Reservas Ucam: Gestor de Reservas de Mesas en Bibliotecas

Aplicación Android para gestionar reservas de mesas en servicios de biblioteca integrada con TakeASpot.

Una aplicación Android nativa desarrollada con **Kotlin + Jetpack Compose** que permite a usuarios autenticarse mediante Firebase y reservar mesas en bibliotecas a través de la API de TakeASpot, con soporte para múltiples cuentas.

Tabla de Contenidos

Visión General

Reservas UCAM es una aplicación Android que resuelve el problema de gestionar reservas de mesas en bibliotecas de forma cómoda e intuitiva. El usuario puede:

1. **Autenticarse** mediante correo y contraseña con Firebase Auth
2. **Gestionar múltiples cuentas** (diferentes cuentas de biblioteca)
3. **Buscar y reservar mesas** en servicios de biblioteca
4. **Ver y cancelar** sus reservas activas
5. **Sincronizar datos** en tiempo real entre dispositivos

Caso de Uso Real

La biblioteca de la UCAM (servicio TakeASpot) requiere que los usuarios reserven mesas para garantizar disponibilidad. Esta app simplifica ese proceso, eliminando la necesidad de acceder a interfaces web complejas.

Tech Stack

Desarrollo

Componente	Tecnología	Versión
Lenguaje	Kotlin	1.9.22
Android	API nivel	26+ (minSdk), 34 (targetSdk)
UI Framework	Jetpack Compose	2024.02.00 BOM
Inyección de Dependencias	Hilt	Latest (with KAPT)
Networking	Retrofit + OkHttp	2.x
JSON Parsing	Gson	Latest
HTML Parsing	Jsoup	1.17.2

Backend & Servicios

Servicio	Propósito
Firebase Auth	Autenticación segura de usuarios
Firestore	Almacenamiento de cuentas y datos en tiempo real
TakeASpot API	Integración con servicio de reservas (biblioteca)
OkHttp CookieJar	Gestión automática de sesiones HTTP

Seguridad

- **EncryptedSharedPreferences**: Almacenamiento cifrado de sesiones
- **API level 26+**: Requisitos de seguridad modernos
- **ProGuard**: Ofuscación de código en builds de release

Arquitectura

El proyecto sigue el patrón **Clean Architecture** con tres capas bien definidas:

```
            UI Layer (Presentation)
  Composables, ViewModels, Navigation



          ViewModel & Domain Layer
  UseCase, Interfaces, Modelos de Negocio
```

```
          Data Layer (Implementation)
  Repositories, APIs, Local Storage
```

Estructura de Directorios

```
app/src/main/java/edu/ucam/reservashack/
  domain/                          # Lógica de negocio pura
     model/                        # Modelos de dominio
        LibraryService.kt
        DaySlots.kt
        TableStatus.kt
        MyBooking.kt
     repository/                   # Interfaces de contrato
        LibraryRepository.kt
        AccountRepository.kt
        SessionRepository.kt
     usecase/                      # Casos de uso opcionales
         RequireActiveAccountUseCase.kt

  data/                            # Implementación de datos
     remote/                       # APIs externas
        TakeASpotApi.kt            # Interface Retrofit
        SessionCookieJar.kt        # Gestión de cookies
        ApiErrorHandler.kt         # Manejo de errores
        dto/                       # Data Transfer Objects
            ServiceDto.kt
            BookingDto.kt
     local/                        # Almacenamiento local
        SessionRepositoryImpl.kt   # Encrypted SharedPrefs
     repository/                   # Implementaciones
         LibraryRepositoryImpl.kt
         AccountRepositoryImpl.kt

  ui/                              # Presentación
     screens/                      # Pantallas por funcionalidad
        home/
            HomeScreen.kt
            HomeViewModel.kt
        mybookings/
            MyBookingsScreen.kt
            MyBookingsViewModel.kt
        login/
            FirebaseLoginScreen.kt
            FirebaseLoginViewModel.kt
```

```
        search/
        profile/
    navigation/                  # Configuración de NavController
        AppNavigation.kt
    shared/                      # Componentes compartidos
        SharedEventViewModel.kt  # Comunicación entre pantallas
    theme/                       # Material Design 3
        ReservasHackTheme.kt

di/                               # Inyección de Dependencias
    FirebaseModule.kt            # Instancias de Firebase
    NetworkModule.kt             # Retrofit, OkHttp, APIs
    RepositoryModule.kt          # Bindings de repositorios

MainActivity.kt                   # Entry point
MainViewModel.kt                  # Estado global
BookingApp.kt                     # Application class con Hilt
```

