UNIVERSIDAD CARLOS III DE MADRID

GRADO EN INGENIERÍA DE SISTEMAS AUDIOVISUALES

TRABAJO DE FIN DE GRADO

DISEÑO Y DESARROLLO DE UNA APLICACIÓN DE APRENDIZAJE MUSICAL

AUTOR: AITOR ESCOLAR CABEZA

TUTORA: MARÍA CELESTE CAMPO VÁZQUEZ

2018

uc3m | Universidad Carlos III de Madrid

Grado en Ingeniería de Sistemas Audiovisuales
2017-2018

Trabajo Fin de Grado

"Diseño y desarrollo de una aplicación de aprendizaje musical"

Aitor Escolar Cabeza

Tutora

María Celeste Campo Vázquez

Leganés, 8 de octubre de 2018

You sell off the kingdom piece by piece and trade it for a horse that will take you anywhere.

Colin Wright

RESUMEN

La presente Memoria trata sobre el diseño y desarrollo de una aplicación Android destinada al aprendizaje de un instrumento -la guitarra-.

Los guitarristas autodidactas se enfrentan a diversos problemas a la hora de aprender de forma autónoma. Estos problemas no se deben a falta de recursos: hay docenas. Hay docenas de métodos de aprendizaje: algunos muy rígidos; otros demasiado ambiguos. Todos ellos tienen en común que resulta casi imposible no descuidar alguna faceta por muy disciplinado que sea el estudiante.

Mucha gente dedica más de una hora diaria a viajar en transporte público. Para este sector de la población, los smartphones se han convertido en compañeros del día a día, ya que resultan una plataforma muy cómoda para actividades como aprender idiomas, consultar información o consumir contenido multimedia durante los trayectos.

Por otro lado, debido a que los smartphones tienen micrófono integrado, las posibilidades de una aplicación de aprendizaje musical aumentan. En un ordenador, no siempre se dispone de micrófono, o no es accesible.

El objetivo es crear una plataforma con una arquitectura escalable apta para trabajos futuros. Para ello, se ha han utilizado patrones de diseño -Modelo Vista Presentador, Clean Architecture-, reglas de estilo propuestas por Google -Material Design- y tecnologías actuales tanto para desarrollo Frontend -Android Studio, Kotlin- como para Backend -Firebase, Google Drive-.

La aplicación desarrollada abarca: registro y autenticación en un servidor en la nube -Firebase Authentication-; listas de contenidos descargadas desde una base de datos -Firebase Realtime Database-; un reproductor de vídeo desarrollado con ExoPlayer como base; una sección de comentarios de usuarios; y una pantalla de perfil.

Al ser una aplicación móvil basada en la nube, es fundamental optimizar el ancho de banda utilizado, por lo que se ha evaluado y probado la utilización de una tecnología de streaming con bitrate adaptativo, MPEG-DASH (Dynamic Adaptive Streaming over HTTP).

A lo largo de la Memoria, se detalla cómo se han integrado las diferentes partes hasta llegar al estado final de la aplicación. Además, se mostrará cómo desarrollar tests unitarios y de instrumentación para verificar el correcto funcionamiento del código de una forma precisa.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE BLOQUES DE CÓDIGO

1. INTRODUCCIÓN

El propósito de este proyecto no es tanto el de crear un curso de música completo, sino el de diseñar y desarrollar una plataforma que lo permita. Para ello, se ha tratado de combinar una arquitectura robusta, una estructura intuitiva y una filosofía de aprendizaje a la carta. También se han añadido ciertas características sociales, con el objetivo de crear una comunidad que pueda añadir valor personal al contenido de la aplicación.

La aplicación será desarrollada para en la plataforma Android, con el fin de que sea accesible desde dispositivos móviles.

Las aplicaciones del sistema operativo Android son desarrolladas en Java, pero recientemente se incluyó Kotlin como lenguaje oficial, el cual, aporta diversas mejoras respecto al primero.

Firebase es una plataforma creada por Google para ofrecer servicios en la nube que simplifican nel tiempo de desarrollo. La cantidad de tráfico está limitada si se usa de forma gratuita.

En cuanto a la reproducción de video, ExoPlayer es un reproductor multimedia de código abierto desarrollado por Google que soporta muchas más funciones que el reproductor por defecto de Android. Entre otras cosas, permite la reproducción de DASH, un formato de streaming que contiene la fuente de video en múltiples calidades y adapta dinámicamente la calidad de reproducción según varíe la velocidad o estabilidad de la red de datos.

1.1 MOTIVACIÓN

Dos de mis pasiones son la programación y la música. La primera, descubierta durante mi periodo de Formación Profesional. La segunda, lleva acompañándome desde mi infancia y jugó un papel fundamental en que, al elegir Grado, me decantara por Sistemas Audiovisuales.

La motivación detrás de este proyecto es combinar estos factores apoyándome en los conocimientos adquiridos a lo largo del grado y en mi experiencia laboral.

1.2 OBJETIVOS

1.2.1 Objetivos técnicos

El principal objetivo técnico que debe satisfacerse es que la aplicación a desarrollar alcance un estado suficiente de madurez como para que la plataforma propuesta sirva como prueba de concepto.

1.2.2 Objetivos profesionales

Profesionalmente, los objetivos a alcanzar con el presente trabajo son:

- Ser capaz de combinar conocimientos del grado con el desarrollo de software.
- Dominar ExoPlayer y la codificación DASH.
- Adquirir un nivel avanzado con el lenguaje Kotlin.
- Aprender una tecnología de Backend.
- Aplicar una arquitectura limpia para crear aplicaciones grandes y fáciles de testear.

1.3 MARCO REGULADOR

1.3.1 Software de terceros

Si la aplicación desarrollada utiliza recursos o librerías de terceros, los creadores retienen ciertos derechos sobre ellas. Lo habitual en estos casos, es añadir en la aplicación una sección llamada “Software de terceros” o “Licencias de software”, en la que se referencien las librerías utilizadas citando el contenido completo de la licencia que se utilizó en cada una.

En software, la mayoría de las librerías están licenciadas bajo Creative Commons [1] y Apache License 2.0 [2].

1.3.2 Permisos

En caso de que la aplicación acceda a información del dispositivo como la cámara, la agenda, debe solicitar dichos permisos al usuario.

Desde Android 6.0, los permisos que deba aceptar el usuario son solicitados en ventanas, pudiendo optar a permitir sólo los que desee.

1.3.3 Ley de protección de datos

En una aplicación que maneje datos de usuario, es necesario tener en cuenta el Reglamento (UE) 2016/679 del Parlamento Europeo y del Consenso de 27 de abril de 2016 [3] relativo a la protección de las personas físicas en lo que respecta al tratamiento de datos personales y a la libre circulación de estos datos y por el que se deroga la Directiva 95/46/CE (Reglamento general de protección de datos).

Lo cual se traduce en que hay que informar al usuario sobre la información que se va a recaudar, y además, permitirle rechazar que se pueda acceder a sus datos.

Esto también aplica al software de terceros utilizado.

La aplicación desarrollada almacenará ciertas estadísticas de los usuarios además de información como dirección de correo electrónico y nombre, que, además, quedará registrada en una plataforma propiedad de Google.

1.4 ESTRUCTURA DEL DOCUMENTO

El documento está estructurado en capítulos que dividen en diferentes contextos.

- Capítulo 2, **Estado del arte**: se introducen las tecnologías utilizadas y se analizan aplicaciones similares.
- Capítulo 3, **Diseño y Requisitos:** se detallan los requisitos y características que debe cumplir la aplicación.
- Capítulo 4, **Pantallas de la aplicación**: se enfoca en el desarrollo de la parte gráfica y la lógica necesaria para interactuar entre ellas.
- Capítulo 5, **Backend**: trata todo lo relacionado con la configuración e implementación de Firebase y la comunicación entre el cliente móvil y el servidor.
- Capítulo 6, **Arquitectura**: es una explicación, a más bajo nivel, de elementos fundamentales de la estructura de la aplicación. En este capítulo se ven también las librerías utilizadas.
- Capítulo 7, **Testing**: muestra con ejemplos dos formas de probar una aplicación; tests unitarios y de instrumentación.
- Capítulo 8, **Impacto socioeconómico**: se evalúa el presupuesto final del desarrollo del proyecto y se estudian las posibilidades de explotación de la aplicación desarrollada.
- Capítulo 9, **Conclusiones**: compara si la implementación final cumple los objetivos definidos en **Diseño y Requisitos** satisfactoriamente. Además, valora algunas líneas futuras de desarrollo.
- Capítulo 10, **Bibliografía**: condensa todas las referencias consultadas en la escritura de la presente Memoria y enlaces a tecnologías utilizadas.

2. ESTADO DEL ARTE

2.1 SISTEMA OPERATIVO ANDROID

Android es un sistema operativo desarrollado por Google basado en un núcleo Linux. Aunque originalmente se ideó para dispositivos móviles, ha evolucionado mucho desde que se desveló en 2007, llegando ahora a habitar desde relojes hasta televisores pasando por cámaras digitales.

Se pueden desarrollar aplicaciones para Android de varias maneras:

Por un lado, está el desarrollo de aplicaciones **nativas**. En este caso, el entorno consiste en un SDK y un IDE. El SDK proporciona las herramientas para poder compilar, depurar, crear simuladores e instalar distintas versiones de la API. El IDE es el entorno de desarrollo propiamente dicho: proporciona la interfaz para los programadores. Desde 2013, el IDE principal es Android Studio, promocionado por Google.

