



MusicForBooks

Documentación de proyecto para ASEE Francisco González Vázquez

Índice

MusicForBooks	1
Índice	2
Descripción de la idea	3
Backend de la aplicación	3
Análisis	4
<i>Requisitos funcionales</i>	4
Buscador de la aplicación	4
Vista detallada: libros	4
Añadir una nueva canción	4
<i>Requisitos no funcionales</i>	4
Datos asociados	4
Diseño de la interfaz de usuario	5
<i>Mapa de navegación</i>	5
<i>Pantallas individuales</i>	6
Dashboard	6
Resultados de búsqueda	6
Vista de libro	7
Añadir una canción	7
Implementación	8
<i>Arquitectura de la aplicación</i>	8
<i>Capa de datos</i>	8
Subcapa de datos de caché	8
Subcapa de datos de API	9
Subcapa de repositorios	9
<i>Capa de presentación</i>	10
<i>Inyección de dependencias</i>	11

Nota: El repositorio Git se encuentra alojado en un repositorio privado de GitHub [aquí](#). La API utilizada para el servicio es código libre y se puede consultar en [este](#) repositorio. Se puede descargar el APK desde [aquí](#).

Vídeo de presentación: <https://youtu.be/3HvuiUs5Jrl>

Descripción de la idea

MusicForBooks es una aplicación que permite a los usuarios encontrar música compatible con sus lecturas. Dicha música será añadida por usuarios de la aplicación, haciendo de esta una herramienta colaborativa.

El principal público objetivo son aquellas personas a las que les gusta leer y tener música de fondo, sin embargo también es necesario que dicho público esté familiarizado con las tecnologías para poder moverse con la aplicación. Por tanto la edad objetivo de esta será entre 16 y 40 años, para aquellos que tengan dispositivos Android.

Backend de la aplicación

La relación entre libros y canciones se gestiona a través de una API propia que ha sido desarrollada con Kotlin, el framework Spring y MongoDB como base de datos. La API se ha alojado temporalmente en [Heroku](#).

Nota: Dado que la API está alojada en el plan gratuito de Heroku, la propia plataforma congela el servidor tras 30 minutos de inactividad, haciendo que la primera vez que se haga una petición a la API después de haberse congelado necesite volver a arrancarse el servidor. Por tanto, si la primera vez que se prueba la aplicación esta devuelve un error al cargar tan sólo hay que esperar unos 15-20 segundos a que vuelva a arrancar la API.

Análisis

Requisitos funcionales

Buscador de la aplicación

El usuario podrá buscar por título de libro, nombre de autor, título de canción o nombre de grupo musical. Tras introducir un término y pulsar “Intro” se le mostrará una lista de resultados que, dependiendo de lo buscado será: los libros que coincidan con el título introducido o pertenezcan al autor que se ha escrito; o bien los libros asociados a la canción buscada o asociados a las canciones del grupo que se introducido.

Vista detallada: libros

En esta vista se mostrará por un lado la portada, título y autor del libro; además de la valoración del mismo en la plataforma GoodReads y un enlace a la página del libro en dicha plataforma.

Se mostrará también una lista con el título de la canción y el artista que han añadido otros usuarios a este libro, las cuales se podrán filtrar por puntuación, nombre y artista o si son o no instrumentales. Se mostrará un botón que permita añadir una nueva canción a la lista.

Añadir una nueva canción

En esta vista se mostrará una barra de búsqueda en la cual el usuario podrá introducir el nombre de un artista o canción a buscar en Spotify, que es el que pretende añadir al libro en el cual se encontraba antes de pulsar el botón de añadir canción.

Requisitos no funcionales

- Todas las comunicaciones entre la API REST y el cliente deben utilizar HTTPS.