Flujo de Datos

```
Composable UI

        observa state via collectAsState()


  ViewModel        Contiene mutableStateOf/StateFlow

        llamadas a funciones


Repository         Abstracción de acceso a datos

        implementa


TakeASpotApi (Retrofit)        HTTP calls
+ SessionCookieJar             Multipart/form-data




API TakeASpot (servidor)       /myturner/api/*
```

Patrones Utilizados

1. Inyección de Dependencias con Hilt

```
@HiltViewModel
class HomeViewModel @Inject constructor(
    private val libraryRepository: LibraryRepository,
    private val sharedEventViewModel: SharedEventViewModel,
) : ViewModel() { ... }
```

Beneficio: Facilita testing, modularidad y reduce acoplamiento.

2. Gestión de Estado con mutableStateOf

```
var state by mutableStateOf<HomeState>(HomeState.Loading)
    private set  // Setter privado para evitar mutaciones externas
```

Beneficio: Recomposición automática y control total del ciclo de vida.

3. Result Type para Manejo de Errores

```
suspend fun getLibraryInfo(): Result<LibraryService> = try {
    Result.success(api.getServices(...).toDomain())
} catch (e: Exception) {
    Result.failure(e)
}
```

Beneficio: Evita excepciones deschecadas; obliga a manejar errores explícitamente.

4. Sealed Classes para Estados

```
sealed class MyBookingsState {
    object Loading : MyBookingsState()
    data class Success(val bookings: List<MyBooking>) : MyBookingsState()
    data class Error(val message: String) : MyBookingsState()
}
```

Beneficio: Exhaustiveness checking en `when` expressions.

5. Eventos Compartidos entre Pantallas

```
// En SharedEventViewModel
val reservationMadeEvent = MutableSharedFlow<Unit>()

// Emisión desde HomeViewModel tras reserva exitosa
viewModelScope.launch {
    sharedEventViewModel.reservationMadeEvent.emit(Unit)
}
```

```
// Suscripción en MyBookingsViewModel para recargar
viewModelScope.launch {
    sharedEventViewModel.reservationMadeEvent.collect {
        loadBookings()
    }
}
```

Beneficio: Comunicación desacoplada entre ViewModels.

Funcionalidades Principales

1. Autenticación de Usuarios

- Login con correo/contraseña mediante Firebase Auth
- Persistencia de sesión automática
- Support para múltiples cuentas de biblioteca

Archivos relevantes: - `FirebaseLoginScreen.kt` / `FirebaseLoginViewModel.kt` - `MainActivity.kt` (AuthStateListener)

2. Gestión de Cuentas Múltiples

- Almacenamiento de múltiples credenciales de biblioteca por usuario Firebase
- Selección de cuenta activa en pantalla de Perfil
- Sincronización via Firestore: `users/{uid}/accounts/{docId}`

Archivos relevantes: - `AccountRepositoryImpl.kt` - `ProfileScreen.kt`

3. Búsqueda y Filtrado de Mesas

- Lista de servicios de biblioteca disponibles (e.g., Universidad de Murcia)
- Selección de fecha y horario
- Visualización de disponibilidad en tiempo real

Archivos relevantes: - `HomeScreen.kt` / `HomeViewModel.kt` - `SearchScreen.kt` - `TakeASpotApi.getServices()` y `getServiceSlots()`

4. Reserva de Mesas

- Selección de mesa y cantidad de personas
- Reserva inmediata con feedback al usuario
- Evento broadcast a otras pantallas para actualización

Archivos relevantes: - `HomeViewModel.makeReservation()` - `TakeASpotApi.makeReservation()`

5. Gestión de Reservas Activas

- Listado de todas las reservas del usuario
- Detalles de cada reserva (fecha, hora, mesa, servicio)
- Cancelación de reservas con confirmación

Archivos relevantes: - `MyBookingsScreen.kt` / `MyBookingsViewModel.kt` - `TakeASpotApi.getMyBookings()` + parsing HTML con Jsoup - `TakeASpotApi.cancelBooking()`

6. Persistencia de Sesión

- Almacenamiento cifrado de cookies de sesión TakeASpot
- Expiración automática de sesiones después de 24h
- CookieJar integrado con OkHttp para inyección automática