Un proyecto nativo típico de Android está construido con Gradle y programado en Java o Kotlin [4]; con XML para el diseño de interfaces.

Por otro lado, están las tecnologías **híbridas** como Ionic [5] o React Native [6]. Éstas permiten el desarrollo de aplicaciones multiplataforma mediante JavaScript, lo cual hace que sea más accesible para desarrolladores web.

Hay dos características que convierten a Android en un sistema operativo muy atractivo para los desarrolladores: por un lado, la facilidad en la distribución de las aplicaciones mediante una tienda de aplicaciones (Play Store); por otro lado, la popularidad del sistema, teniendo aproximadamente el 85% [7] de la cuota de mercado de smartphones.

Google actualiza Android con mucha frecuencia, añadiendo mejoras y nuevas funcionalidades. Sin embargo, los dispositivos no se actualizan con la misma frecuencia. Sin ir más lejos, la última versión de la API disponible para desarrollar es la 29, también conocida como Android P o Android 9.0, mientras que más del 90% de dispositivos aún tienen una versión igual o inferior a Android 6.0 (***ilustración 1***).

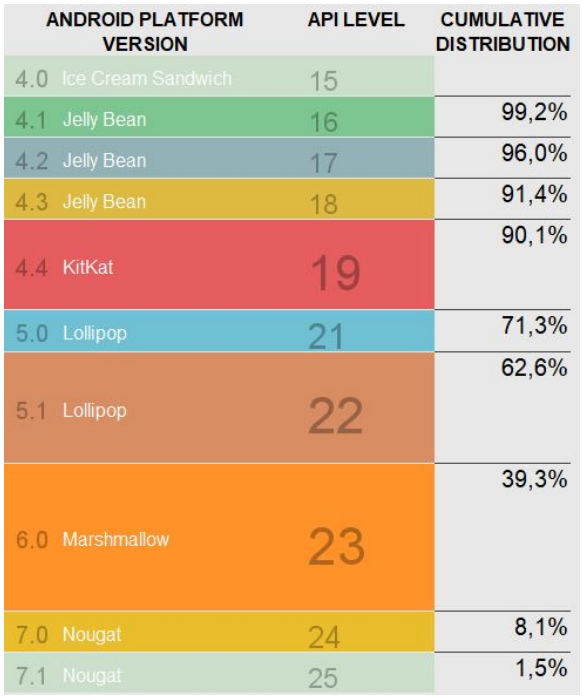


Ilustración 1: Gráfico de ayuda al crear un proyecto extraída del asistente de Android Studio

Por ello, con cada nueva versión, Google también publica librerías de compatibilidad o soporte, que permiten utilizar funcionalidades nuevas en dispositivos antiguos.

El proyecto del que hablará esta memoria será uno **nativo**, desarrollado en **Kotlin**, con API mínima soportada 22 (Android **Lollipop**) y utilización de librerías de soporte.

2.2 KOTLIN

Desde Android Studio 3.0 (octubre de 2017), Google soporta el desarrollo en Kotlin para Android. Poco después, lo anunciaron como lenguaje oficial.

Kotlin, desarrollado por JetBrains [8], es un lenguaje estáticamente tipado que puede funcionar tanto en la JVM como compilarse a JavaScript. Es totalmente interoperable con Java, lo que permite que proyectos con grandes bases de código desarrollado en Java puedan mezclarse sin ningún problema con nuevas funcionalidades creadas en Kotlin.

Kotlin permite un desarrollo mucho más expresivo que Java, eliminando gran parte de su verbosidad. Es un lenguaje con muchas ventajas respecto a Java y ningún inconveniente. Como principal ventaja, que aplica especialmente a Android, es *null safe*[1] (seguro frente a nulos), y permite una sintaxis mucho más limpia cuando se trabaja con nulos.

[1] Seguro frente a nulos. Una causa común de error en aplicaciones Android desarrolladas en Java.

```
Object objetoNulo = null;
objetoNulo.toString(); // NullPointerException
if(objetoNulo != null) { // Null check
    objetoNulo.toString();
}
```

Código 1: NullPointerException y null check en Java

```
var objetoNoNulo: Object = null // Error en tiempo de compilación
var objetoNulo: Object? = null // El carácter '?' indica que puede ser
nulo
objetoNulo.toString() // Error en tiempo de compilación
objetoNulo!!.toString() // NullPointerException en tiempo de ejecución
objetoNulo?.toString() // Llamada null safe
```

Código 2: Kotlin es null safe

La doble exclamación sirve para decir al compilador que algo que puede ser nulo, no lo es. Es recomendable evitarlo en la medida de lo posible, pero hay algunas excepciones en las que se debe utilizar, como se verá más adelante.

La llamada null safe de Kotlin es especialmente útil en casos en los que hay varias llamadas anidadas que pueden ser nulas:

```
if(objeto != null && objeto.get() != null && objeto.get().get() != null) {
    objeto.get().get().metodo();
}
```

Código 3: Posibles nulls anidados en Java

```
objeto?.get()?.get()?.metodo()
```

Código 4: Posibles nulls anidados en Kotlin

Otras técnicas interesantes para trabajar con nulos son el "operador Elvis" (?:), la función 'let' y el hecho de que en Kotlin las estructuras de control condicionales son expresiones que retornan valor.

```
String cadena = null;
if(condicion) {
    cadena = "...";
} else {
    cadena = "";
}
cadena = null; // Se puede eliminar una referencia a posteriori sin ningún
tipo de problema
```

Código 5: Asignación de valores a un objeto tipo String en Java

```
val cadena = if(condicion) "..." else "" // Asignando en un if
cadena = null // Error en tiempo de compilación, 'val' es inmutable
val elvisOperator = posibleObjetoNulo?.toString() ?: "..." // si el objeto
es nulo, el valor de la variable será "..."
posibleObjetoNulo?.let {
  // ejecuta código si posibleObjetoNulo no es null
}
```

Código 6: Ejemplos de if como expresión, Elvis Operator y let

Como se puede ver en los bloques de código, especificar el tipo de una variable es opcional si se le asigna un valor inmediatamente, pero esto no se debe a que sea débilmente tipado como JavaScript, sino a que se infiere el tipo en tiempo de compilación.

```
var cadena = "string" // Equivale a var cadena: String = "string"
cadena = 3 // Error en tiempo de compilación, cadena es de tipo String
```

Código 7: El tipo es inferido y no se puede cambiar a posteriori

A lo largo de este proyecto se han utilizado numerosas funcionalidades de este lenguaje al mismo tiempo que se demuestra la interoperabilidad con Java, ya que los tests unitarios y de instrumentación están desarrollados en este último, pero desde este momento se obviarán las comparaciones con Java a no ser que se requiera.

2.3 Introducción al desarrollo en Android

Las diferentes pantallas de una aplicación Android se denominan *Activity* [9]. Éstas tienen un ciclo de vida particular en el que suceden eventos según la pantalla sea creada, pausada, rotada, llevada a segundo plano, eliminada, restaurada, etc.

En la ***ilustración 2*** puede verse el diagrama del ciclo de vida de una *Activity*.

Una *Activity* puede hacer referencia a un *layout*[2] que describa los elementos visuales y su ubicación en pantalla.

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

Código 8: Añadiendo un layout en una Activity

[2] Agrupación de vistas definidas en un fichero XML (Extensible Markup Language)

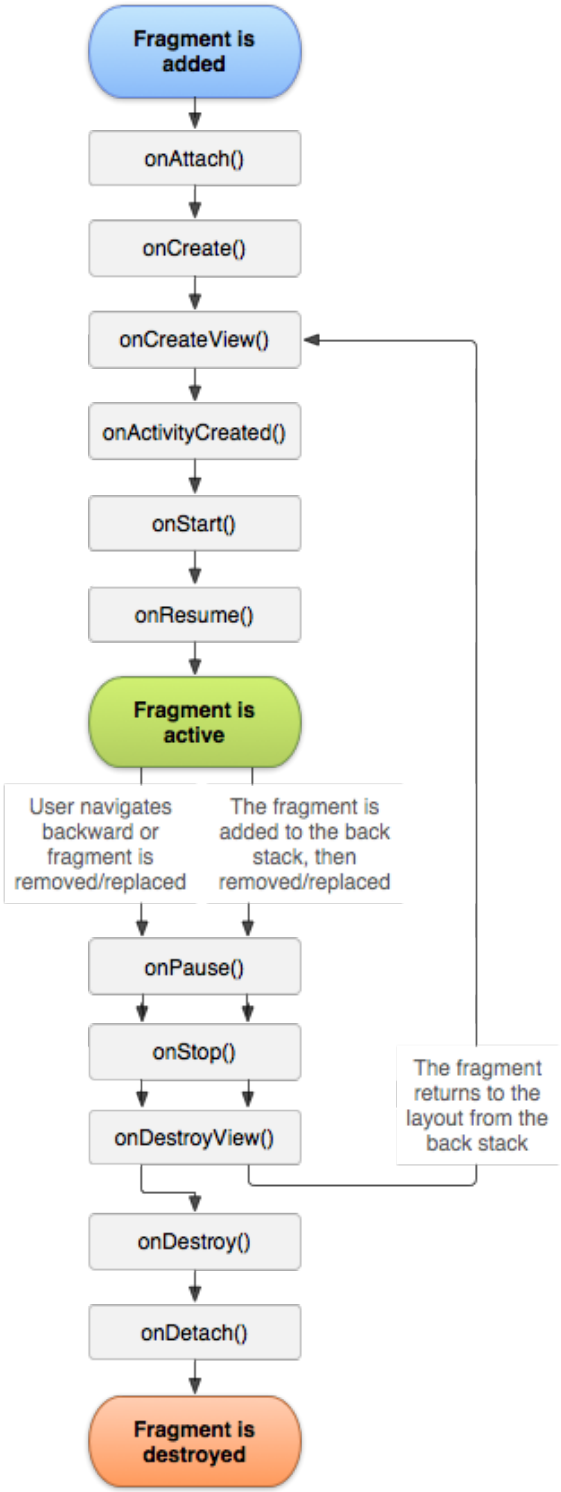


Ilustración 2: Ciclo de vida de una Activity *[9]*

En una vista puede haber componentes como texto, cajas de formulario, botones, listas, imágenes, barras de progreso, barras de herramientas, etcétera. Todos los componentes extienden de la clase *View*, por lo que, además, es posible crear componentes personalizados si heredan de ésta.

