uso prácticos 14
4. Multimedia 15
4.1. Reproducción de Audio y Vídeo 15
MediaPlayer — Audio básico 15
ExoPlayer — Vídeo y streaming (recomendado) 16
4.2. Captura de Imagen y Vídeo con CameraX 16
Permisos de Cámara y Micrófono 17
5. Widgets 19
5.1. Creación de AppWidgets 19
Componentes necesarios 19
Widget Info XML (res/xml/widget_info.xml) 19
AppWidgetProvider 20
5.2. Mejores Prácticas de Widgets 20
6. Servicios Web 21
6.1. Retrofit: Consumo de APIs REST 21
Stack tecnológico completo 21

Configuración de Retrofit 21
Interface de la API 22
Uso en ViewModel con corrutinas 22
6.2. Autenticación con JWT 23
7. Bluetooth 24
7.1. Bluetooth Clásico vs BLE 24
Permisos necesarios (API 31+) 24
7.2. Bluetooth Low Energy (BLE) 24
Escaneo de dispositivos BLE 25
Conexión GATT y lectura de características 25
8. Resumen y Actividades 27
8.1. Puntos Clave del Tema 27
8.2. Actividades Prácticas 28
8.3. Recursos Adicionales 28

1. Datos y Base de Datos



1.1. Almacenamiento Local

Android ofrece varias opciones para almacenar datos en el dispositivo. La elección depende del volumen, estructura y privacidad requeridos:

SharedPreferences

Sistema clásico de almacenamiento de pares clave-valor para datos primitivos:

- **Tipos soportados:** String, Int, Boolean, Float, Long
- **Ideal para:** Configuraciones de usuario, preferencias, estado de sesión
- **Modos:** `MODE_PRIVATE` (solo la app) o `MODE_WORLD_READABLE` (compartido)
- **apply() vs commit():** apply() es asíncrono (recomendado), commit() es síncrono y bloqueante

```
// Guardar datos
val prefs = getSharedPreferences("mi_app_prefs", Context.MODE_PRIVATE)
prefs.edit().apply {
    putString("nombre", "Juan")
    putInt("edad", 30)
    putBoolean("sesionActiva", true)
}.apply()  // asíncrono; usar commit() para síncrono

// Leer datos
val nombre = prefs.getString("nombre", "default")
val edad   = prefs.getInt("edad", 0)

// Eliminar dato / Limpiar todo
prefs.edit().remove("nombre").apply()
prefs.edit().clear().apply()
```

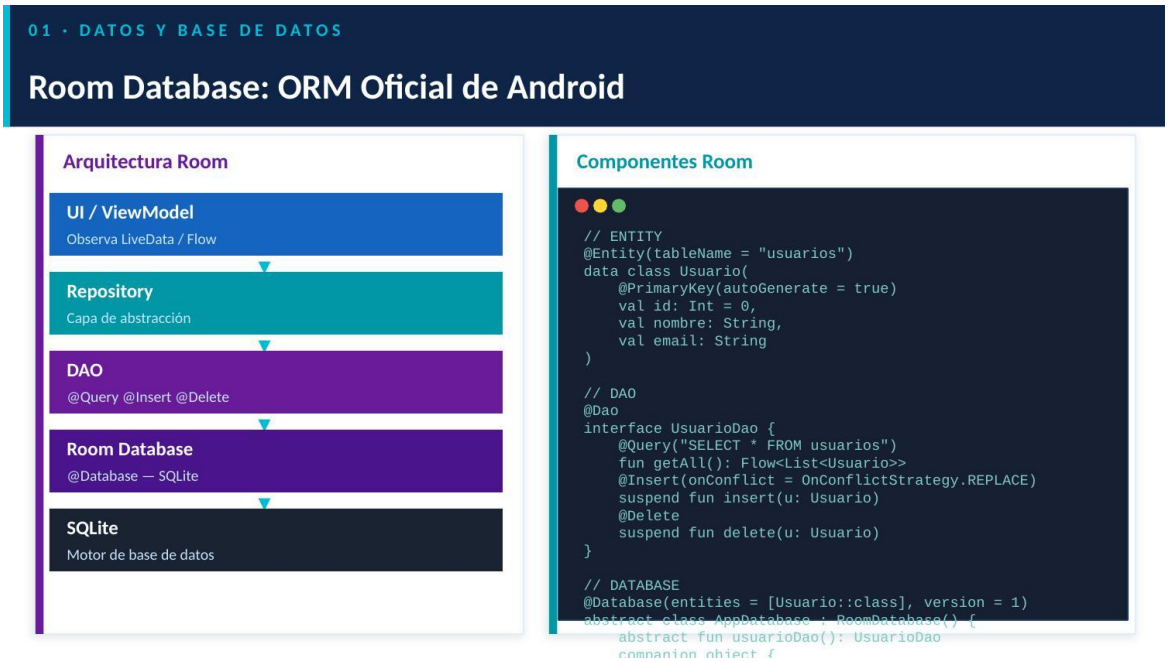
DataStore (Recomendado — Reemplaza SharedPreferences)

DataStore es la solución moderna de Jetpack para almacenamiento de pares clave-valor. Usa corrutinas y Flow para operaciones seguras y asíncronas:

- **Preferences DataStore:** Equivalente mejorado de SharedPreferences
- **Proto DataStore:** Almacena objetos tipados con Protocol Buffers
- **Ventajas:** Manejo de errores, operaciones no bloqueantes, soporte de Flow reactivo

```
// Definir claves
val NAME_KEY = stringPreferencesKey("nombre")
val AGE_KEY  = intPreferencesKey("edad")

// Crear DataStore (extensión en Context)
val Context.dataStore: DataStore<Preferences>
    by preferencesDataStore(name = "settings")

// Guardar (dentro de una corrutina)
lifecycleScope.launch {
    dataStore.edit { prefs ->
        prefs[NAME_KEY] = "Juan"
        prefs[AGE_KEY]  = 30
    }
}

// Leer como Flow
val nombreFlow: Flow<String> = dataStore.data
    .map { prefs -> prefs[NAME_KEY] ?: "" }
```

1.2. SQLite y Room Database



Room es el ORM (Object Relational Mapper) oficial de Android que actúa como capa de abstracción sobre SQLite. Simplifica drásticamente el acceso a base de datos con anotaciones y verificación en tiempo de compilación.