Archivos relevantes: - `SessionCookieJar.kt` - `SessionRepositoryImpl.kt` - `EncryptedSharedPreferences`

Instalación y Compilación

Requisitos Previos

- **Android Studio** Hedgehog (2023.1.1) o superior
- **SDK de Android** minSdk=26, targetSdk=34
- **JDK 8+** (recomendado 11 o 17)
- **Kotlin** 1.9.22 (incluido en Android Studio)
- Dispositivo o emulador con **Android 8.0+**

Pasos de Instalación

1. Clonar el Repositorio

```
git clone <url-del-repositorio>
cd reservashack
```

2. Configurar `local.properties` Crear archivo `local.properties` en la raíz del proyecto:

```
sdk.dir=/ruta/a/Android/Sdk
```

Nota: En Windows, Android Studio puede auto-generar este archivo.

3. Configurar Firebase

1. Ir a Firebase Console
2. Crear nuevo proyecto o usar uno existente
3. Registrar aplicación Android (package: `edu.ucam.reservashack`)
4. Descargar `google-services.json`

5. Colocar en `app/google-services.json`

Nota: El archivo está `.gitignore` por seguridad. Es obligatorio para compilar.

4. Compilar la Aplicación **Build Debug (desarrollo)**:

```
./gradlew clean build
```

o desde Android Studio:

```
Build > Make Project (Ctrl+F9)
```

Build Release (producción):

```
./gradlew clean assembleRelease
```

Esto genera: - APK en: `app/build/outputs/apk/release/app-release.apk` - Minificado y ofuscado con ProGuard

5. Instalar en Dispositivo/Emulador **Vía Android Studio**: 1. Conectar dispositivo o iniciar emulador 2. Clic en **Run** (Shift+F10)

Vía Terminal:

```
# Instalar APK debug
./gradlew installDebug

# Iniciar aplicación
adb shell am start -n edu.ucam.reservashack/.MainActivity
```

6. Verificar Instalación En el dispositivo, debería aparecer la app "ReservasHack" con pantalla de login.

Configuración Inicial

1. **Registro de usuario**:
   - Ingresar correo y contraseña en Firebase
   - Confirmar correo si es requerido
2. **Agregar cuenta de biblioteca**:
   - Ir a pantalla Perfil
   - Ingresar credenciales de TakeASpot
   - Seleccionar como cuenta activa
3. **Realizar reserva**:
   - Ir a pantalla Home
   - Buscar servicio, fecha, hora y mesa
   - Confirmar reserva

Problemas Encontrados y Soluciones

1. Configuración de Hilt y KAPT

Problema: - Errores de compilación como "Cannot find symbol @HiltViewModel" - KAPT no genera archivos de inyección - Conflictos entre plugins (Hilt, Google Services)

Causa Raíz: - KAPT debe ejecutarse antes de la compilación principal - Orden incorrecto de plugins en `build.gradle.kts`

Solución:

```
// build.gradle.kts (app level)
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.kotlin.android)
    id("kotlin-kapt")              // ANTES de Hilt
    alias(libs.plugins.hilt.android) // Hilt es AFTER
    alias(libs.plugins.google.services)
}

dependencies {
    implementation(libs.hilt.android)
    kapt(libs.hilt.compiler)        // Usar kapt, NO implementation
}
```

Lección aprendida: El orden de plugins importa mucho. Consultar documentación oficial de Hilt.

---

2. Gestión Dual de Sesiones (Firebase + TakeASpot)

Problema: - Firebase Auth guarda un usuario - TakeASpot requiere cookies de sesión separadas - Logout de Firebase no limpia cookies de TakeASpot - Múltiples cuentas TakeASpot por usuario Firebase son complicadas

Causa Raíz: - Firebase y TakeASpot son dos sistemas de autenticación independientes - OkHttp CookieJar maneja cookies globalmente (una sesión a la vez) - No hay sincronización automática