En los componentes *View*, ciertos parámetros se pueden especificar desde el XML, por ejemplo, el identificador, el ancho y la altura.

```
<TextView
    android:id="@+id/text_view"
    android:layout_width="0dp"
    android:layout_height="0dp"
    />
```

Código 9: Componente de texto en el XML

Desde la *Activity*, los componentes del XML pueden ser obtenidos como objetos a partir del identificador para manipular sus atributos:

```
val mTextView = findViewById<TextView>(R.id.text_view)
mTextView.text = "texto"
```

Código 10: Obteniendo el componente de texto como objeto para añadirle texto

También, existen componentes llamados *Fragment* [10], definidos como componentes que definen un comportamiento concreto dentro de una *Activity* con su propio ciclo de vida. Al igual que las *Activity*, pueden estar asociados con un fichero XML. Entre sus utilidades, están la de agrupar distintos componentes *View* para poder reutilizarlos en varias pantallas, reemplazarlos dentro de una misma *Activity* o hacer interfaces que se vean diferentes en teléfono y tablet (véase Gmail).

Google especifica que cada sección con un contexto debería ser una *Activity*, mientras que secciones o módulos de una sección pueden ser representadas por *Fragment*.

Otro elemento importante en el desarrollo Android son los *Intents* [11]. Estos son objetos que se utilizan para solicitar acciones como comenzar otra *Activity*, iniciar un servicio o entregar un mensaje a otro componente o aplicación.

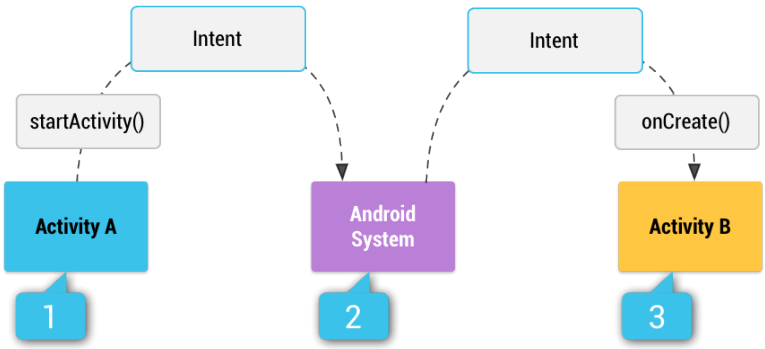


Ilustración 3: Flujo de un intent para comenzar otra actividad [11].

2.4 FIREBASE

Firebase [12] es una plataforma de desarrollo que ofrece numerosos servicios al desarrollo web y móvil. Fue creada por Firebase, Inc en 2011 y adquirida por Google en 2014.

En este proyecto, se ha utilizado de forma extensiva, ya que ofrece una forma rápida y sencilla de crear un backend. Algunas de sus funciones más destacadas son:

- **Firebase authentication**: provee servicios backend y librerías para autenticar usuarios en la aplicación a través de Facebook, Twitter, Google, SMS y correo electrónico.
- **Firebase realtime database**: base de datos no relacional que utiliza un formato similar a *json*[3]. Permite configurar permisos avanzados.
- Cloud messaging: plataforma de mensajes multiplataforma para notificar a los usuarios de forma sencilla.
- Analytics: aplicación que informa sobre detalles de uso y se puede utilizar para captar mejor a los usuarios.
- **Storage**: almacenamiento de objetos como texto, imágenes y video.
- **Crashlytics**: reporte de crashes en tiempo real que permite hacer un seguimiento de problemas de estabilidad en la aplicación.

En negrita, las funciones utilizadas por la aplicación desarrollada. Tiene muchas más funciones y van añadiendo nuevas regularmente. Por ejemplo, recientemente publicaron la beta de Cloud Firestore, otra base de datos no relacional más escalable que Realtime Database. La lista completa puede consultarse en la página principal de Firebase.

2.5 CONTROL DE VERSIONES

Los sistemas de control de versiones (SCV), pertenecen a una categoría de software que permiten gestionar el código a lo largo del tiempo.

Un VCS registra cada cambio en el código y permite al usuario o equipo guardar dichos cambios y hacerlos públicos (llevarlos al repositorio base o remoto).

Permiten deshacer cambios y volver a un punto anterior en caso de cometer errores.

En este proyecto, se ha hecho uso de Mercurial, mediante la herramienta TortoiseHg.

[3] Formato de texto consistente en una colección de elementos clave-valor utilizado para transmitir información

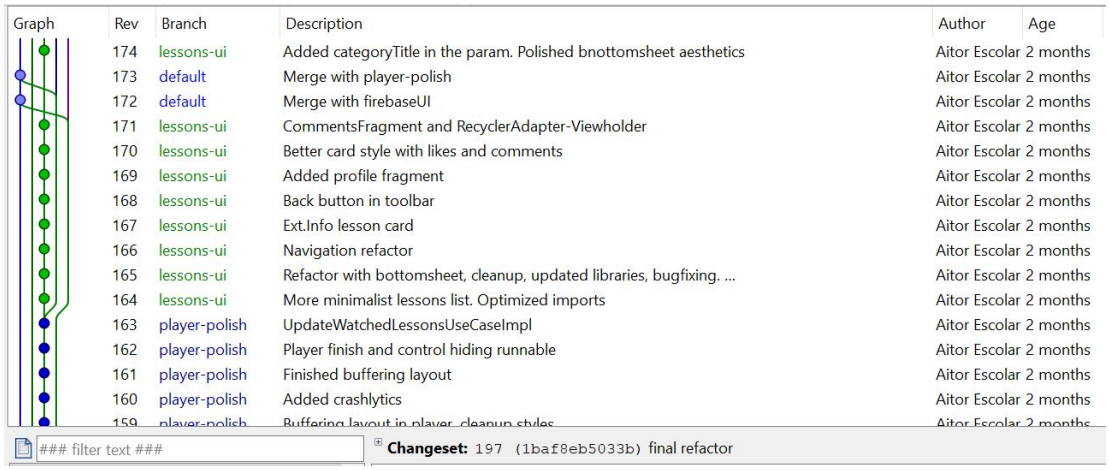


Ilustración 4: Vista de cambios guardados en TortoiseHg

Como el desarrollo se ha llevado a cabo tanto en un ordenador de mesa como en uno portátil, se ha utilizado un repositorio remoto al que subir los cambios para tener una fuente común. La opción elegida ha sido Bitbucket.

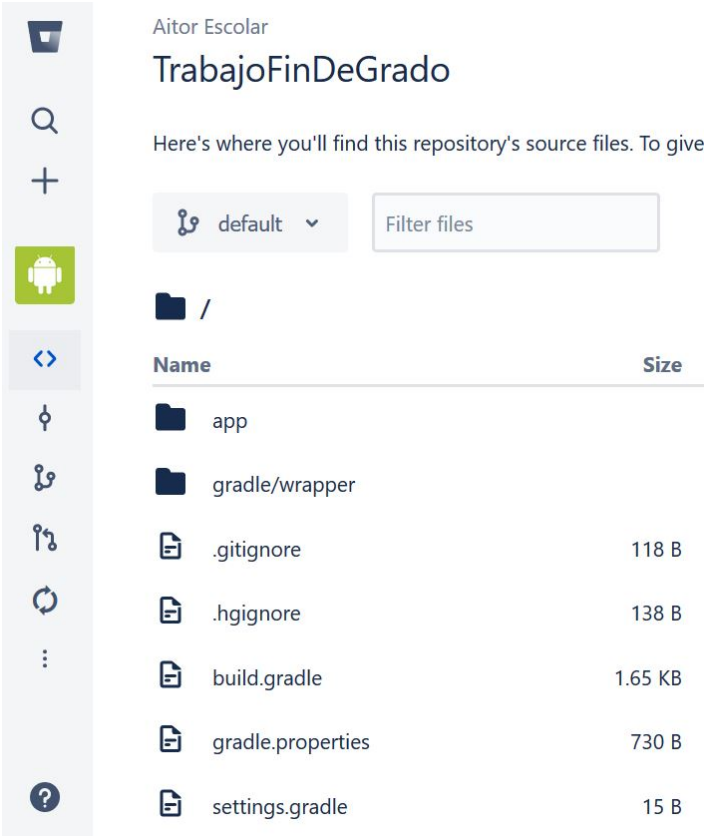


Ilustración 5: Repositorio remoto del proyecto

El repositorio del proyecto es público, y puede consultarse en la siguiente dirección:

https://bitbucket.org/aescolar/trabajofindegrado/

2.6 OTRAS APLICACIONES DE APRENDIZAJE

En cuanto a alternativas para el aprendizaje, hay tres ejemplos de aplicaciones con las que este proyecto puede compararse. Los puntos fuertes de cada una de las tres han servido de inspiración en la idea final de la aplicación desarrollada.