Componente	Descripción	Anotación principal
Entity	Clase que representa una tabla de la BD	@Entity(tableName)
DAO	Interface con métodos de acceso a datos	@Dao
Database	Clase abstracta que gestiona la BD	@Database(entities, version)
Repository	Capa que abstrae las fuentes de datos	Patrón de diseño (no Room)

Entidad (Entity)

```
@Entity(tableName = "usuarios")
data class Usuario(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    @ColumnInfo(name = "nombre") val nombre: String,
    @ColumnInfo(name = "email")  val email: String,
    @ColumnInfo(name = "edad")   val edad: Int
)
```

DAO (Data Access Object)

```
@Dao
interface UsuarioDao {
    @Query("SELECT * FROM usuarios ORDER BY nombre")
    fun getAll(): Flow<List<Usuario>>

    @Query("SELECT * FROM usuarios WHERE id = :id")
    suspend fun getById(id: Int): Usuario?

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(usuario: Usuario)

    @Update
    suspend fun update(usuario: Usuario)

    @Delete
    suspend fun delete(usuario: Usuario)
}
```

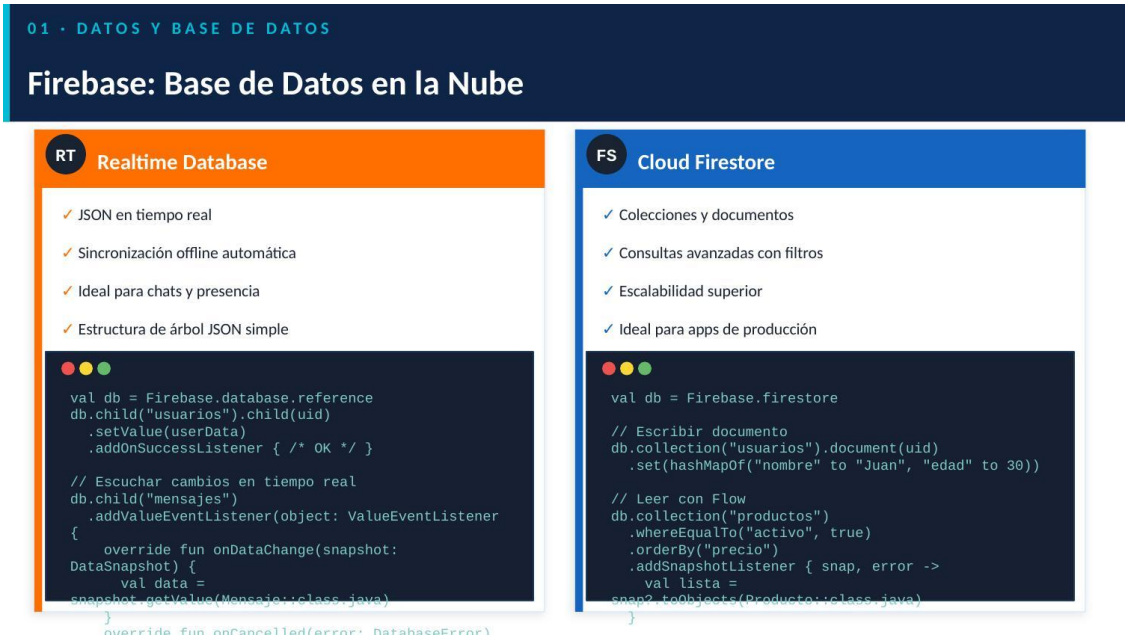
Database y Repository

```
// Database (Singleton)
@Database(entities = [Usuario::class], version = 1, exportSchema = false)
abstract class AppDatabase : RoomDatabase() {
    abstract fun usuarioDao(): UsuarioDao
    companion object {
        @Volatile private var INSTANCE: AppDatabase? = null
        fun getInstance(context: Context): AppDatabase =
            INSTANCE ?: synchronized(this) {
                Room.databaseBuilder(context.applicationContext,
                    AppDatabase::class.java, "app_db")
```

```
                    .fallbackToDestructiveMigration()
                    .build().also { INSTANCE = it }
            }
    }
}

// Repository
class UsuarioRepository(private val dao: UsuarioDao) {
    val todos: Flow<List<Usuario>> = dao.getAll()
    suspend fun insertar(u: Usuario) = dao.insert(u)
    suspend fun eliminar(u: Usuario) = dao.delete(u)
}
```

> 💡 *Patrón recomendado: ViewModel → Repository → DAO → Room. La UI solo observa StateFlow/LiveData del ViewModel. Nunca acceder a la base de datos desde la UI directamente.*

1.3. Firebase: Base de Datos en la Nube



Firebase Realtime Database

Base de datos NoSQL alojada en la nube que sincroniza datos en tiempo real entre todos los clientes conectados:

- Estructura en árbol JSON
- Sincronización automática cuando se recupera la conexión (offline support)
- **Ideal para:** chats, indicadores de presencia, datos colaborativos en tiempo real

```
val db = Firebase.database.reference

// Escribir datos
db.child("usuarios").child(uid).setValue(userData)
```

```
  .addOnSuccessListener { /* Guardado correctamente */ }
  .addOnFailureListener { e -> Log.e("DB", e.message ?: "") }

// Leer datos una vez
db.child("usuarios").child(uid).get()
  .addOnSuccessListener { snapshot ->
      val user = snapshot.getValue(Usuario::class.java)
  }

// Escuchar cambios en tiempo real
db.child("mensajes").addValueEventListener(object : ValueEventListener {
    override fun onDataChange(snapshot: DataSnapshot) {
        val lista = snapshot.children.mapNotNull {
            it.getValue(Mensaje::class.java)
        }
    }
    override fun onCancelled(error: DatabaseError) {}
})
```

Cloud Firestore

Base de datos documental escalable con consultas más potentes que Realtime Database:

- **Estructura:** Colecciones → Documentos → Campos (similar a MongoDB)
- **Consultas:** Filtros, ordenación, paginación con cursores
- **Mejor para:** apps de producción con datos complejos y consultas avanzadas

```
val db = Firebase.firestore

// Escribir documento
db.collection("productos").add(producto)
  .addOnSuccessListener { ref -> Log.d("FS", "ID: ${ref.id}") }

// Consulta con filtros
db.collection("productos")
  .whereEqualTo("activo", true)
  .orderBy("precio", Query.Direction.ASCENDING)
  .limit(20)
  .addSnapshotListener { snapshot, error ->
      val lista = snapshot?.toObjects(Producto::class.java)
  }
```

2. Mapas y GPS

2.1. Google Maps SDK

Para integrar mapas en Android es necesario obtener una API Key en **Google Cloud Console** y añadir el permiso `ACCESS_FINE_LOCATION` en el AndroidManifest.