Solución Implementada:

```
// SessionCookieJar intercepta requests y respuestas
class SessionCookieJar @Inject constructor(
    private val sessionRepository: SessionRepository
) : CookieJar {
    override fun saveFromResponse(url: HttpUrl, cookies: List<Cookie>) {
        // Al login, guardar cookies de sesión en repo encriptado
        val sessionCookie = cookies.find { it.name == "takeaspot_session" }
```

```
            val xsrfToken = cookies.find { it.name == "XSRF-TOKEN" }
            if (sessionCookie != null) {
                sessionRepository.saveSession(
                    Session(
                        sessionCookie = sessionCookie.value,
                        xsrfToken = xsrfToken?.value,
                        expiresAt = System.currentTimeMillis() + SESSION_DURATION_MS
                    )
                )
            }
        }

        override fun loadForRequest(url: HttpUrl): List<Cookie> {
            // Al hacer request, inyectar cookies guardadas
            val session = sessionRepository.getSession() ?: return emptyList()
            return listOf(
                Cookie.Builder().name("takeaspot_session")
                    .value(session.sessionCookie).domain(url.host).build()
            )
        }
    }

    // Inyectar en OkHttp
    class NetworkModule {
        @Provides
        @Singleton
        fun provideOkHttpClient(
            cookieJar: SessionCookieJar
        ): OkHttpClient {
            return OkHttpClient.Builder()
                .cookieJar(cookieJar)
                .addInterceptor(HttpLoggingInterceptor().setLevel(...))
                .build()
        }
    }
```

Resultado: Cada usuario Firebase puede tener múltiples cuentas TakeASpot almacenadas en Firestore; la activa se usa para validación.

---

3. Parsing HTML de Bookings con Jsoup

Problema: - La API de TakeASpot devuelve bookings como página HTML, no JSON - Estructura HTML frágil y propensa a cambios - Necesidad de parsear tabla con Jsoup

Causa Raíz: - API legacy (TakeASpot) no tiene endpoint JSON para bookings - Server devuelve HTML renderizado

Solución:

```
// En LibraryRepositoryImpl.getMyBookings()
override suspend fun getMyBookings(): Result<List<MyBooking>> {
    return try {
        val htmlResponse = api.getMyBookings() // String con HTML
        val doc = Jsoup.parse(htmlResponse)

        // Selector CSS para tabla de bookings
        val bookings = doc.select("table.bookings tbody tr").mapNotNull { row ->
            val cells = row.select("td")
            if (cells.size >= 5) {
                MyBooking(
                    id = cells[0].text().toInt(),
                    date = cells[1].text(),
                    time = cells[2].text(),
                    people = cells[3].text().toInt(),
                    table = cells[4].text(),
                )
            } else null
        }

        Result.success(bookings)
    } catch (e: Exception) {
        Result.failure(e)
    }
}
```

Riesgo: Si TakeASpot cambia estructura HTML, parsing falla. **Mitigación**: Agregar logs y fallback a datos en caché.

---

4. Multipart Requests con Retrofit

Problema: - Retrofit necesita enviar datos en forma `multipart/form-data` - RequestBody requiere conversión manual de strings - Olvido de conversión causa errores 400 Bad Request

Causa Raíz: - Retrofit no auto-convierte strings a RequestBody - Documentación de TakeASpot requiere tipos específicos

Solución:

```
// En TakeASpotApi
@Multipart
```

```
@POST("myturner/api/make-booking")
suspend fun makeReservation(
    @Part("people") people: RequestBody,        // Convertir manualmente
    @Part("date") date: RequestBody,            // Ej: "2024-01-15"
    @Part("hour") hour: RequestBody,            // Ej: "10:00-11:00"
    @Part("serviceId") serviceId: RequestBody,
    @Part("tableId") tableId: RequestBody,
): Response<ReservationResponse>

// En HomeViewModel
fun makeReservation(tableId: Int, peopleCount: Int, date: String, hour: String) {
    viewModelScope.launch {
        val result = try {
            api.makeReservation(
                people = peopleCount.toString().toRequestBody("text/plain".toMediaTypeOrNull
                date = date.toRequestBody("text/plain".toMediaTypeOrNull()),
                hour = hour.toRequestBody("text/plain".toMediaTypeOrNull()),
                serviceId = SERVICE_ID.toString().toRequestBody("text/plain".toMediaTypeOrNu
                tableId = tableId.toString().toRequestBody("text/plain".toMediaTypeOrNull()
            )
            // Validar response...
        } catch (e: Exception) {
            // Error handling
        }
    }
}
```

Mejor práctica: Crear extensión helper para evitar repetición:

```
fun String.toRequestBody(): RequestBody =
    this.toRequestBody("text/plain".toMediaTypeOrNull())
```