2.6.1 Geared [13]

Plataforma web con las lecciones ordenadas por categorías. En cada lección hay un video y una descripción del contenido. No cuenta con aplicación móvil aún.

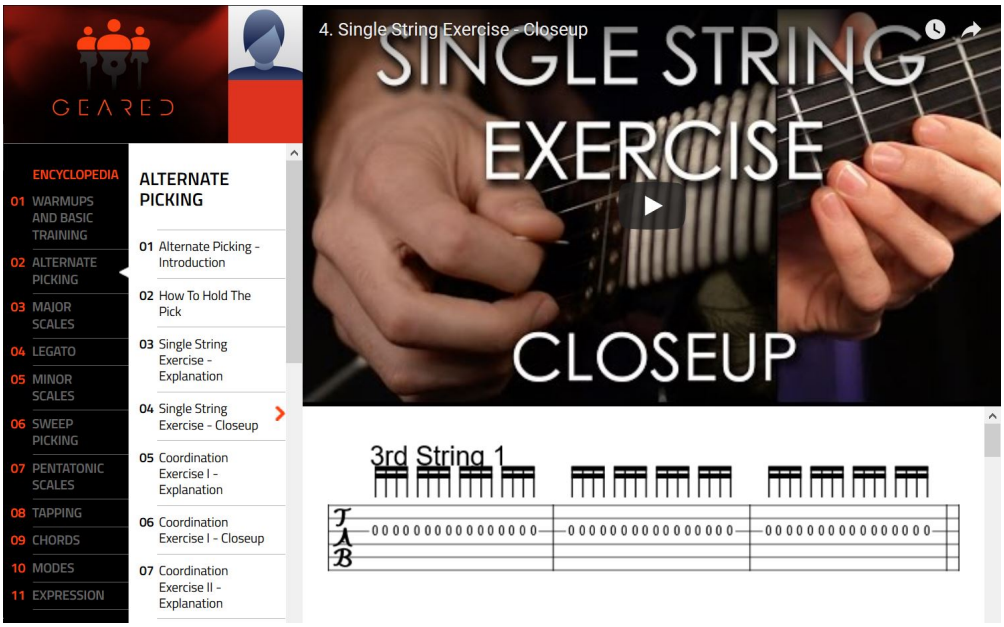


Ilustración 6: Vista de lección en Geared

Su formato es muy apropiado para la **pantalla de un ordenador** y tiene gran variedad de **contenidos bien ordenados**.

2.6.2 SoloLearn [14]

Aplicación gratuita para aprender lenguajes de programación. Cada lección tiene una **sección de comentarios** para que la propia comunidad pueda aportar valor y feedback.

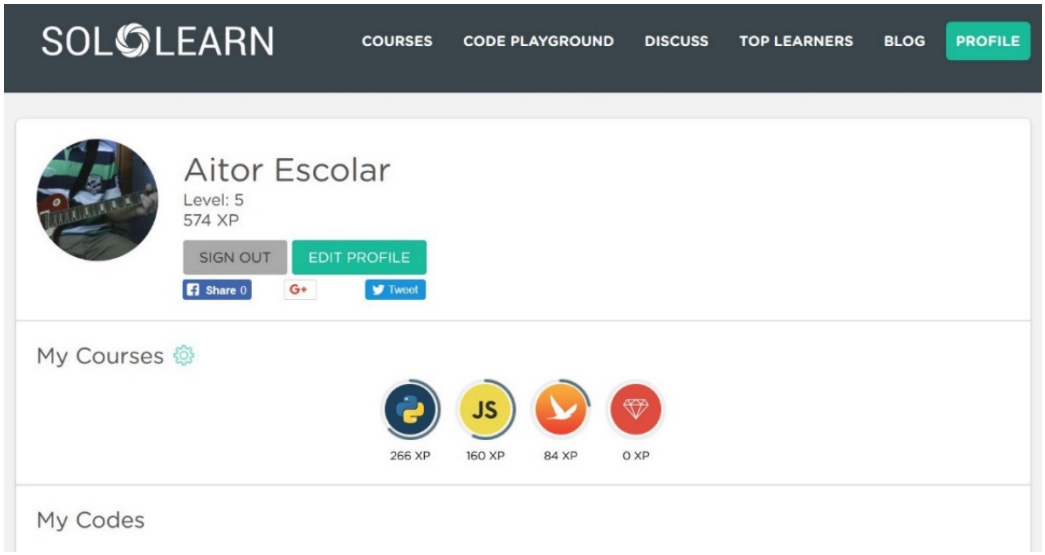


Ilustración 7: Página de perfil

La sección de perfil muestra progresos y logros para **ludificar** el aprendizaje.

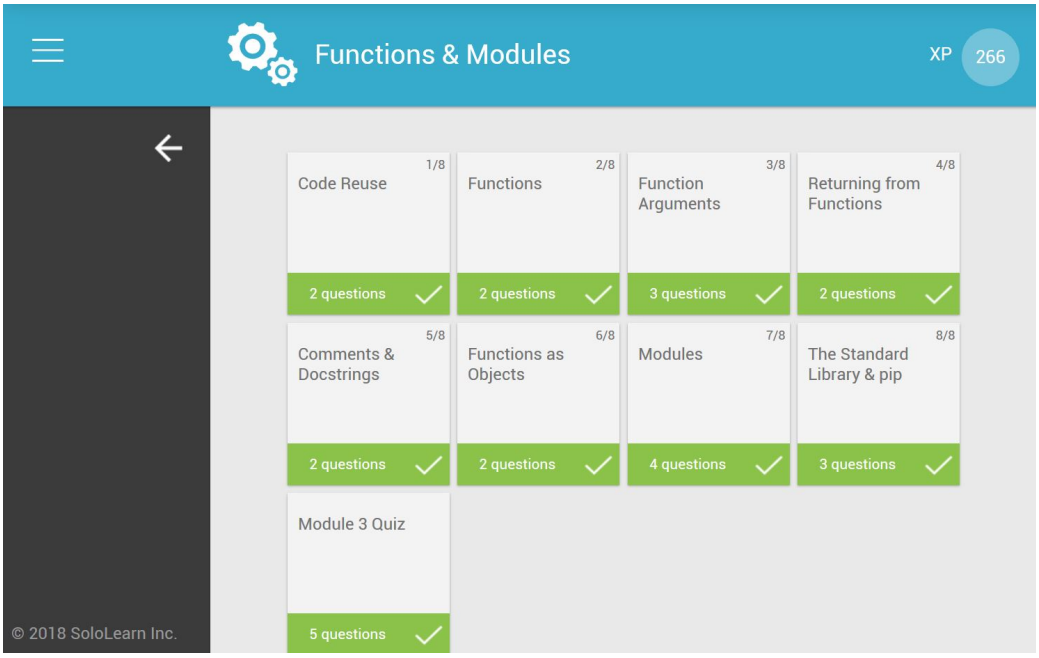


Ilustración 8: Vista de lecciones en tarjetas

La interfaz sigue las pautas de Material Design [15].

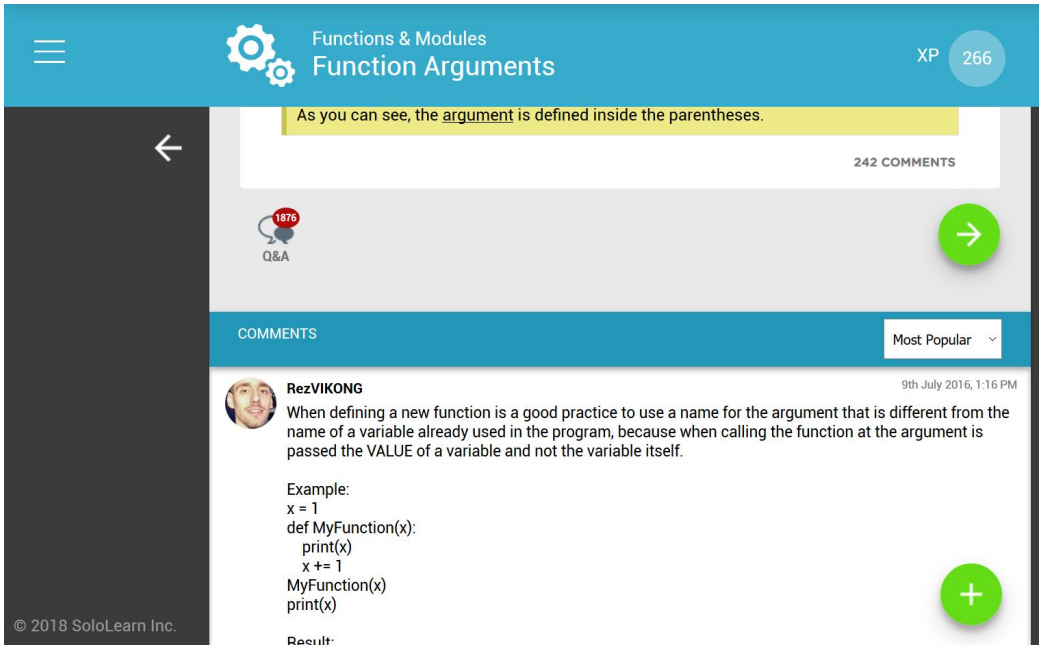


Ilustración 9: Vista de comentarios en una lección

El contenido se complementa con **sección de comentarios** en cada lección en la que los usuarios pueden preguntar y responder dudas o aportar **información adicional**.