Dependencias y configuración

```
// build.gradle.kts
implementation("com.google.android.gms:play-services-maps:18.x")
implementation("com.google.android.gms:play-services-location:21.x")

// AndroidManifest.xml
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<meta-data
    android:name="com.google.android.geo.API_KEY"
    android:value="TU_API_KEY"/>
```

Funcionalidades principales del mapa

Funcionalidad	Método / Clase	Descripción
Marcador	googleMap.addMarker(MarkerOptions())	Punto de interés con título e icono
Tipo de mapa	googleMap.mapType = MAP_TYPE_*	Normal, Satélite, Terreno, Híbrido
Cámara / Zoom	CameraUpdateFactory.newLatLngZoom()	Mover y enfocar el mapa
Polilínea	googleMap.addPolyline(PolylineOptions())	Trazar rutas y caminos

Funcionalidad	Método / Clase	Descripción
Polígono	googleMap.addPolygon(PolygonOptions())	Delimitar áreas geográficas
Círculo	googleMap.addCircle(CircleOptions())	Radio alrededor de un punto
Info Window	marker.showInfoWindow()	Ventana emergente al tocar marcador

Implementación básica

```
class MapActivity : AppCompatActivity(), OnMapReadyCallback {
    private lateinit var googleMap: GoogleMap

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_map)
        val mapFragment = supportFragmentManager
            .findFragmentById(R.id.map) as SupportMapFragment
        mapFragment.getMapAsync(this)
    }

    override fun onMapReady(map: GoogleMap) {
        googleMap = map
        val madrid = LatLng(40.4168, -3.7038)
        googleMap.addMarker(MarkerOptions()
            .position(madrid)
            .title("Madrid"))
        googleMap.animateCamera(
            CameraUpdateFactory.newLatLngZoom(madrid, 12f))
    }
}
```

2.2. Geolocalización

FusedLocationProviderClient es la API recomendada para obtener la ubicación del dispositivo. Fusiona GPS, Wi-Fi y redes móviles para ofrecer la mejor precisión con el menor consumo energético:

```
val fusedClient = LocationServices.getFusedLocationProviderClient(this)

// Última ubicación conocida
fusedClient.lastLocation.addOnSuccessListener { location ->
    location?.let {
        val lat = it.latitude
        val lng = it.longitude
    }
}

// Solicitar actualizaciones periódicas
val request = LocationRequest.Builder(
    Priority.PRIORITY_HIGH_ACCURACY, 5000L)
    .setMinUpdateIntervalMillis(2000L)
    .build()
```

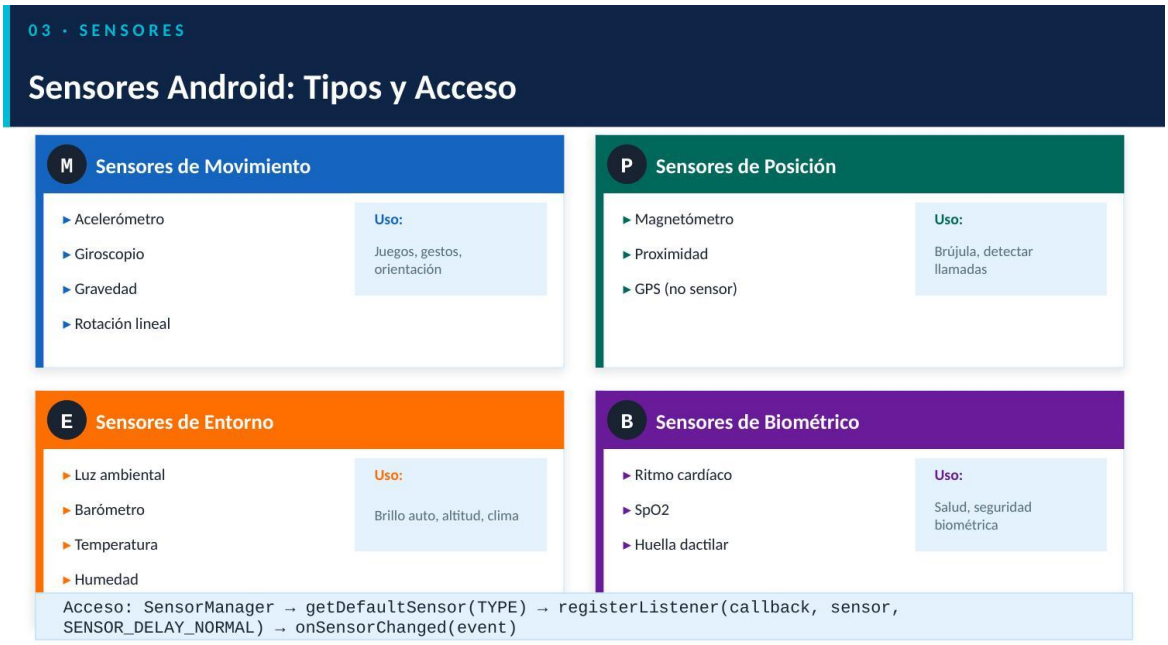
```
fusedClient.requestLocationUpdates(request, locationCallback,
    Looper.getMainLooper())

// Callback de ubicación
val locationCallback = object : LocationCallback() {
    override fun onLocationResult(result: LocationResult) {
        result.lastLocation?.let { loc ->
            Log.d("GPS", "Lat: ${loc.latitude}, Lng: ${loc.longitude}")
        }
    }
}
```