Datos asociados

Dado que la inmensa mayoría de los datos vamos a recibirlas de APIs externas a la aplicación guardaremos una caché para no tener que hacer las mismas peticiones una y otra vez. La caché consistirá en:

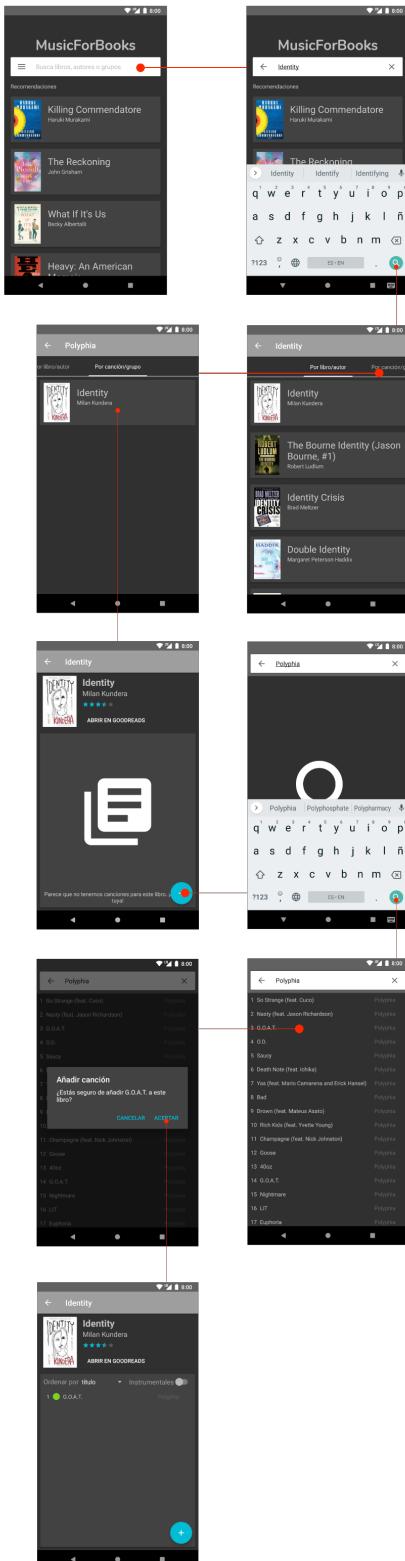
- **Dashboard:** la pantalla principal de la aplicación aloja una lista de libros principales (que se recarga cada mes) como recomendación al usuario. Dado que el tiempo de recarga de los libros es tan grande, es esencial guardar estos datos en caché para no tener que hacer peticiones innecesarias. Estos datos son: nombre del autor (String), nombre del libro (String), ID del libro en GoodReads (Int) y URL de la portada del libro¹ (String)
- **Tokens de Spotify:** para acceder a los datos que nos proporciona la API de Spotify necesitamos un token que se recarga cada hora. Dado que este token puede ser utilizado infinitud de veces en una hora, se guarda en caché para ser reutilizado. Los datos asociados a este token son: el propio token (String) y un tiempo de expiración (Instant)
- **Invalidaciones de caché:** se guardan también los tiempos de invalidaciones de las cachés mencionadas anteriormente. Estos son: tipo de caché (hasta ahora sólo FEED, que pertenece a la dashboard) y tiempo de expiración (Instant)

¹ El contenido de la imagen se cachea también pero por medio de la librería Glide

