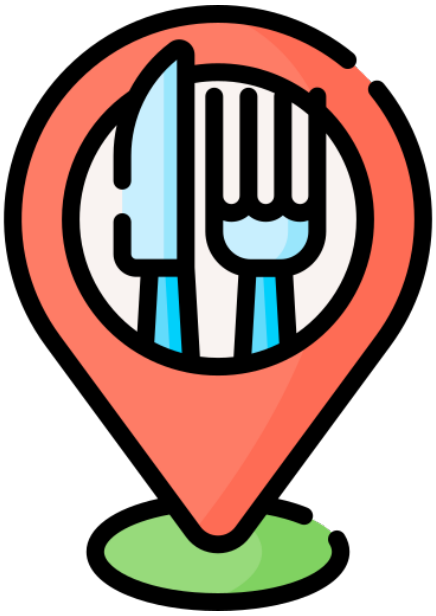

Documentación Técnica:
Desarrollo de apps para Restaurante

Autor: Pedro José Meixús Belsol

16 de noviembre de 2025

Índice general

Capítulo 1

Mayores complicaciones y desafíos

En este documento se detallaran los desafíos, complicaciones y problemas encontrados durante el desarrollo de dos apps para un restaurante, una móvil para que los clientes puedan pedir y una de escritorio para gestionar las mesas y pedidos. Esto, conectado a una API Express que a su vez conecta con una BBDD Mongo Atlas.

1.1. Aplicación móvil

1.1.1. Lenguaje e IDE

Para empezar, he trabajado con Android Studio, siendo una de las primeras veces que lo utilizo, sumado al uso de Kotlin, que aunque lo he usado en una práctica anterior, todavía no tengo la soltura necesaria para poder trabajar comodamente, han hecho que el desarrollo un poco más cuesta arriba.

1.1.2. RecyclerView

Para esta app he utilizado cuatro RecyclerView y aunque ahora ya los manejo un poco más, usa mucha sintaxis nueva y con la que aún me estoy familiarizado.

El que me resultó más complicado fue el que gestiona y guarda los pedidos, que son dos RecyclerView (uno de platos y otro de bebidas) que trabajan con el mismo adaptador y llevan el conteo de cuánto se ha pedido de que.

1.1.3. Retrofit

Esta era la segunda vez que trabajaba con Retrofit (si tenemos en cuenta la practica anterior) y a pesar de no ser tanto en cuanto a líneas de código, es de lo más rompecabezas.

Tuve unos cuantos problemas para conectar Retrofit con Express y que hiciera correctamente las peticiones necesarias. Aquí cabe destacar el único PUT que utilizo, que es el encargado de que cuando ya se ha pedido, si se vuelve a pedir se "sume.[el] pedido ya realizado con lo añadido en el nuevo de manera correcta.

Así es como definí el PUT en la interfaz de Retrofit:

```
@PUT("pedidos/{mesaId}")
suspend fun actualizarPedido(@Path("mesaId") mesaId: Int, @Body
    items: ItemsPayload): Response<Unit>
```

Código 1.1: Definición del PUT en la interfaz de Retrofit

1.2. Aplicación de escritorio

1.2.1. CompletableFuture

Esta fue la primera vez que trabajaba con esta función y me costó un poco entender el concepto y cómo implementarlo correctamente. Aún así me resultó bastante más sencillo de implementar que Retrofit.

Sobretodo lo que mas problemas me generó al principio fue manejar las peticiones asíncronas.

Así es cómo implementé una petición GET para obtener las mesas con pedido:

```
public static CompletableFuture<String> obtenerMesasConPedido() {
    HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create(BASE_URL + "/pedidos/estados"))
            .build();

    return client.sendAsync(request, HttpResponse.BodyHandlers.
        ofString())
            .thenApply(HttpResponse::body);
}
```

Código 1.2: Obtener mesas con pedido usando CompletableFuture

1.2.2. Refrescar estados

Para refrescar los estados de las mesas en la app de escritorio, utilicé un 'scheduleAtFixedRate' que ejecuta una tarea periódica cada cierto intervalo de tiempo. Esta tarea realiza una petición al servidor para obtener el estado actualizado de las mesas y actualiza la interfaz gráfica en consecuencia.

Fue algo sencillo de implementar, pero tuve que asegurarme de que la interfaz gráfica se actualizara correctamente sin bloquear el hilo principal de la aplicación.

Esta es la implementación que hice:

```
    scheduler = Executors.newSingleThreadScheduledExecutor();
    scheduler.scheduleAtFixedRate(this::refrescarEstados, 0, 3, TimeUnit
        .SECONDS);
```

Código 1.3: Refrescar estados de las mesas periódicamente

1.3. API/Express

Respecto a la API el problema es similar a Retrofit, es de las primeras que hago y me llevó más tiempo del esperado. Igual que en Retrofit, lo que más problemas causó fue el PUT y que interactue como debe con la BBDD y con Retrofit.

Esta es la implementación de uno de los PUT que hice en Express para actualizar, este se encarga de modificar un pedido existente:

```
app.put('/pedidos/:mesaId', async (req, res) => {
    const mesaId = parseInt(req.params.mesaId, 10);
    const { items } = req.body;
    if (isNaN(mesaId) || !items || !items.length) {
        return res.status(400).json({ message: 'Faltan datos.' });
    }
    try {
        const pedidosCollection = db.collection('pedidos');
        const pedidoExistente = await pedidosCollection.findOne({ mesaId
            : mesaId, estado: { $in: ['abierto', 'confirmado'] } });
        if (!pedidoExistente) {
            return res.status(404).json({ message: 'No se encontro un
                pedido abierto para actualizar.'});
        }
        await pedidosCollection.updateOne(
            { _id: pedidoExistente._id },
            { $push: { items: { $each: items } }, $set: { estado: '
                abierto' } }
        );
        res.status(200).json({ message: 'Pedido actualizado
            correctamente.' });
    } catch (error) {
        console.error("Error al actualizar pedido:", error);
        res.status(500).json({ message: 'Error al actualizar el pedido'
            });
    }
});
```

Código 1.4: Implementación del PUT en Express

Capítulo 2

Bibliografía

- Retrofit: https://square.github.io/retrofit/
- Corrutinas: https://developer.android.com/topic/libraries/architecture/coroutines?hl=es-419
- Map Kotlin: https://kotlinlang.org/api/core/kotlin-stdlib/kotlin.collections/-map/
- Clases internas: https://kotlinlang.org/docs/nested-classes.html#inner-classes
- RecyclerView: https://developer.android.com/develop/ui/views/layout/recyclerview?hl=es-419
- Data Class: https://kotlinlang.org/docs/data-classes.html
- CompletableFuture: https://www.baeldung.com/java-completablefuture y https://docs.oracle.com
- Scheduler: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledExecutorSe
- GSON: https://mkyong.com/java/how-do-convert-java-object-to-from-json-format-gson-api/
- HashMap: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html