La aplicación es muy completa y está disponible para todas las plataformas. Además, **permite a los usuarios crear contenido**.

2.6.3 Rocksmith 2014 [16]

Videojuego de Ubisoft para aprender a tocar la guitarra. La guitarra se conecta mediante una interfaz jack-usb y posee un potente algoritmo de **detección de notas y acordes**. Tiene distintos modos y minijuegos para aprender a dominar el instrumento, además de permitir la compra de packs de canciones para aprender a tocarlas desde el juego. Su filosofía de aprendizaje se basa en un **aumento gradual de la dificultad** según el jugador mejora su precisión.

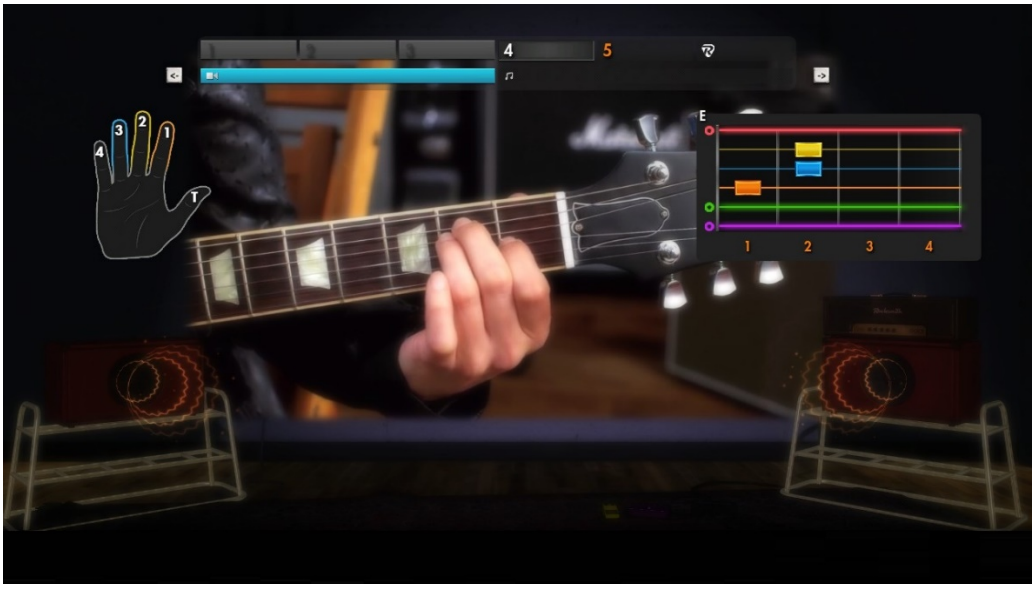


Ilustración 10: Lección teórica en Rocksmith

En las lecciones, el aprendizaje se apoya en un video de fondo y una imagen del mástil con las posiciones.

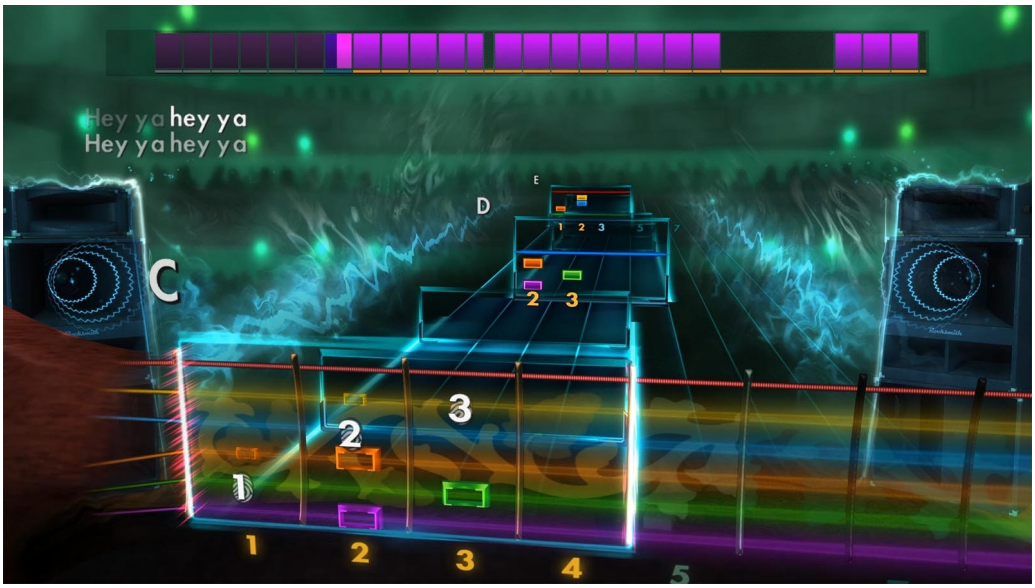


Ilustración 11: Aprendiendo a tocar una canción

En una partida, se pueden ver las posiciones de la mano izquierda en el tiempo según avanza la canción. En el momento que las notas o el acorde pasan por el mástil, el jugador debe ejecutar dicha posición con la mano derecha. Si la posición no era correcta o no se realizó a tiempo, se indicará el error.

Como inconvenientes, **sólo está disponible para PC y consolas** y **no dispone de comunidad**. Además, es **de pago** y los packs de canciones implican desembolso adicional.

2.7 CONCLUSIONES

En resumen, la siguiente tabla muestra los puntos fuertes y débiles de las aplicaciones comparadas:

Aplicación	Puntos positivos	Puntos negativos
Geared	• Información bien ordenada • Contenido abundante	• No tiene comunidad • No apta para dispositivos móviles
SoloLearn	• Metodología de aprendizaje muy intuitiva • Comunidad de usuarios • Disponible desde móviles	• Su ámbito es distinto al del campo de estudio en este proyecto (programación)
Rocksmith 2014	• Algoritmo de detección de acordes • Centrada en la práctica	• Sólo para plataformas de sobremesa • Sin comunidad de usuarios

El objetivo del proyecto es utilizar las tecnologías más actuales para desarrollar una aplicación que combine las fortalezas de las anteriormente expuestas.

3. DISEÑO Y REQUISITOS

3.1 REQUISITOS FUNCIONALES

Para que la aplicación logre el objetivo de enseñar a tocar un instrumento, se han definido varios requisitos esenciales. Estos requisitos son:

- Sección de **lecciones** ordenadas por categorías y dificultad. No se bloquea ningún contenido con requisitos previos para no bloquear el progreso, pero se muestran en un orden lógico.
- Cada lección debe tener una explicación teórica y un video demostrativo.
- Es necesario crear un **reproductor de video** para poder ver dichas lecciones.
- El reproductor de video debe tener controles táctiles para silenciar, adelantar/retrasar diez segundos, pausar/reanudar y navegar a una posición del video deslizando el dedo en la barra de progreso.
- Los usuarios tienen que tener la posibilidad de opinar en cada contenido. Esto permite crear una **comunidad** que estimula aún más el aprendizaje.
- Sección de **práctica** para practicar lo aprendido. Mediante el micrófono del dispositivo, se ofrecerá feedback al usuario sobre la ejecución. Idealmente, la práctica será planteada como un videojuego en el que las pruebas se generen proceduralmente.
- Sección de **perfil** desde la que el usuario podrá ver estadísticas personales, como lecciones completadas de cada categoría y progreso en las diferentes técnicas.
- Gestión de usuarios y autenticación para poder **guardar el progreso de cada usuario**.

3.2 REQUISITOS TÉCNICOS

Para que la estructura del código tenga una estructura intuitiva y no se vuelva excesivamente complejo o difícil de mantener al escalar en magnitud, los requisitos serán:

- Usar los principios de arquitectura limpia.
- Basarse en un patrón de diseño arquitectural.
- Recurrir a tecnologías de streaming adaptativo para optimizar el ancho de banda consumido por parte del usuario.

- En cuanto a la interfaz de usuario, utilizar las reglas de estilo definidas por Material Design.
- Implementar tests que aseguren el correcto funcionamiento del código a largo plazo.

4. PANTALLAS DE LA APLICACIÓN

La aplicación consta de tres *Activities*, siendo una de ellas proporcionada por FirebaseUI-Android.