Geofencing

Las geovallas permiten detectar cuando el usuario entra o sale de una zona geográfica definida:

```
val geofence = Geofence.Builder()
    .setRequestId("oficina")
    .setCircularRegion(lat, lng, 150f)  // radio en metros
    .setTransitionTypes(
        Geofence.GEOFENCE_TRANSITION_ENTER or
        Geofence.GEOFENCE_TRANSITION_EXIT)
    .setExpirationDuration(Geofence.NEVER_EXPIRE)
    .build()

val request = GeofencingRequest.Builder()
    .addGeofence(geofence)
    .setInitialTrigger(GeofencingRequest.INITIAL_TRIGGER_ENTER)
    .build()

geofencingClient.addGeofences(request, pendingIntent)
```

3. Sensores



3.1. Tipos de Sensores Disponibles

Categoría	Sensores	Aplicaciones
Movimiento	Acelerómetro, Giroscopio, Gravedad, Rotación lineal	Juegos, gestos, orientación, detección de movimiento
Posición	Magnetómetro (brújula), Proximidad	Mapas, orientación, apagar pantalla en llamadas
Entorno	Luz ambiental, Barómetro, Temperatura, Humedad	Brillo automático, altitud, predicción del tiempo
Biométrico	Ritmo cardíaco, SpO2, Huella dactilar	Salud, fitness, autenticación biométrica

3.2. Acceso y Gestión de Sensores

El acceso a los sensores en Android se realiza siempre a través del patrón SensorManager + SensorEventListener:

```
class SensorActivity : AppCompatActivity(), SensorEventListener {
    private lateinit var sensorManager: SensorManager
    private var sensor: Sensor? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        sensorManager = getSystemService(Context.SENSOR_SERVICE) as
SensorManager
        sensor = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)
    }

    override fun onResume() {
        super.onResume()
        // Registrar en onResume para ahorrar batería
        sensorManager.registerListener(this, sensor,
            SensorManager.SENSOR_DELAY_NORMAL)
    }

    override fun onPause() {
        super.onPause()
        // SIEMPRE desregistrar en onPause
        sensorManager.unregisterListener(this)
    }

    override fun onSensorChanged(event: SensorEvent) {
        val x = event.values[0]  // Eje X
        val y = event.values[1]  // Eje Y
        val z = event.values[2]  // Eje Z
    }

    override fun onAccuracyChanged(sensor: Sensor?, accuracy: Int) {}
}
```

Velocidades de muestreo (SENSOR_DELAY)

Constante	Aprox. (µs)	Uso recomendado
SENSOR_DELAY_NORMAL	200.000	Cambios de orientación y UI
SENSOR_DELAY_UI	60.000	Actualizaciones de interfaz
SENSOR_DELAY_GAME	20.000	Juegos y realidad aumentada
SENSOR_DELAY_FASTEST	0 (máx. velocidad)	Análisis y procesamiento en tiempo real

Casos de uso prácticos

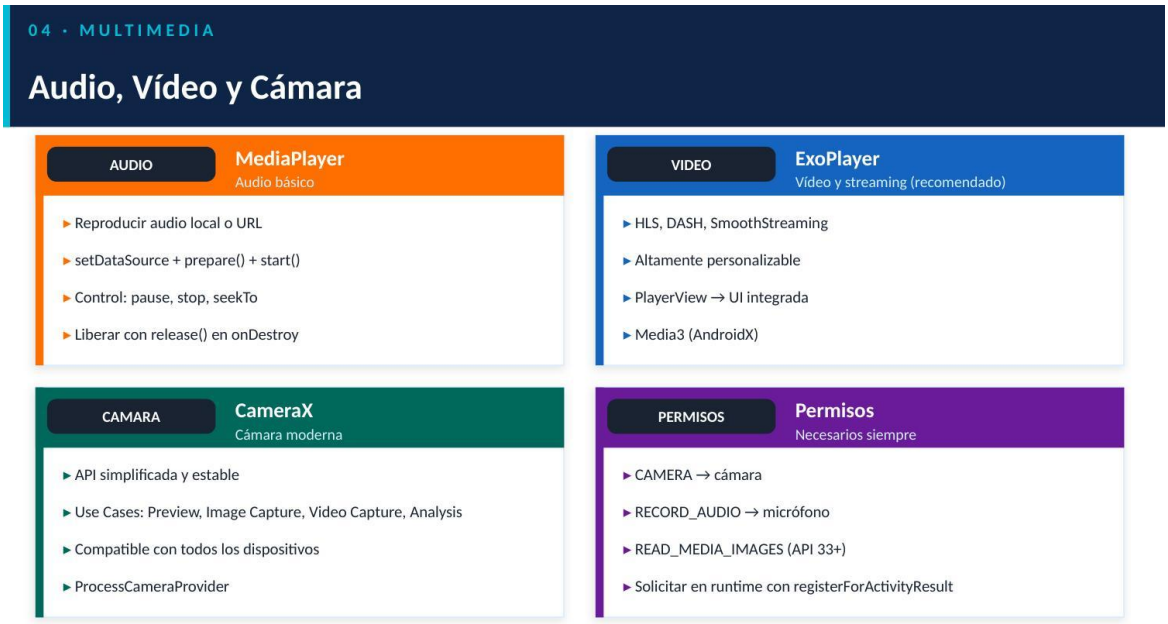
Detector de agitación (Shake Detector):

```
override fun onSensorChanged(event: SensorEvent) {
    if (event.sensor.type == Sensor.TYPE_ACCELEROMETER) {
        val x = event.values[0]
        val y = event.values[1]
        val z = event.values[2]
        val magnitude = Math.sqrt((x*x + y*y + z*z).toDouble())
        if (magnitude > SHAKE_THRESHOLD) {  // Ej: 15.0
            onShakeDetected()
        }
    }
}
```