---

5. Persistencia de Cookies y Expiración

Problema: - OkHttp no persiste cookies entre sesiones si no hay CookieJar - Cookies expiradas no se limpian automáticamente - Usuario queda "atrapado" con sesión inválida

Causa Raíz: - Default MemoryCookieJar de OkHttp solo vive en memoria - No hay validación de timestamp de expiración

Solución:

```
// SessionRepository con EncryptedSharedPreferences
class SessionRepositoryImpl(
    private val encryptedSharedPrefs: SharedPreferences // Encrypted
) : SessionRepository {
```

```
    override suspend fun saveSession(session: Session) {
        encryptedSharedPrefs.edit().apply {
            putString("session_cookie", session.sessionCookie)
            putString("xsrf_token", session.xsrfToken)
            putLong("expires_at", session.expiresAt)
            apply()
        }
    }

    override suspend fun getSession(): Session? {
        val cookie = encryptedSharedPrefs.getString("session_cookie", null) ?: return null
        val expiresAt = encryptedSharedPrefs.getLong("expires_at", 0)

        // Validar expiración
        if (System.currentTimeMillis() > expiresAt) {
            clearSession() // Auto-cleanup
            return null
        }

        return Session(
            sessionCookie = cookie,
            xsrfToken = encryptedSharedPrefs.getString("xsrf_token", null),
            expiresAt = expiresAt
        )
    }
}

// En SessionCookieJar, al guardar
val expiresAt = if (newSessionCookie != null) {
    System.currentTimeMillis() + SESSION_DURATION_MS // 24h por defecto
} else {
    currentSession?.expiresAt ?: (System.currentTimeMillis() + SESSION_DURATION_MS)
}
```

---

6. Comunicación entre Pantallas (Shared Events)

Problema: - Cuando usuario reserva mesa en HomeScreen, MyBookingsScreen debe actualizarse - ViewModels no se comunican directamente (acoplamiento) - SharedPreferences o LiveData global es anti-pattern

Causa Raíz: - Falta mecanismo de eventos desacoplado

Solución:

```
@HiltViewModel
class SharedEventViewModel @Inject constructor() : ViewModel() {
```

```
    val reservationMadeEvent = MutableSharedFlow<Unit>(replay = 0)
    val reloadDataEvent = MutableSharedFlow<Unit>(replay = 0)
    val logoutEvent = MutableSharedFlow<Unit>(replay = 0)

    suspend fun emitReservationMade() = reservationMadeEvent.emit(Unit)
    suspend fun emitReloadData() = reloadDataEvent.emit(Unit)
    suspend fun emitLogout() = logoutEvent.emit(Unit)
}

// En HomeViewModel, tras reserva exitosa
viewModelScope.launch {
    sharedEventViewModel.emitReservationMade()
    // MyBookingsViewModel lo escucha automáticamente
}

// En MyBookingsViewModel
init {
    viewModelScope.launch {
        sharedEventViewModel.reservationMadeEvent.collect {
            loadBookings() // Auto-refresh
        }
    }
}
```

Ventaja: Desacoplamiento total; ViewModels no se conocen entre sí.

---

7. Validación de Cuenta Activa

Problema: - Usuario puede perder sesión TakeASpot pero Firebase aún lo ve loggeado - APIs fallan cuando cuenta TakeASpot es inválida - No hay validación clara antes de hacer API calls

Causa Raíz: - Autenticación dual (Firebase + TakeASpot) sin sincronización

Solución:

```
// UseCase dedicado
class RequireActiveAccountUseCase @Inject constructor(
    private val accountRepository: AccountRepository,
    private val sessionRepository: SessionRepository
) {
    suspend operator fun invoke() {
        val activeAccountId = accountRepository.getActiveAccountId()
            ?: throw IllegalStateException("No active account selected")

        val session = sessionRepository.getSession()
```

```
            ?: throw IllegalStateException("Session expired. Please re-login")
    }
}

// En ViewModels, validar antes de API calls
fun loadBookings() {
    viewModelScope.launch {
        try {
            requireActiveAccount() // Lanza excepción si falla validación
            val result = libraryRepository.getMyBookings()
            // ... handle result
        } catch (e: IllegalStateException) {
            _state.value = MyBookingsState.Error(e.message ?: "Validation error")
        }
    }
}
```

Fortalezas de la Aplicación

1. Arquitectura Clean Architecture

Separación clara de capas: Domain → Data → UI
Testabilidad: Repositorios son interfaces, fácil mock para tests
Mantenibilidad: Cambiar fuente de datos no afecta UI
Escalabilidad: Agregar nuevas features sin modificar capas existentes

Ejemplo: Si quisiéramos usar una API alternativa a TakeASpot, solo reemplazamos `LibraryRepositoryImpl` sin tocar `HomeViewModel` o `HomeScreen`.