- *AuthUI* proporcionada por la librería FirebaseUI-Android [17], se crea automáticamente mediante un *Builder* al que se puede proporcionar un estilo y una lista de opciones de inicio de sesión (Facebook, Google, Twitter, correo electrónico o teléfono). Es utilizada para el proceso de registro e inicio de sesión tanto por cuenta de Google como por correo electrónico. **4.1**.
- *ApplicationActivity* con multiples *Fragments* que sigue el patrón *Single Pattern Activity*[4] [18]. Tiene orientación vertical, *Toolbar* y *BottomNavigationView*. **Sección 4.2**.
- *PlayerActivity* y otra con orientación horizontal, destinada al reproductor de vídeo y controles táctiles de las lecciones, vista en la **sección 4.3**.

Cada actividad hay que añadirla a un fichero llamado *AndroidManifest*. En este fichero se especifican también parámetros como permisos requeridos por la apliación, los temas y orientación de cada actividad o la actividad principal.

[4] Las preferencias de estilo o configuraciones y los elementos compartidos como barras de navegación superior o inferior se ubican en la *Activity*, mientras que el contenido de cada pantalla se encuentra en un *Fragment*.

```
<activity
    android:name=".activities.application.ApplicationActivity"
    android:label="@string/app_name"
    android:screenOrientation="portrait"
    android:windowSoftInputMode="adjustPan"
    android:theme="@style/AppTheme.NoActionBar">

    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>

<activity
    android:name=".activities.player.PlayerActivity"
    android:configChanges="keyboardHidden|orientation|screenSize"
    android:label="@string/title_activity_player"
    android:screenOrientation="landscape"
    android:theme="@style/PlayerTheme" />
```

Código 11: Declaración de actividades en el Manifest

En el fragmento de **código 11**, se puede ver que no es necesario añadir la actividad de autenticación, y que *ApplicationActivity* requiere un *intent-filter* para especificar que es la principal y el punto de arranque.

En un ciclo de vida de la aplicación, antes de crear la vista de *ApplicationActivity*, se comprueba si existe un usuario logado. En caso de no haberlo, se lanza la actividad *AuthUI*.

4.1 AUTHUI

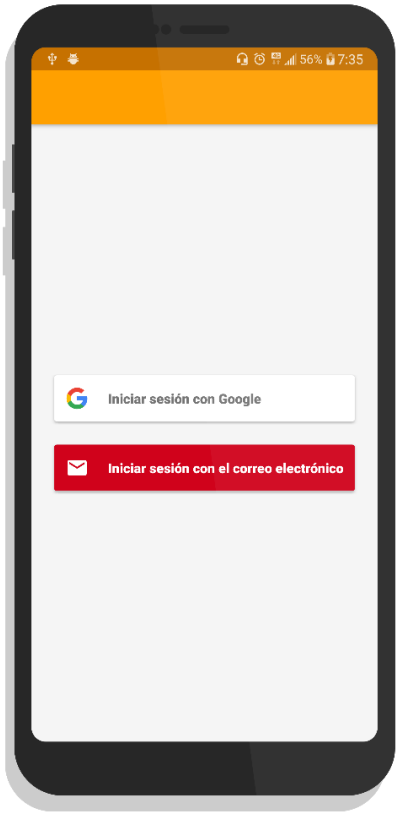


Ilustración 12: AuthActivity

Dentro de *ApplicationActivity*, en el método *onCreate()*, se ha añadido un *listener* a la instancia de *FirebaseAuth* que escuche el estado y sea llamado cada vez que cambie. Esto sucede ante un inicio de sesión o al salir de la cuenta.

```
obtainStoredUserUseCase = Injection.provideObtainFirebaseUserUseCase()

authStateListener = FirebaseAuth.AuthStateListener {

    userLoggedIn = obtainStoredUserUseCase.userExists()
    if (!userLoggedIn) {
        val providers = listOf(
                AuthUI.IdpConfig.GoogleBuilder().build(),
                AuthUI.IdpConfig.EmailBuilder().build())

        startActivityForResult(
                AuthUI.getInstance()
                        .createSignInIntentBuilder()
                        .setIsSmartLockEnabled(false)
                        .setTheme(R.style.FirebaseTheme)
                        .setAvailableProviders(providers)
                        .build(),
                RC_SIGN_IN)
    }
}
```

Código 12: Detectando si hay usuario autenticado

Al utilizar el método *startActivityForResult*, debemos también sobrescribir el método *onActivityResult*, que será llamado cuando finalice la actividad de autenticación.

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data:
Intent?) {
    super.onActivityResult(requestCode, resultCode, data)

    if (requestCode == RC_SIGN_IN) {
        if (resultCode == Activity.RESULT_OK) {
            navigate(CategorySelectFragment.TAG)
        } else {
            finish()
        }
    }
}
```

Código 13: Volviendo de la actividad de inicio de sesión

El primer argumento, requestCode, identifica el *intent*, por lo que se compara que el código sea el mismo que con el que se inició la actividad (*RC_SIGN_IN*); después, se comprueba que el resultado, resultCode, es correcto para empezar la navegación, en caso contrario, se finaliza la aplicación.

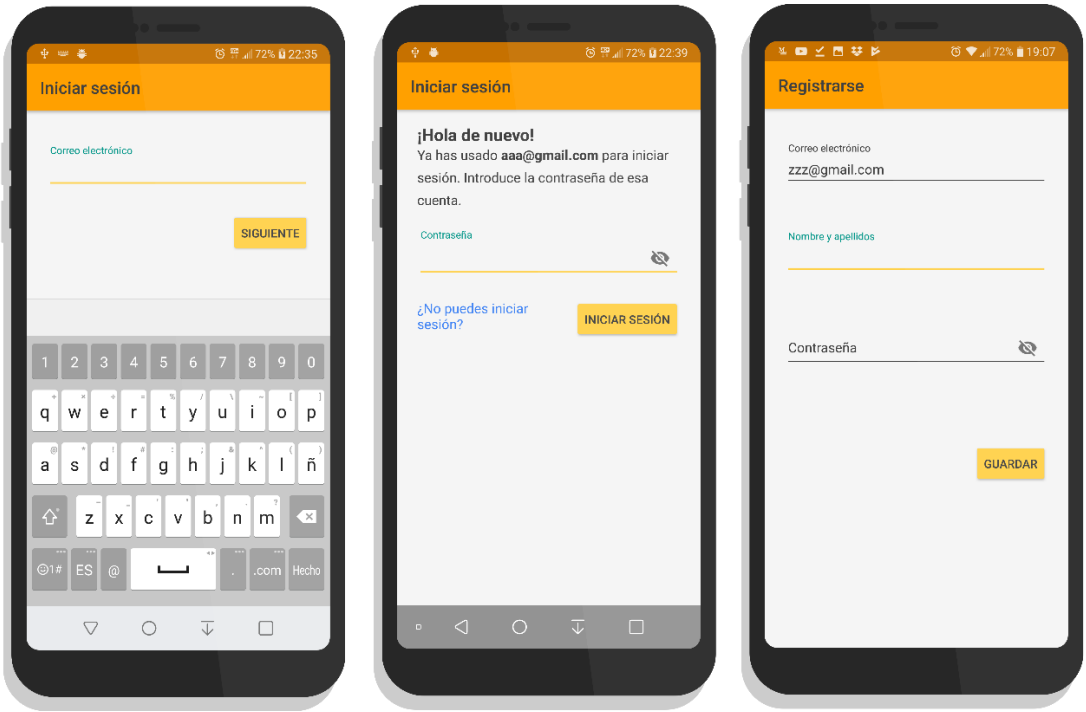


Ilustración 13: Pantallas de inicio de sesión con correo electrónico

4.2 MainActivity

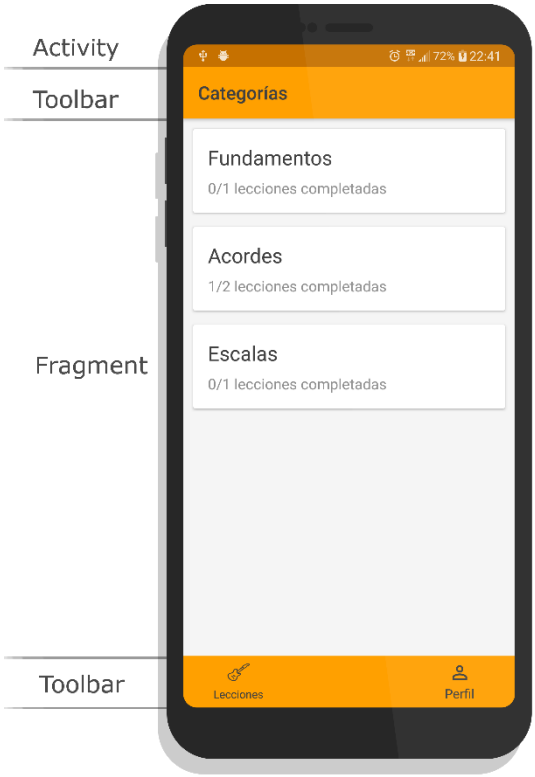


Ilustración 14: Vista principal de la aplicación

En la ***ilustración 14***, se puede ver la pantalla principal, *CategorySelectFragment*, a la que se navega una vez algún usuario se ha autenticado.

En la *Toolbar* superior, se puede ver el título del contexto actual. En la inferior, se encuentran los iconos para navegar entre secciones.

El contenido de la actividad (*Fragment*) se presenta mediante un componente en el XML que permite contener fragmentos, *FrameLayout*.