Diseño de la interfaz de usuario

Mapa de navegación

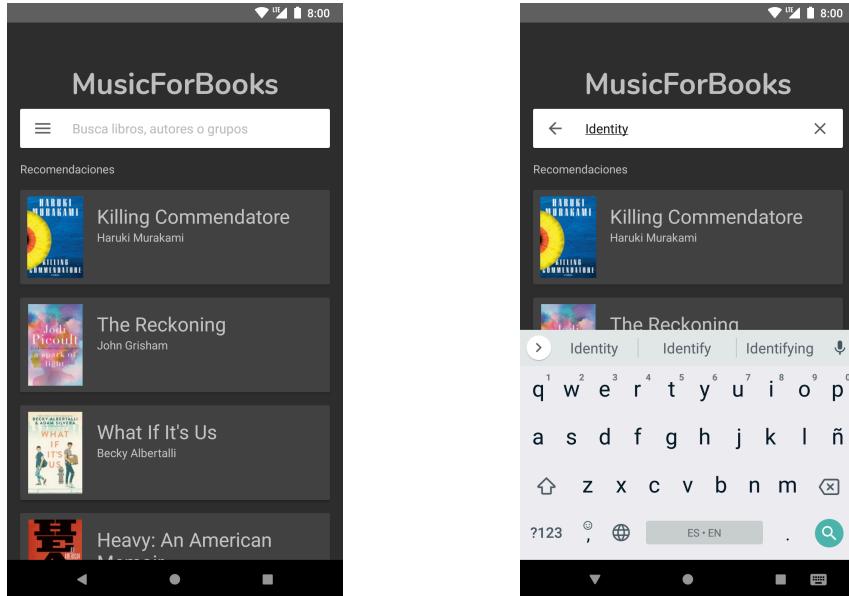
Nota: La imagen a resolución completa se halla en la carpeta *mockups/images* dentro del repositorio junto con el archivo Pixelmator.



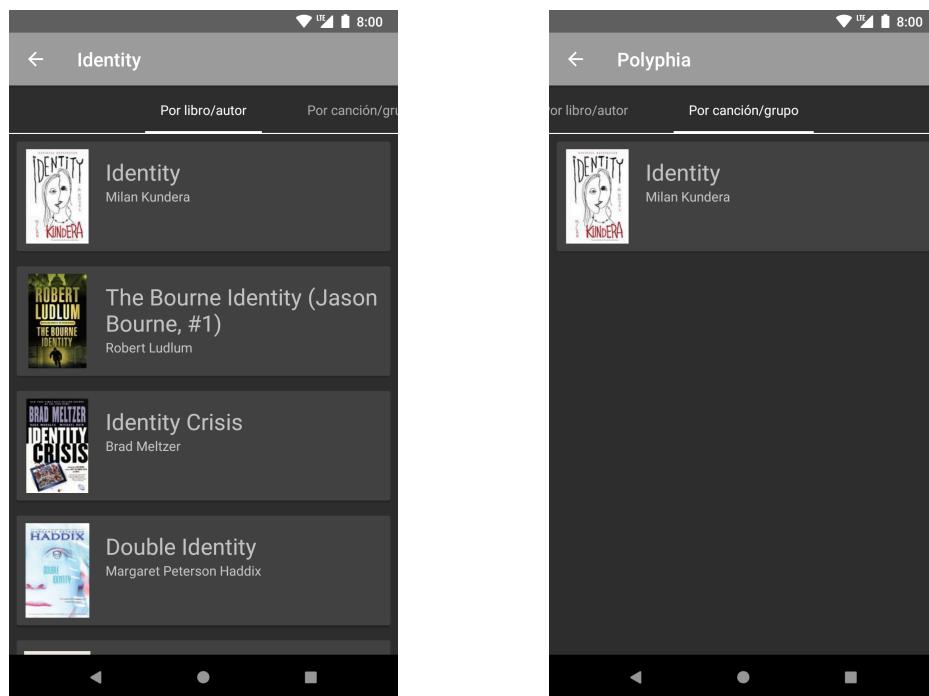
Pantallas individuales

Nota: Todas las pantallas han sido diseñadas con Sketch. Tanto el archivo de Sketch como las distintas imágenes en alta resolución están alojadas en la carpeta *mockups* en el repositorio Git.