---

2. Inyección de Dependencias con Hilt

Eliminación de boilerplate: `@HiltViewModel` auto-configura ViewModels
Construcción automática de gráficos: Hilt resuelve dependencias complejas
Singletons manejados: Firebase, OkHttp, APIs son Singletons sin código manual
Testing simplificado: Usar `HiltTestApplication` para tests

---

3. Moderna Stack de UI con Jetpack Compose

Código declarativo: Menos boilerplate que XML
Recomposición inteligente: Solo actualiza lo necesario

Material Design 3: Tema moderno con soporte dark mode
Hot reload: Cambios en tiempo real durante desarrollo

Comparación con XML:

```
// Compose: 10 líneas
Column(
    modifier = Modifier.fillMaxSize(),
    horizontalAlignment = Alignment.CenterHorizontally
) {
    Text("Mis Reservas", style = MaterialTheme.typography.titleLarge)
    // ...
}

// XML: 30+ líneas de boilerplate
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <TextView
        android:text="Mis Reservas"
        ...
    />
</LinearLayout>
```

---

4. Gestión Robusta de Errores

Result Type: Fuerza manejo explícito de errores
Try-catch estructurado: Mensajes de error claros al usuario
Validación de sesión: Previene API calls con credenciales inválidas

---

5. Seguridad

EncryptedSharedPreferences: Cookies cifradas en almacenamiento local
Firebase Auth: Autenticación segura sin almacenar contraseñas
API level 26+: Requisitos de seguridad modernos (TLS 1.2+, etc.)
ProGuard: Ofuscación en builds release

---

6. Experiencia de Usuario

Multi-cuenta soportada: Cambiar entre diferentes cuentas sin re-login
Persistencia de sesión: Usuarios no re-ingresan tras cerrar app

SwipeRefresh: Actualización manual de listas
Estados claros: Loading, Success, Error bien diferenciados

Debilidades y Limitaciones

1. Acoplamiento a TakeASpot API

Problema: Toda la lógica asume la estructura de TakeASpot
Impacto: Cambios en API requieren refactoring mayor
Evidencia: `SERVICE_ID = 845` hardcodeado; parsing HTML frágil

Mitigación propuesta: - Crear abstracción `BookingServiceProvider` que impl múltiples APIs - Adapters para cada proveedor (TakeASpot, BookingSystem, etc.)

---

2. Parsing HTML con Jsoup

Fragilidad: Si TakeASpot cambia estructura HTML, el app falla
Performance: Parsear HTML es más lento que JSON
Mantenimiento: Selectores CSS hardcodeados sin comentarios

Evidencia:

```
// ¿Qué sucede si TakeASpot cambia "table.bookings tbody tr" a "div.booking-item"?
val bookings = doc.select("table.bookings tbody tr").mapNotNull { row -> ... }
```

Mitigación: - Contactar a TakeASpot para JSON endpoint - Implementar parser más flexible con fallbacks - Tests para validar cambios en estructura

---

3. Sin Testing Automatizado

Sin tests unitarios: ViewModels, Repositories no tienen coverage
Sin tests de integración: API calls no validados contra servidor real
Deuda técnica: Cambios futuros riesgo de regresiones

Estado actual:

```
$ find . -name "*Test.kt" -o -name "*Mock*.kt"
# Solo directorios vacíos
```

Necesario para prod: - Unit tests: `LibraryRepositoryImpl`, ViewModels - Integration tests: `SessionCookieJar`, API mocking - UI tests: Composables críticos

---

4. Sin Manejo de Offline

Sin caché: Si no hay internet, no funciona nada
Sin sincronización: Cambios offline no se guardan
Experiencia degradada: "No connection" es única opción

Solución propuesta: - Room database para caché local - Sincronización diferida (WorkManager) - Indicador de modo offline en UI

---

5. Logging y Debugging

Logs insuficientes: Difícil debuggear en producción
Sin analytics: No sabemos cuál screen falla más
Errores genéricos: "Error al cargar reservas" sin detalles