```
<!-- Content -->
<FrameLayout
    android:id="@+id/main_screen_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginTop="@dimen/toolbar_height"
    android:layout_marginBottom="@dimen/toolbar_height"
    />
```

Código 14: Contenedor en el que se colocan los fragments

Para reemplazar su contenido y mostrar las distintas secciones se ha creado una función de extensión de la clase *FragmentActivity*, de la cual extiende *ApplicationActivity*, que se muestra a continuación:

```
fun FragmentActivity.showNewFragment(fragmentToShow: Fragment,
                                     fragmentPlaceHolder: Int,
                                     addToBackStack: Boolean = true) {

    with(supportFragmentManager) {
        val transaction = beginTransaction()
                .setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE)
                .replace(fragmentPlaceHolder, fragmentToShow)

        if(addToBackStack) {
            transaction.addToBackStack(null)
        } else {
            popBackStack(null, FragmentManager.POP_BACK_STACK_INCLUSIVE)
        }

        transaction.commit()
    }
}
```

Código 15: Código para reemplazar un fragment por otro

Este método recibe tres parámetros.

- fragmentToShow: la instancia de *Fragment* que va a reemplazar al actual.
- fragmentPlaceHolder: la id del contenedor en el que se está colocando el *Fragment*, en este caso "*main_screen_container*"
- addToBackStack (opcional, true por defecto): Si es un *Fragment* raíz, es decir, el de categorías o perfil, se llamará a la función con valor *false*, de modo que se limpie la cola y al pulsar el botón de "back" del dispositivo se finalice la aplicación. En caso contrario, se irán añadiendo los nuevos *Fragment* al *backstack*[5], para guardar un historial de la navegación y poder retroceder a los fragmentos anteriores mediante el botón "back" o la flecha del *Toolbar* superior.

En la **sección 6.4 Navegación**, se verá cómo dentro del ciclo de vida de un *Fragment*, se notifica a *ApplicationActivity* la navegación entre fragmentos.

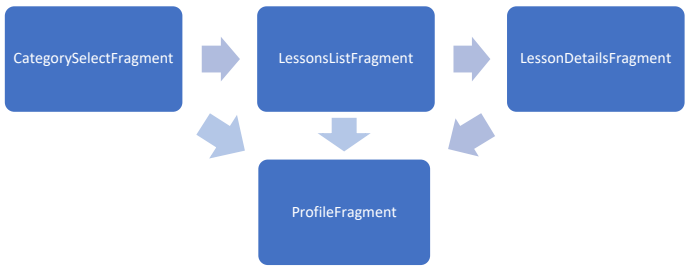


Ilustración 15: Flujo de navegación de los fragments en MainActivity

[5] El backstack es una pila, o lista LIFO, que sirve para guardar el orden de navegación y deshacerlo. Más información en: https://developer.android.com/guide/components/activities/tasks-and-back-stack

4.2.1 CategorySelectFragment

La primera vista, *CategorySelectFragment*, puede verse en la ***ilustración 14***. Consiste en una lista con tarjetas ordenadas por prioridad que corresponden a las categorías leídas de Firebase.

El proceso de creación de listas de la aplicación puede verse en detalle en la **sección 6.5**.

El contenido de cada item de la lista se define con un modelo de datos llamado *Category*:

```
data class Category(var id: String = "",
                    var title: String = "",
                    var lessonsCompleted: Int = 0,
                    var totalLessons: Int = 0) : Serializable
```

Código 16: Modelo de categoría

Para los modelos de la aplicación, se ha hecho uso de las clases de datos de Kotlin, ya que, con muy poco código, se obtiene el equivalente a una clase de Java con todos los métodos get, set, equals y hashCode. Todos los modelos de la aplicación son *serializables*[6] para poder añadirlos a *Bundles*[7].

Para navegar a la lista de lecciones de una categoría (*LessonsListFragment)*, es necesario crear el *intent* con la id y el título de la categoría:

```
class LessonsListFragment : Fragment(), LessonsListContract.View {
    companion object {
        val TAG = LessonsListFragment::class.java.simpleName
        const val CATEGORY_TO_REQUEST_PARAM = "CATEGORY_TO_REQUEST_PARAM"
        const val CATEGORY_TITLE_PARAM = "CATEGORY_TITLE_PARAM"

        fun getStartIntent(categoryId: String, title: String): Intent {
            val args = Bundle().apply {
                putString(CATEGORY_TO_REQUEST_PARAM, categoryId)
                putString(CATEGORY_TITLE_PARAM, title)
            }
            return Intent(TAG).putExtras(args)
        }

        fun newInstance(args: Bundle?): LessonsListFragment =
                LessonsListFragment().apply { arguments = args }

    }
    // ...
```

Código 17: Companion object en LessonsListFragment

Un companion object de Kotlin sirve para declarar métodos y variables que son accesibles sin necesidad de crear una instancia. Son el equivalente a los estáticos de Java.

[6] Codificable en un medio almacenable para su transmisión, como por ejemplo bytes en serie o un Json
[7] Contenedor de datos que se utiliza tanto para enviar información entre actividades como para guardar estados y recuperarlos más adelante

Cada ítem de la vista se crea de la siguiente forma:

```
override fun onBindViewHolder(holder: CategoryView, position: Int) {
    with(mCategoriesList[position]) {
        holder.bind(id, title, lessonsCompleted, totalLessons) {
            mView.navigateTo(LessonsListFragment.getStartIntent(id, title))
        }
    }
}
```

Código 18: Pasando información del modelo a la vista del item de categoría

Donde

- mCategoriesList es la lista de instancias de Category.
- navigateTo es el método que se llamará cuando se haga click en un ítem.

En la *Vista*, se hace uso de una función de extensión de *FragmentActivity* que se puede ver en la **sección 6.6**:

```
override fun navigateTo(intent: Intent) {
    activity?.broadcastTo(intent)
}
```

Código 19: Navegación a otro fragment

4.2.2 LessonsListFragment

Tras hacer click en una categoría, se listan las lecciones que contiene:

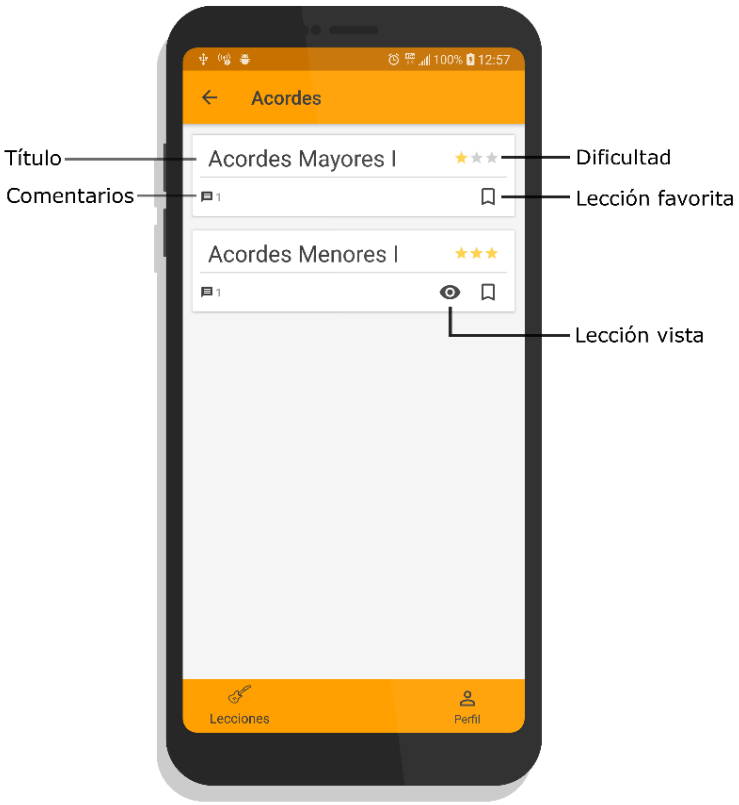


Ilustración 16: Lista de lecciones dentro de una categoría

Las únicas características no implementadas de la pantalla son las de mostrar el número de comentarios y la de añadir a favoritos, quedando como mejoras a futuro.

El marcador de lección vista se actualiza cuando el video asociado a la lección se ha reproducido, igual que marcar una lección como favorita.

En este caso, necesito un modelo que además de la información de la categoría, contenga la lista de lecciones:

```
data class CategoryContainer(val lessons: ArrayList<Lesson> = ArrayList(),
                             val category: Category = Category(),
                             var priority: Long = 1) {

    init {
        lessons.forEach { it.category = category.title }
        lessons.sortedWith(compareBy({it.difficulty}, {it.title}))
        category.totalLessons = lessons.size
    }
}
```

Código 20: El bloque init se ejecutará cada vez que se llame al constructor

Las lecciones mostradas se ordenan por dificultad y título.

Hacer click en una tarjeta, navegará a la vista en detalle de la lección, *LessonDetailsFragment:*