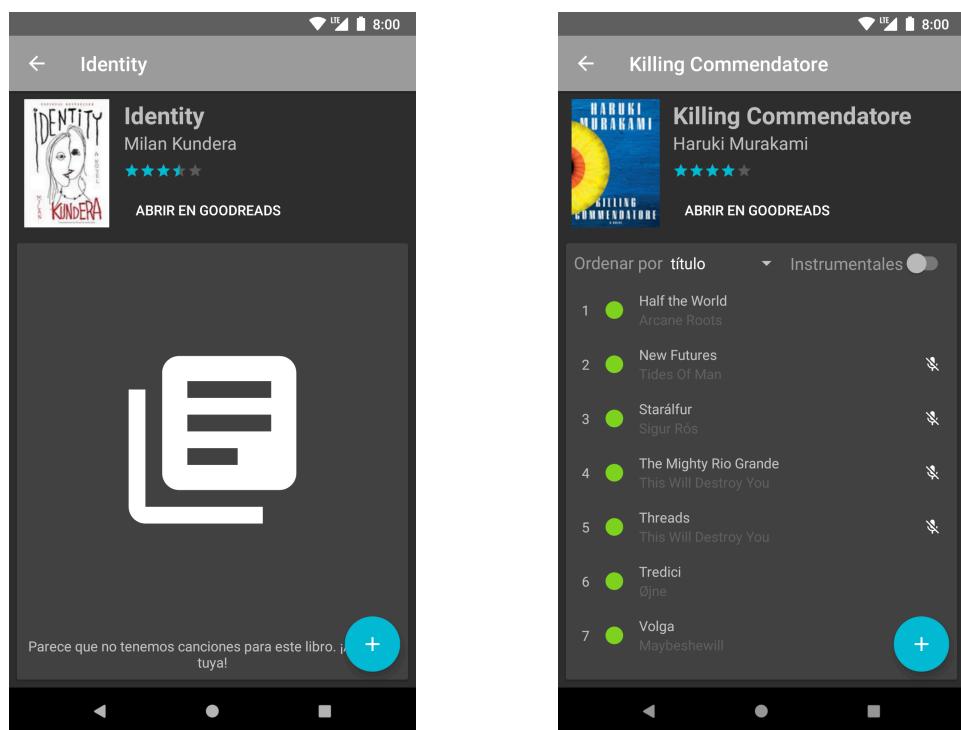
Dashboard



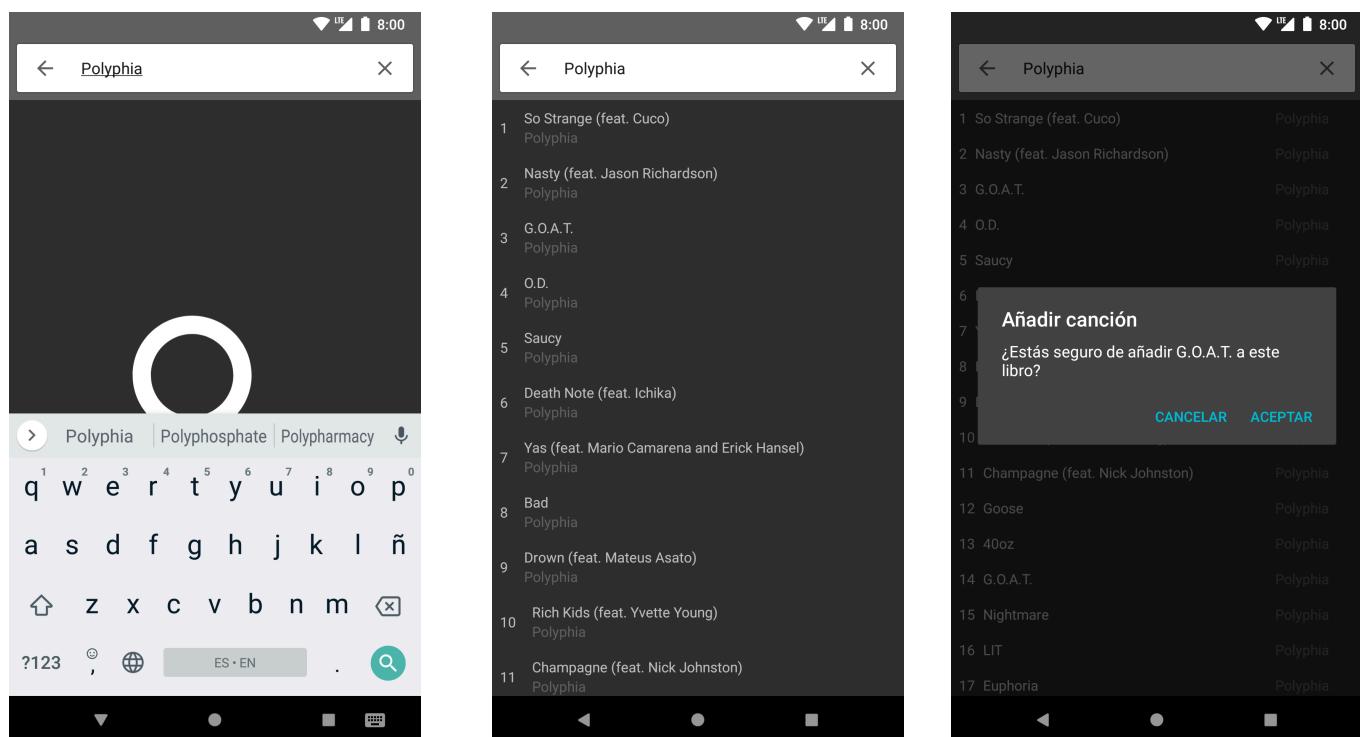
Resultados de búsqueda



Vista de libro



Añadir una canción



Implementación

Arquitectura de la aplicación

Para la organización del código de la aplicación se ha optado por un diseño por capas, en concreto dos principales que se subdividen en varias y las cuales tienen como dependencia únicamente la capa inferior; estas capas se comunican entre ellas a través de *Data Transfer Objects* para así evitar utilizar modelos de datos en capas superiores y no llenar los modelos de datos de anotaciones que no le corresponden.

Las capas creadas son:



Capa de datos

En esta capa se encuentran las operaciones relacionadas con el acceso a datos que, en nuestro caso, proviene de dos fuentes principales: caché, almacenada en una base de datos local, y remota, que se obtiene a partir del uso de la propia API de MusicForBooks, además de dos APIs externas: GoodReads y Spotify.

Esta capa, por tanto, se subdivide en dos: la capa de datos propiamente dicha, que incluye caché y acceso a APIs remotas, y la capa de repositorio que es la utilizada en la capa superior para abstraer el acceso a datos.

Subcapa de datos de caché

En esta subcapa se tiene todo lo relacionado con el acceso y guardado a caché utilizando Android Room para abstraernos de SQLite. Aquí se encuentran, por tanto, las interfaces de acceso a Room (*CacheInvalidationTime*, *FeedCache*, *SpotifyTokenCache*) además de una interfaz extra que se implementa en esta misma capa llamada *CacheStrategy* que determina si un objeto de la caché es o no válido. Se almacenan también los modelos utilizados en estas interfaces