Contador de pasos (Step Counter):

```
sensor = sensorManager.getDefaultSensor(Sensor.TYPE_STEP_COUNTER)
// TYPE_STEP_COUNTER: acumula pasos desde el último reinicio del dispositivo
// TYPE_STEP_DETECTOR: dispara evento por cada paso detectado
override fun onSensorChanged(event: SensorEvent) {
    val pasosTotales = event.values[0].toInt()
}
```

4. Multimedia



4.1. Reproducción de Audio y Vídeo

MediaPlayer — Audio básico

MediaPlayer permite reproducir audio desde recursos locales o URLs remotas. Es síncrono y requiere gestión cuidadosa del ciclo de vida:

```
var mediaPlayer: MediaPlayer? = null

// Reproducir desde recursos (res/raw/)
mediaPlayer = MediaPlayer.create(this, R.raw.musica)
mediaPlayer?.start()

// Reproducir desde URL
mediaPlayer = MediaPlayer().apply {
    setDataSource("https://ejemplo.com/audio.mp3")
    setOnPreparedListener { start() }
    setOnErrorListener { _, what, _ ->
        Log.e("MP", "Error: $what"); true
    }
    prepareAsync()  // No bloquea el hilo principal
}

// Controles
mediaPlayer?.pause()
mediaPlayer?.seekTo(30000)  // milisegundos
mediaPlayer?.stop()

// IMPORTANTE: liberar recursos
override fun onDestroy() {
    super.onDestroy()
    mediaPlayer?.release()
    mediaPlayer = null
}
```

ExoPlayer — Vídeo y streaming (recomendado)

ExoPlayer (ahora **Media3**) es la solución moderna para reproducción de vídeo, streaming HLS/DASH y audio avanzado:

- **Dependencia:** `implementation("androidx.media3:media3-exoplayer:1.x")`
- Soporta HLS, DASH, SmoothStreaming, MP4, MP3, OGG...
- **PlayerView:** componente UI que incluye controles de reproducción

```
// Layout: <androidx.media3.ui.PlayerView android:id="@+id/player_view"/>

private var player: ExoPlayer? = null

override fun onStart() {
    super.onStart()
    player = ExoPlayer.Builder(this).build().also {
        binding.playerView.player = it
        val mediaItem = MediaItem.fromUri(
            "https://ejemplo.com/video.m3u8")
        it.setMediaItem(mediaItem)
        it.prepare()
        it.playWhenReady = true
    }
}

override fun onStop() {
    super.onStop()
    player?.release()
    player = null
}
```

4.2. Captura de Imagen y Vídeo con CameraX

04 · MULTIMEDIA

ExoPlayer y CameraX — Código Esencial

ExoPlayer (Media3)

```
// Dependencia
// implementation("androidx.media3:media3-
exoplayer:1.x")

// Layout: <PlayerView
android:id="@+id/player_view"/>

private var player: ExoPlayer? = null

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    val player = ExoPlayer.Builder(this).build()
    binding.playerView.player = player

    val mediaItem = MediaItem.fromUri(
        "https://example.com/video.mp4")
    player.setMediaItem(mediaItem)
    player.prepare()
    player.play()
}

override fun onDestroy() {
    super.onDestroy()
    player?.release()
    player = null
}
```

CameraX — Preview + Captura

```
// Dependencia
// implementation("androidx.camera:camera-camera2:1.x")

private lateinit var cameraProvider:
ProcessCameraProvider
private lateinit var imageCapture: ImageCapture

fun startCamera() {
    val cameraProviderFuture =
        ProcessCameraProvider.getInstance(this)

    cameraProviderFuture.addListener({
        cameraProvider = cameraProviderFuture.get()
        val preview = Preview.Builder().build().also {
            it.setSurfaceProvider(
                binding.viewFinder.surfaceProvider)
        }
        imageCapture = ImageCapture.Builder().build()

        cameraProvider.bindToLifecycle(
            this,
            CameraSelector.DEFAULT_BACK_CAMERA,
            preview, imageCapture)
    }, ContextCompat.getMainExecutor(this))
}
```

CameraX es la API moderna de cámara de AndroidX. Proporciona una interfaz consistente en todos los dispositivos y versiones de Android:

- **Preview:** Vista previa en tiempo real
- **ImageCapture:** Capturar fotografías
- **VideoCapture:** Grabar vídeo
- **ImageAnalysis:** Análisis de frames en tiempo real (IA, códigos QR...)

```
// Dependencias
// implementation("androidx.camera:camera-camera2:1.x")
// implementation("androidx.camera:camera-lifecycle:1.x")
// implementation("androidx.camera:camera-view:1.x")

private lateinit var imageCapture: ImageCapture

fun startCamera() {
    val cameraProviderFuture = ProcessCameraProvider.getInstance(this)
    cameraProviderFuture.addListener({
        val cameraProvider = cameraProviderFuture.get()

        val preview = Preview.Builder().build().also {
            it.setSurfaceProvider(binding.viewFinder.surfaceProvider)
        }
        imageCapture = ImageCapture.Builder()
            .setCaptureMode(ImageCapture.CAPTURE_MODE_MINIMIZE_LATENCY)
            .build()

        cameraProvider.bindToLifecycle(
            this, CameraSelector.DEFAULT_BACK_CAMERA,
            preview, imageCapture)
    }, ContextCompat.getMainExecutor(this))
}

fun tomarFoto() {
    val file = File(filesDir, "foto.jpg")
    val output = ImageCapture.OutputFileOptions.Builder(file).build()
    imageCapture.takePicture(output,
        ContextCompat.getMainExecutor(this),
        object : ImageCapture.OnImageSavedCallback {
            override fun onImageSaved(result: OutputFileResults) {
                Log.d("CameraX", "Foto guardada")
            }
            override fun onError(exc: ImageCaptureException) {}
        })
}
```

Permisos de Cámara y Micrófono

> *Los permisos CAMERA y RECORD_AUDIO son permisos peligrosos que deben solicitarse en tiempo de ejecución. En Android 13+ también se requiere READ_MEDIA_IMAGES para acceder a la galería.*

```
// AndroidManifest.xml
```