Mejora:

```
class LibraryRepositoryImpl @Inject constructor(
    private val logger: Logger // Inyectar
) {
    override suspend fun getMyBookings(): Result<List<MyBooking>> {
        return try {
            logger.debug("Fetching bookings for user...")
            val result = api.getMyBookings()
            logger.debug("Successfully fetched ${result.size} bookings")
            Result.success(result)
        } catch (e: Exception) {
            logger.error("Failed to fetch bookings: ${e.message}", e)
            Result.failure(e)
        }
    }
}
```

---

6. Sin Notificaciones Push

Sin recordatorios: Usuario olvida sus reservas
Sin alertas: Cambios en disponibilidad no notificados

Implementación: - Firebase Cloud Messaging (FCM) - WorkManager para tareas periódicas

---

7. UI/UX Limitado

Sin animations: Transiciones entre pantallas muy básicas
Sin gestos: Drag-to-dismiss, swipe actions no implementados

Accesibilidad limitada: Sin soporte para screen readers

Conclusiones

De la Práctica

Esta práctica ha sido un proyecto **end-to-end realista** que integra:

1. **Arquitectura moderna**: Clean Architecture + MVVM con Hilt demostró ser escalable y testeable. El código es ordenado y profesional.
2. **Tecnologías actuales**:
   - Jetpack Compose simplifica UI comparado con XML
   - Hilt reduce boilerplate significativamente
   - Coroutines hacen async code más legible que callbacks
3. **Integración de APIs reales**: Trabajar con TakeASpot (API legacy con HTML) fue desafiante pero educativo. Aprendí:
   - Cómo manejar APIs inconsistentes
   - Importancia de documentación (falta de ella en TakeASpot)
   - Parseo HTML vs JSON
4. **Gestión dual de autenticación**: Firebase + TakeASpot requería solución creativa. El SessionCookieJar y EncryptedSharedPreferences funcionan bien, pero evidencia necesidad de abstracción mejor.
5. **Problemas reales de producción**:
   - Ciclo de vida de sesiones
   - Comunicación entre pantallas
   - Manejo de errores en UI
   - Seguridad (cifrado de credenciales)

De la Asignatura

Positivos: - Enseñanza de arquitectura (Clean Architecture) es relevante y aplicable - Hilt / DI es estándar en Android profesional - Compose es el futuro; aprender es crítico - Proyecto integrador fuerza a pensar en scalability

Mejoras propuestas: - Más énfasis en testing (unit + integration) - Introducir problemas reales (APIs legacy, rate limiting, etc.) - Sessions management y security patterns - Performance optimization (profiling, ANR detection)

Aplicabilidad: Este proyecto es **portfolio-ready**. Demuestra: - Comprensión de arquitectura - Uso correcto de DI - UI moderna con Compose - Integración de APIs reales - Manejo de complejidad (múltiples fuentes de datos)

Vías Futuras

Corto Plazo (1-2 semanas)

1. **Agregar Tests**

```
# Unit tests para HomeViewModel
# Integration tests para LibraryRepositoryImpl
# UI tests para críticas Composables
```

2. **Mejorar Manejo de Errores**
   - Errores específicos por tipo (timeout, auth, network)
   - Retry logic con exponential backoff
   - Error reporting a Firebase Crashlytics
3. **Optimizar Performance**
   - Caché de servicios con TTL
   - Paginación en lista de bookings
   - Lazy loading de detalles

Mediano Plazo (1 mes)

4. **Offline Support**
   - Room database para caché
   - WorkManager para sync diferido
   - Modo offline en UI
5. **Notificaciones**
   - Firebase Cloud Messaging
   - Recordatorios de reservas próximas
   - Alertas de cancelación
6. **Analytics & Logging**
   - Firebase Analytics para eventos de negocio
   - Crashlytics para bug tracking
   - Timber/custom logger para debugging
7. **Mejorar UX**
   - Animations smooth entre screens
   - Gesture support (swipe, drag)
   - Accessibility (TalkBack support)

Largo Plazo (2+ meses)

8. **Soporte Multi-Biblioteca**
   - Abstraer `BookingServiceProvider`
   - Soportar múltiples APIs (BookingSystem, Glide, etc.)
   - Búsqueda cross-provider
9. **Funcionalidades Avanzadas**
   - Historial de reservas
   - Favoritos/bibliotecas guardadas