(CacheInvalidationTimelItem, FeedCacheItem, SpotifyTokenCacheItem) y una serie de conversores para Room que permiten almacenar los tipos *Instant* y *CacheType* en la base de datos.

Subcapa de datos de API

En esta subcapa se almacena lo relacionado a la comunicación con servicios externos (la API de la aplicación, GoodReads y Spotify). Para todo esto utilizamos la librería *Retrofit*, que nos permite tan solo declarar una interfaz con el objeto a recibir, objeto a mandar y endpoint a consultar para generar automáticamente la llamada al servicio con todos los datos necesarios. Además pone a nuestra disposición una serie de transformadores que podemos utilizar para convertir los datos recibidos en formatos JSON o XML a un POJO que podemos utilizar en nuestro código. En nuestro caso hemos utilizado *JSON* (usado por la API de la aplicación y Spotify) y *TikXML* para transformar XML (usado por la API de GoodReads).

Tenemos por tanto las interfaces de comunicación con las APIs externas y los modelos asociados a estas con anotaciones de datos para parsear correctamente la respuesta. También se encuentran el *CredentialsProvider*, encargado de encapsular las claves de desarrollo de las APIs que son cargadas en tiempo de compilación, y el *HeadersProvider*, que se encarga de generar cabeceras HTTP para las llamadas de la API.