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

// Solicitar permisos en runtime
val requestPermission = registerForActivityResult(
    ActivityResultContracts.RequestMultiplePermissions()) { perms ->
    val cameraOk = perms[Manifest.permission.CAMERA] == true
    val audioOk  = perms[Manifest.permission.RECORD_AUDIO] == true
    if (cameraOk && audioOk) startCamera()
}

requestPermission.launch(arrayOf(
    Manifest.permission.CAMERA,
    Manifest.permission.RECORD_AUDIO
))
```

5. Widgets

5.1. Creación de AppWidgets

Un **AppWidget** es un componente que muestra información de la app directamente en el escritorio del usuario. Se actualiza con **RemoteViews**, que es una vista especial que puede ejecutarse en otro proceso.

Componentes necesarios

Componente	Descripción
Layout XML	Diseño del widget con Views compatibles con RemoteViews
Widget Info XML	Configuración: tamaño mínimo, periodo de actualización, preview
AppWidgetProvider	BroadcastReceiver que gestiona los eventos del widget
AndroidManifest	Registro del receptor con el filtro ACTION_APPWIDGET_UPDATE

Widget Info XML (res/xml/widget_info.xml)

```
<appwidget-provider
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:minWidth="110dp"
    android:minHeight="40dp"
    android:updatePeriodMillis="1800000"
    android:previewImage="@drawable/widget_preview"
    android:initialLayout="@layout/widget_layout"
    android:resizeMode="horizontal|vertical"
    android:widgetCategory="home_screen"/>
```

AppWidgetProvider

```
class MiWidget : AppWidgetProvider() {
    override fun onUpdate(
        context: Context,
        manager: AppWidgetManager,
        appWidgetIds: IntArray
    ) {
        appWidgetIds.forEach { id ->
            val views = RemoteViews(context.packageName,
                R.layout.widget_layout)

            // Actualizar texto
            views.setTextViewText(R.id.tv_widget,
                "Actualizado: ${java.util.Date()}")

            // Acción al tocar el widget
            val intent = Intent(context, MainActivity::class.java)
            val pi = PendingIntent.getActivity(context, 0, intent,
                PendingIntent.FLAG_IMMUTABLE)
            views.setOnClickPendingIntent(R.id.tv_widget, pi)

            manager.updateAppWidget(id, views)
        }
    }
}
```

5.2. Mejores Prácticas de Widgets

- **updatePeriodMillis:** El mínimo permitido es 30 minutos (1.800.000 ms). Para actualización frecuente usar AlarmManager o WorkManager.
- **Views disponibles:** Solo las subclases de RemoteViews soportadas: TextView, ImageView, Button, ProgressBar, ListView, GridView, StackView...
- **Rendimiento:** El widget se ejecuta en un proceso separado. Mantener el código del onUpdate() ligero y rápido.
- **Colecciones:** Para listas en widgets usar RemoteViewsService + RemoteViewsFactory.

6. Servicios Web

6.1. Retrofit: Consumo de APIs REST

Retrofit es la librería más popular para consumir APIs REST en Android. Convierte interfaces Kotlin en llamadas HTTP automáticamente, integrándose con corrutinas y Gson/Moshi para serialización JSON.

Stack tecnológico completo

Librería	Rol en la arquitectura
Retrofit	Cliente HTTP — define los endpoints y convierte respuestas
OkHttp	Capa de red — interceptores, caché, timeouts, logs
Gson / Moshi	Serialización — convierte JSON en objetos Kotlin y viceversa
Corrutinas	Asincronía — suspend functions para llamadas no bloqueantes

Configuración de Retrofit

```
// Dependencias (build.gradle.kts)
implementation("com.squareup.retrofit2:retrofit:2.x")
implementation("com.squareup.retrofit2:converter-gson:2.x")
implementation("com.squareup.okhttp3:logging-interceptor:4.x")

// Singleton Retrofit
object RetrofitClient {
    private const val BASE_URL = "https://api.ejemplo.com/"

    val instance: ApiService by lazy {
        val logging = HttpLoggingInterceptor().apply {
            level = HttpLoggingInterceptor.Level.BODY
```

```
        }
        val client = OkHttpClient.Builder()
            .addInterceptor(logging)
            .connectTimeout(30, TimeUnit.SECONDS)
            .readTimeout(30, TimeUnit.SECONDS)
            .build()

        Retrofit.Builder()
            .baseUrl(BASE_URL)
            .client(client)
            .addConverterFactory(GsonConverterFactory.create())
            .build()
            .create(ApiService::class.java)
    }
}
```

Interface de la API

```
interface ApiService {
    @GET("users")
    suspend fun getUsers(): Response<List<User>>

    @GET("users/{id}")
    suspend fun getUser(@Path("id") id: Int): Response<User>

    @POST("users")
    suspend fun createUser(@Body user: UserRequest): Response<User>

    @PUT("users/{id}")
    suspend fun updateUser(
        @Path("id") id: Int,
        @Body user: UserRequest
    ): Response<User>

    @GET("products")
    suspend fun searchProducts(
        @Query("q") query: String,
        @Query("page") page: Int = 1
    ): Response<ProductResponse>
}
```

Uso en ViewModel con corrutinas

```
class UserViewModel : ViewModel() {
    private val _users = MutableStateFlow<List<User>>(emptyList())
    val users: StateFlow<List<User>> = _users.asStateFlow()