- Share bookings con amigos
- QR code para entrada

10. **Internacionalización (i18n)**
    - Traducciones (Español, Inglés, Portugués)
    - Soporte regional (timezones, formatos de fecha)
11. **Webversion**
    - PWA con React/Vue
    - Shared backend con app
    - Sincronización en tiempo real (Firebase Realtime DB)

Webgrafía y Referencias

Documentación Oficial

Recurso	URL	Uso
Android Developers	https://developer.android.com	Referencia general Android, APIs
Jetpack Compose	https://developer.android.com/jetpack/compose	UI framework, Composables, state management
Hilt	https://dagger.dev/hilt	Inyección de dependencias
Retrofit	https://square.github.io/retrofit	HTTP client, API definitions
OkHttp	https://square.github.io/okhttp	HTTP client, interceptors, cookies
Firebase Auth	https://firebase.google.com/docs/auth	Autenticación
Firebase Firestore	https://firebase.google.com/docs/firestore	Database (futuro)
Jsoup	https://jsoup.org	HTML parsing
Kotlin Coroutines	https://kotlinlang.org/docs/coroutines-overview.html	Async programming
Room	https://developer.android.com/training/data-storage/room	Database local (futuro)
WorkManager	https://developer.android.com/topic/libraries/architecture/workmanager	Background tasks (futuro)

Artículos y Blogs

Artículo	Autor/Sitio	Tema
Clean Architecture in Android	Robert C. Martin	Architecture patterns

Artículo	Autor/Sitio	Tema
Jetpack Compose State Management	Android Developers	State in Compose
Hilt Dependency Injection	Android Blog	Hilt setup
Handling HTTP Cookies with OkHttp	Square	Cookie management
Kotlin Sealed Classes	Kotlin Docs	Type-safe patterns
Firebase Authentication Best Practices	Firebase Docs	Security

Herramientas y Tecnologías

Herramienta	Versión	Propósito
Android Studio	Hedgehog 2023.1.1+	IDE desarrollo
Gradle	8.x	Build system
GitHub Copilot	-	Asistencia en código (usado extensivamente)
Postman	-	Testing API (TakeASpot endpoints)
Firebase Console	-	Configuración backend
Logcat	-	Debugging

Código Generado/Asistido por IA

GitHub Copilot fue utilizado extensivamente para:

1. **Generación de boilerplate**:
   - Composables básicas (screens)
   - ViewModels con estado
   - DTOs y mappers
2. **Patrones estándar**:
   - Inyección de dependencias
   - Error handling con Result
   - State management
3. **Documentación**:
   - Comentarios explicativos
   - README estructura
   - Docstrings

Ejemplo de uso:

```
Prompt: "Generate a ViewModel with mutableStateOf for loading/success/error states"
Output: HomeViewModel.kt estructura completa en 2 segundos
```

Disclaimer: Copilot proporciona templates; la lógica de negocio y arquitectura fue manual.

Recursos Consultados Ocasionalmente

- **Stack Overflow**: Debugging de errores específicos (KAPT, Hilt, Retrofit)
- **GitHub Issues**: Problemas comunes en librerías
- **Medium**: Artículos sobre arquitectura Android
- **YouTube**: Tutoriales de Jetpack Compose (Google/Philipp Lackner)

Notas Finales

Para Futuros Desarrolladores

1. **Setup inicial**:
   - Clonar repo, configurar `google-services.json`
   - Build con `./gradlew build`
   - Verificar logcat para KAPT errors
2. **Puntos de entrada**:
   - `MainActivity.kt`: Entry point + auth listener
   - `BookingApp.kt`: Hilt initialization
   - `AppNavigation.kt`: BottomNav setup
3. **Workflow típico**:
   - Editar UI en `ui/screens/`
   - Lógica en `ViewModel`
   - API calls en `LibraryRepositoryImpl`
   - Nuevos endpoints en `TakeASpotApi`
4. **Testing**:
   - Emulador: API 30+ (performance)
   - Dispositivo real: Probar cookies y sesiones
   - Diferentes cuentas TakeASpot

Contacto / Créditos

- **Desarrollador**: [Tu Nombre]
- **Universidad**: Universidad Católica de Murcia (UCAM)
- **Asignatura**: [Nombre Asignatura]
- **Fecha**: Enero 2025
- **Versión**: 1.0

Última actualización: 18 de Enero de 2025
Estado: Proyecto finalizado con funcionalidades core implementadas. Mejoras futuras documentadas en Vías Futuras.