Subcapa de repositorios

Por último, en esta subcapa se almacena el punto de unión entre la capa de datos y la capa de presentación. Los repositorios abstraen del acceso a datos a la capa superior, de modo que si es necesario obtener un dato de la red porque no se tiene cacheado, esta se encarga de realizar la petición, cachear el resultado y devolverlo a la capa superior.

Por tanto aquí encontramos las interfaces de los repositorios (*BookRepository*, *CacheRepository*, *FeedRepository*, *SearchRepository*, *SongRepository*, *SpotifyTokenRepository*) y sus implementaciones, las cuales llevan el mismo nombre. Se encuentran también los modelos utilizados para esta subcapa, los cuales, a diferencia de las dos anteriores, no tienen anotaciones de datos.

En esta capa, además, se puede ver el uso que se ha hecho en la aplicación de *RxJava* para orquestar el acceso a datos asíncronos de forma sencilla. Dado que todo el acceso a caché y especialmente a datos de la red se debe realizar de forma asíncrona fuera del hilo principal de ejecución para así no bloquear la interfaz de usuario, se ha optado por usar esta librería que nos provee de métodos capaces de redireccionar ciertas funciones a un hilo separado y traer los resultados al hilo principal. También nos provee de cientos de métodos de utilidad traídos directamente de la programación funcional para poder realizar operaciones sobre los datos recuperados. De este modo, por ejemplo, en el *FeedRepository* se realiza la siguiente operación:

```
override fun getFeed(): Single<List<BookItem>> {
    return getCachedFeed()
        .switchIfEmpty(requestFeed())
}
```

- El método devuelve un *Single*, que representa una *promesa* que se notificará al suscriptor de la operación con un sólo valor que será una lista de *BookItem*.
- Ambos métodos *getCachedFeed* y *requestFeed* devuelven el mismo tipo que este método.
- Sin embargo, dado que la caché puede no existir o no ser válida, se utiliza el método *switchIfEmpty* para realizar la petición a la red en caso de que no exista o no sea válida dicha caché.

La operación de `getCachedFeed`, por ejemplo, llama por debajo al `CacheRepository` para realizar esta operación:

```
override fun getCachedFeedItems(): Maybe<List<FeedCacheItem>> {
    return feedCache
        .getAll()
        .flatMapMaybe { items →
            getCacheInvalidationTimeOf(CacheInvalidationTimeItem.CacheType.FEED)
                .switchIfEmpty(
                    Maybe.just(
                        CacheInvalidationTimeItem(
                            cacheType = CacheInvalidationTimeItem.CacheType.FEED,
                            expirationTime = Instant()
                        )
                    )
                )
            )
        }
        .map { Pair(it, items) }
    }
    .flatMap { (invalidationTime, items) →
        if (cacheStrategy.isCacheValid(invalidationTime)) {
            Maybe.just(items)
        } else {
            Maybe.empty()
        }
    }
}
```

La cual:

- Devuelve un `Maybe`, una estructura de datos que define un objeto que puede existir o no, en este caso, si existe, será una lista de `FeedCacheItem`.
- Se obtienen todos los elementos de la caché
- Se obtienen el tiempo de invalidación de la caché del `Feed` la cual, en caso de no existir (`switchIfEmpty`), se establece a una nueva con el tiempo de expiración puesto a este instante, de modo que tendremos que recargar los datos ya que no existe registro anterior de caché
- Se comprueba la validez de los ítems obtenidos de la caché en el `flatMap` para decidir con la `cacheStrategy` si los objetos son o no válidos; en caso de serlo, se devuelven; en caso contrario, se devuelve un `Maybe.empty`, que simboliza que no existe registro de caché de estos elementos

Utilizando estas técnicas de programación funcional podemos dejar de utilizar `null`, que tantos errores puede darnos, además de centrarnos simplemente en las transformaciones que se realizan con los datos.