    fun loadUsers() {
        viewModelScope.launch {
            try {
                val response = RetrofitClient.instance.getUsers()
                if (response.isSuccessful) {
                    _users.value = response.body() ?: emptyList()
                } else {
                    Log.e("VM", "Error: ${response.code()}")
```

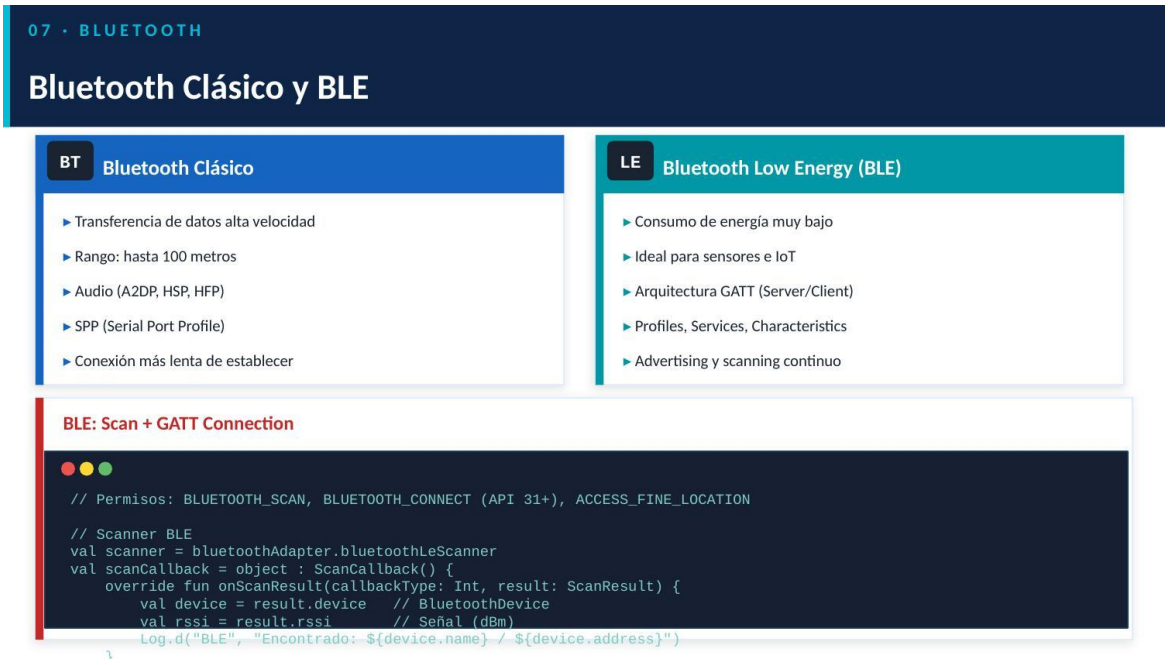
```
                }
            } catch (e: IOException) {
                Log.e("VM", "Sin conexión: ${e.message}")
            }
        }
    }
}
```

6.2. Autenticación con JWT

El patrón más común para autenticación en APIs REST es JWT (JSON Web Token). Se obtiene un token en el login y se adjunta a todas las peticiones posteriores:

```
// Interceptor para añadir el token automáticamente
class AuthInterceptor(private val tokenProvider: () -> String?) :
Interceptor {
    override fun intercept(chain: Interceptor.Chain): Response {
        val original = chain.request()
        val token = tokenProvider()
        return if (token != null) {
            val request = original.newBuilder()
                .header("Authorization", "Bearer $token")
                .build()
            chain.proceed(request)
        } else {
            chain.proceed(original)
        }
    }
}
```

7. Bluetooth



7.1. Bluetooth Clásico vs BLE

Característica	Bluetooth Clásico	BLE (Low Energy)
Velocidad datos	Alta (hasta 3 Mbps)	Baja (1 Mbps)
Consumo energía	Alto	Muy bajo
Rango	Hasta 100 metros	Hasta 100 metros
Latencia conexión	Alta (~2 seg)	Baja (~3 ms)
Casos de uso	Audio, transferencia de archivos	IoT, wearables, sensores
Arquitectura	Perfiles (A2DP, SPP, HFP...)	GATT (Servicios + Características)

Permisos necesarios (API 31+)

```
<!-- AndroidManifest.xml -->
<uses-permission android:name="android.permission.BLUETOOTH_SCAN"
    android:usesPermissionFlags="neverForLocation"/>
<uses-permission android:name="android.permission.BLUETOOTH_CONNECT"/>
<uses-permission android:name="android.permission.BLUETOOTH_ADVERTISE"/>
<!-- Para API < 31: BLUETOOTH, BLUETOOTH_ADMIN, ACCESS_FINE_LOCATION -->
```

7.2. Bluetooth Low Energy (BLE)

BLE usa la arquitectura GATT (Generic Attribute Profile) para comunicarse. Un dispositivo GATT Server expone Servicios que contienen Características con valores legibles/escribibles:

Escaneo de dispositivos BLE

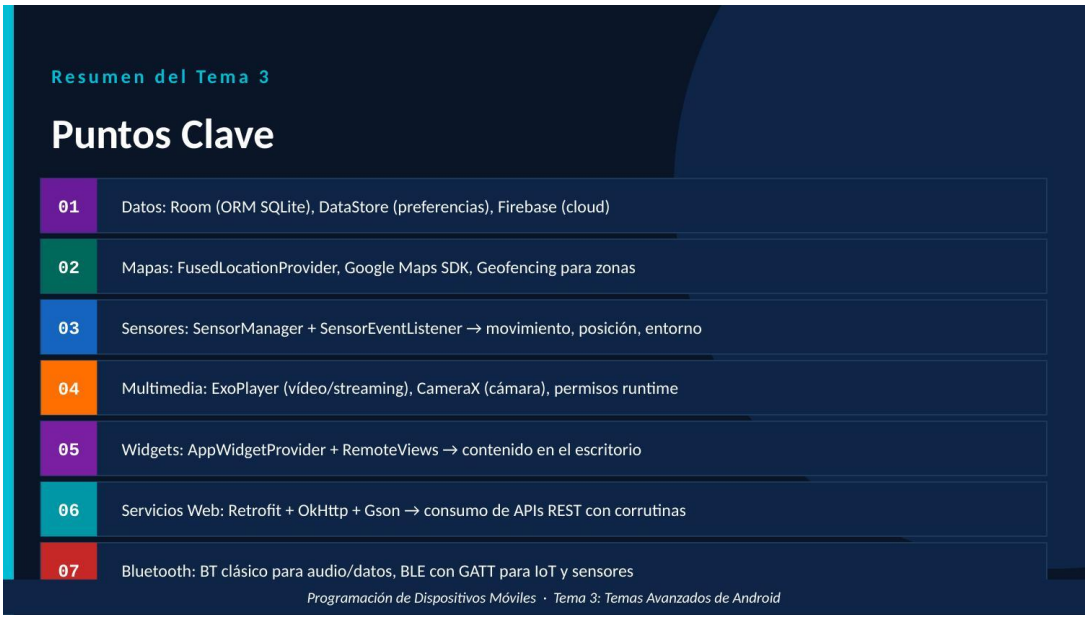
```
val bluetoothManager = getSystemService(BluetoothManager::class.java)
val adapter = bluetoothManager.adapter
val scanner = adapter.bluetoothLeScanner

// Configurar filtros de escaneo
val filters = listOf(ScanFilter.Builder()
    .setServiceUuid(ParcelUuid(MY_SERVICE_UUID))
    .build())
val settings = ScanSettings.Builder()
    .setScanMode(ScanSettings.SCAN_MODE_LOW_LATENCY)
    .build()

val scanCallback = object : ScanCallback() {
    override fun onScanResult(callbackType: Int, result: ScanResult) {
        val device = result.device
        Log.d("BLE", "Dispositivo: ${device.name} / ${device.address}")
    }
    override fun onScanFailed(errorCode: Int) {
        Log.e("BLE", "Error de escaneo: $errorCode")
    }
}

scanner.startScan(filters, settings, scanCallback)
// Detener escaneo (importante para ahorrar batería)
scanner.stopScan(scanCallback)
```

Conexión GATT y lectura de características

```
val gatt = device.connectGatt(context, false,
    object : BluetoothGattCallback() {

    override fun onConnectionStateChange(
        gatt: BluetoothGatt, status: Int, newState: Int) {
        if (newState == BluetoothProfile.STATE_CONNECTED) {
            gatt.discoverServices()  // Descubrir servicios
        }
    }

    override fun onServicesDiscovered(
        gatt: BluetoothGatt, status: Int) {
        val service = gatt.getService(SERVICE_UUID)
        val characteristic = service
            ?.getCharacteristic(CHARACTERISTIC_UUID)
        characteristic?.let { gatt.readCharacteristic(it) }
    }

    override fun onCharacteristicRead(
        gatt: BluetoothGatt,
        characteristic: BluetoothGattCharacteristic,
        value: ByteArray, status: Int) {
        val datos = String(value)  // Procesar datos recibidos
        runOnUiThread { actualizarUI(datos) }
    }

    override fun onCharacteristicChanged(
        gatt: BluetoothGatt,
```

```
        characteristic: BluetoothGattCharacteristic,
        value: ByteArray) {
        // Notificaciones automáticas del servidor
        val temperatura = value[0].toFloat() / 10f
    }
})

// No olvidar cerrar la conexión al terminar
gatt?.close()
```

8. Resumen y Actividades



8.1. Puntos Clave del Tema

> *Datos: SharedPreferences para preferencias simples, DataStore (Jetpack) para configuraciones reactivas, Room para bases de datos locales con SQLite, Firebase para sincronización en tiempo real o cloud.*

> *Mapas: FusedLocationProviderClient ofrece la ubicación más precisa con menor consumo. Google Maps SDK permite marcadores, polilíneas, polígonos y tipos de mapa. Geofencing detecta entradas/salidas de zonas.*

> *Sensores: SensorManager + SensorEventListener con registerListener en onResume y unregisterListener en onPause. Categorías: movimiento (acelerómetro, giroscopio), posición (magnetómetro, proximidad), entorno (luz, barómetro) y biométrico.*

> *Multimedia: MediaPlayer para audio básico, ExoPlayer/Media3 para vídeo y streaming (HLS, DASH), CameraX para cámara con Preview, ImageCapture y ImageAnalysis. Permisos CAMERA y RECORD_AUDIO en runtime.*

> *Widgets: AppWidgetProvider + RemoteViews para mostrar contenido en el escritorio. El periodo mínimo de actualización automática es 30 minutos; para más frecuencia usar WorkManager.*

> *Servicios Web: Retrofit + OkHttp + Gson/Moshi para consumir APIs REST con corrutinas (suspend functions). Interceptores de OkHttp para añadir headers JWT. Patrón: ViewModel → Repository → ApiService.*

> *Bluetooth: BT clásico para audio y transferencia de datos de alta velocidad. BLE para IoT y wearables con bajo consumo, usando GATT (Servicios + Características). Permisos BLUETOOTH_SCAN y BLUETOOTH_CONNECT desde API 31.*

8.2. Actividades Prácticas

1. **Room — Gestión de contactos:** Crear app con Room para CRUD completo de contactos. Incluir Entity, DAO, Repository y ViewModel. Mostrar lista con RecyclerView y permitir añadir, editar y eliminar.
2. **Mapas — Seguimiento de rutas:** Implementar app que registre la ruta del usuario con Google Maps y FusedLocationProvider. Dibujar la ruta como polilínea en el mapa en tiempo real.
3. **Sensores — Podómetro:** Desarrollar podómetro que cuente pasos con TYPE_STEP_COUNTER y muestre la distancia estimada. Añadir detector de agitación para reiniciar el contador.
4. **Multimedia — Reproductor de música:** Crear reproductor con ExoPlayer que cargue canciones de una URL, con controles de play/pause/siguiente y barra de progreso.
5. **Widget del clima:** Diseñar AppWidget que muestre temperatura y descripción del tiempo. Actualizar cada 30 minutos con WorkManager y una API meteorológica.
6. **APIs REST — Consumo de JSONPlaceholder:** Consumir la API pública JSONPlaceholder (https://jsonplaceholder.typicode.com) con Retrofit. Mostrar lista de posts y permitir crear nuevos.
7. **BLE Scanner:** Escanear dispositivos BLE cercanos y mostrarlos en una lista con nombre, dirección MAC y nivel de señal RSSI. Conectar a uno y leer una característica.

8.3. Recursos Adicionales

- Room Database Documentation — Guía oficial de almacenamiento con Room
- Google Maps Platform — SDK de Mapas para Android
- Sensors Overview — Referencia completa de sensores Android
- Media3 / ExoPlayer — Documentación oficial de reproducción multimedia
- App Widgets Guide — Crear widgets para el escritorio
- Retrofit — Librería HTTP para Android
- Bluetooth Overview — Guía de Bluetooth y BLE en Android