Capa de presentación

La capa de presentación está directamente acoplada con el framework de Android y utiliza tan sólo las funciones dadas por la subcapa de repositorio. En esta capa tenemos los componentes de la interfaz gráfica de usuario (`Activities`, `Fragments`, etc) y por las clases auxiliares de estas, los `ViewModel`.

Tan sólo los `ViewModel` tienen acceso a la capa de repositorios y los componentes de UI son los encargados de llamar al `ViewModel` para que les devuelva los datos por medio de los llamados `LiveData`.

Todos los `LiveData` de la aplicación tienen asociado un modelo que identifica los distintos estados por los que va pasando la carga de datos de modo que la interfaz gráfica pueda reaccionar a dichos eventos. Por ejemplo, la vista de libro (`BookActivity`) tiene asociados dos modelos distintos: `BookResponse` y `SongResponse`, cada uno para la carga de los datos del libro y de canciones relacionadas con este, respectivamente.

Cuando el *ViewModel* (*BookViewModel*) comienza a cargar los datos le manda al *fragment* el estado *Loading* para que éste pueda mostrar el ícono de carga. Una vez ha finalizado la carga de datos, le notifica un *Success* junto con los datos que han sido cargados o, si ha habido un error, se notifica un *Error* con la excepción que se ha lanzado.

Además tenemos dos componentes que se han reutilizado en varias *activities*: *BookListAdapter* y *SongListAdapter*. Ambos son los *adapters* que deben asociarse con una *RecyclerView* cuando quieran mostrarse una lista de libros o de canciones, de modo que se han creado como componentes aparte para ser reusables.

Inyección de dependencias

A lo largo de todo el proyecto se ha utilizado la librería *Koin* para la inyección de dependencias. Para ello, en *App* se inicializa el grafo de dependencias de la aplicación, extrayendo además las inicializaciones de cada una de las capas inferiores con los archivos *RepositoryProviders*, *CacheProviders* y *ApiProviders*.

Esta librería nos permite declarar las dependencias con un sencillo *DSL*, por ejemplo, así se declaran las dependencias de los repositorios:

```
val module = module {
    factory<FeedRepository>
    { io.fgonzaleva.musicforbooks.data.repositories.FeedRepository() }

    factory<SearchRepository>
    { io.fgonzaleva.musicforbooks.data.repositories.SearchRepository() }

    factory<BookRepository>
    { io.fgonzaleva.musicforbooks.data.repositories.BookRepository() }

    factory<SpotifyTokenRepository>
    { io.fgonzaleva.musicforbooks.data.repositories.SpotifyTokenRepository() }

    factory<SongRepository>
    { io.fgonzaleva.musicforbooks.data.repositories.SongRepository(get(), get(), get()) }

    factory<CacheRepository> {
        io.fgonzaleva.musicforbooks.data.repositories.CacheRepository()
    }
}
```

Una dependencia declarada como *factory* crea una nueva instancia de la misma cada vez que es inyectada en una clase; además, si esta requiere de otras dependencias en el constructor —como es el caso del *SongRepository*— tan sólo hay que anotar el constructor con *get* para que *Koin* las resuelva automáticamente.

Para inyectar las dependencias tenemos dos opciones: la primera consiste en declarar las dependencias en el constructor y, mientras esta clase sea creada por *Koin*, se inyectarán automáticamente tal como he descrito anteriormente; la otra opción es declararlas como miembros de la clase y anotarlas con *by inject*. El único requerimiento es que la clase implemente la interfaz vacía *KoinComponent*:

```
class SearchRepository : SearchRepository, KoinComponent {

    private val musicForBooksService: MusicForBooksService by inject()
    private val goodReadsService: GoodReadsService by inject()
    private val songRepository: SongRepository by inject()
    private val bookRepository: BookRepository by inject()
    private val credentialsProvider: CredentialsProvider by inject()

    ...
}
```