```
override fun onBindViewHolder(holder: LessonView, position: Int) {
    with(lessonsList[position]) {
        holder.bind(title, difficulty, isCompleted, isFavorite) {
            mView.navigateTo(LessonDetailsFragment.getStartIntent(this))
        }
    }
}
```

Código 21: Pasando información del modelo de lección a la vista del ítem

4.2.3 LessonDetailsFragment

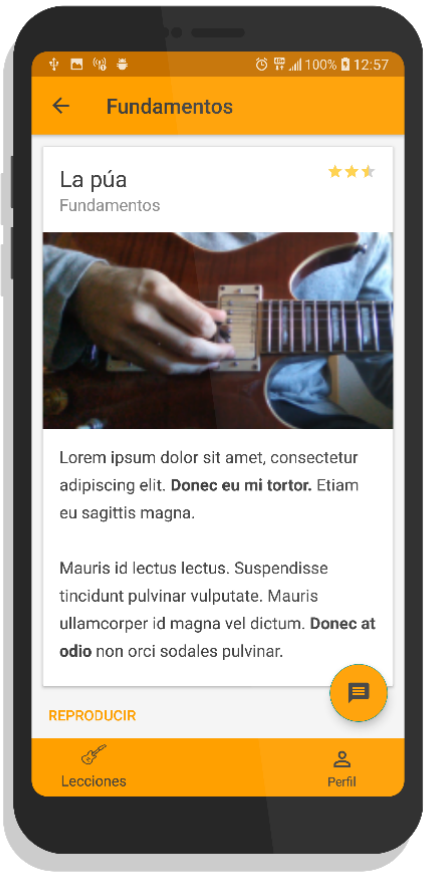


Ilustración 17: Vista de detalle de una lección

Esta pantalla muestra todos los contenidos del modelo *Lesson*:

```
data class Lesson(
        var id: String ="",
        var snapshot_path: String = "",
        var video_path: String = "",
        var title: String = "",
        var category: String = "",
        var dscr: String = "",
        var difficulty: Float = 0f,
        var isCompleted: Boolean = false,
        var isFavorite: Boolean = false) : Serializable {

    companion object {
        const val LESSON_KEY = "lesson_ley"
    }
}
```

Código 22: Modelo de Lección

Es una tarjeta deslizable en caso de que el contenido no quepa en la pantalla.

En el encabezado se inserta la imagen descargada de snapshot_path. El texto mostrado se parsea desde un HTML, por lo que es posible formatear párrafos y mostrar texto en negrita o cursiva.

El botón para abrir la pantalla de comentarios se ha implementado mediante un elemento llamado *FloatingButton* [19], para que siempre esté presente. La vista de comentarios utiliza un *BottomSheet* [20], por lo que es scrollable por el usuario y se puede cerrar de forma táctil.

Finalmente, el botón de reproducir se encuentra abajo a la izquierda, después de la tarjeta, de modo que se fomente la lectura de esta antes de reproducir el vídeo.

4.2.4 CommentsFragment

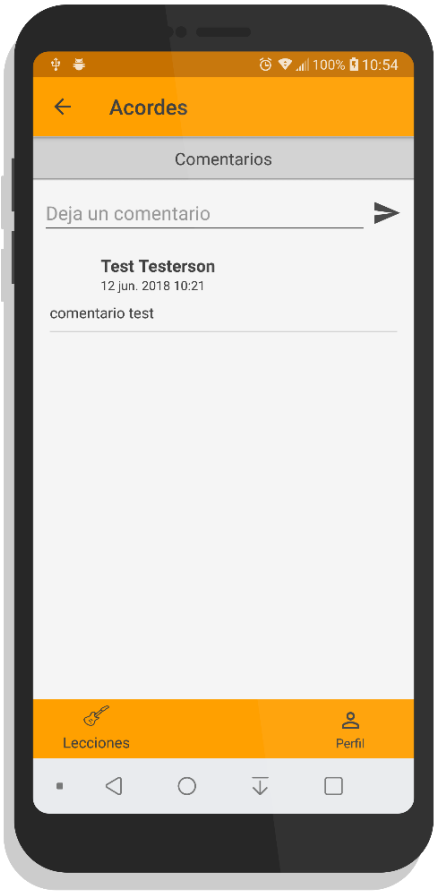


Ilustración 18: Vista de comentarios

Este *Fragment* no forma parte del flujo de navegación, ya que está contenido dentro de *LessonDetailsFragment*. Se ha implementado como *Fragment* por sencillez y coherencia con la arquitectura, pero como desarrollo futuro, se podría convertir este *Fragment* en una *View* personalizada para evitar anidar demasiados componentes con ciclo de vida propio.

CommentsFragment utiliza el comportamiento de *BottomSheet*. Para conseguirlo, se necesita incluir varios atributos en el *layout* del *BottomSheet* e incluirlo en un *CoordinatorLayout*.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:id="@+id/bottom_sheet"
    android:orientation="vertical"
    app:behavior_hideable="true"
    app:behavior_peekHeight="@dimen/peek_height"
    app:layout_behavior="android.support.design.widget.BottomSheetBehavior">
```

Código 23: Layout que implementa el behavior de BottomSheet

Donde

- behavior_hideable: permite que el layout pueda ocultarse por completo.
- behavior_peekHeight: es el tamaño de la pestaña para deslizar el componente.
- layout_behavior: añade el comportamiento de BottomSheet propiamente dicho.

En el *layout* de *LessonDetailsFragment*, se añade al final la siguiente línea para incluir el *BottomSheet*:

```
<include layout="@layout/bottom_sheet_messages" />
```

Código 24: Incluyendo el layout anterior en un CoordinatorLayout

Desde el editor, al abrir la pestaña de Preview del XML de *LessonDetailsFragment*, podemos observar que funciona correctamente:

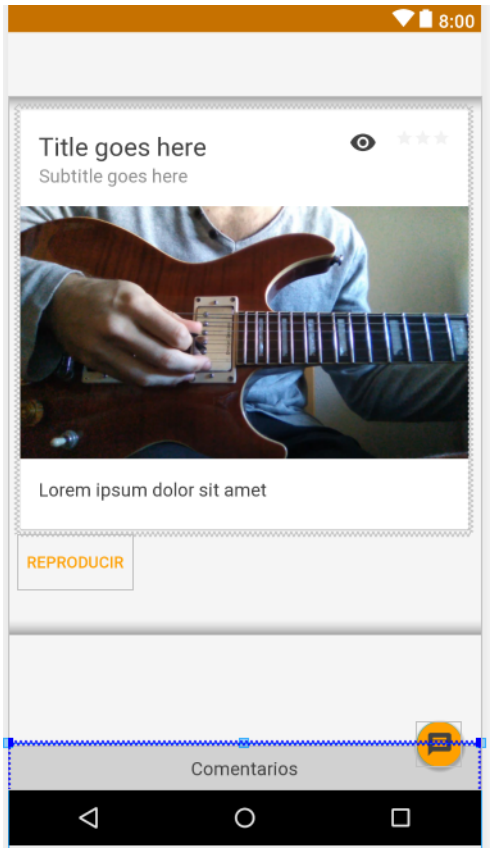


Ilustración 19: Previsualización desde el editor de Android Studio

Aparece cerrado y sólo sobresale la altura *behavior_peekHeight* especificada.

Para que el *BottomSheet* tenga el comportamiento deseado, hay que modificarlo por código:

```
override fun initializeFabButton() {
    fab_messages.setOnClickListener {
        if (!isPeekVisible) {
            bottomSheetBehavior.state = BottomSheetBehavior.STATE_EXPANDED
            isPeekVisible = true
        }
    }
}
```

Código 25: Inicialización de la acción del botón flotante

Con esto, al pulsar el botón flotante, se desplegará la *Vista* de comentarios.

En cuanto al comportamiento del panel deslizante:

```
override fun initializeBottomSheet() {
    bottomSheetBehavior.state = BottomSheetBehavior.STATE_HIDDEN
    bottomSheetBehavior.setBottomSheetCallback(object : BottomSheetCallback()
{
        override fun onSlide(bottomSheet: View, slideOffset: Float) {
            if (isPeekVisible) {
                fab_messages.animate()
                        .scaleX(1 - (Math.max(slideOffset, 0f)))
                        .scaleY(1 - (Math.max(slideOffset, 0f)))
                        .setDuration(0)
                        .start()
            }
        }

        override fun onStateChanged(bottomSheet: View, newState: Int) {
            if (newState == BottomSheetBehavior.STATE_HIDDEN) {
                fab_messages.setVisible()
            }
        }
    })
}
```

Código 26: Inicialización del comportamiento del BottomSheet

Esto hará varias cosas:

- Por defecto, se mostrará oculto. Esto es, no se visualizará la pestaña de scroll en la pantalla.
- El botón flotante realizará una animación de encogimiento/agrandamiento según se expanda o minimice el componente, llegando a ser invisible cuando está totalmente visible.

El modelo de datos de un comentario es el siguiente:

```
data class UserComment(val userId: String = "",
                       val userName: String = "",
                       val userPicturePath: String? = "",
                       val content: String = "",
                       val postDate: String = "") : Serializable
```

Código 27: Modelo de comentario

Donde

- userId: el token único del usuario en Firebase.
- userName: nombre del usuario que escribió el comentario.
- userPicturePath: ruta de la imagen de perfil del usuario, en caso de que iniciase sesión con una cuenta de Google.
- content: el texto del comentario.
- postDate: la fecha y hora del comentario siguiendo el estándar ISO_8601 [21]

Las clases relacionadas con fechas de java 6 y 7, por defecto, no manejan el formato ISO_8601. La opción escogida fue usar las utilidades incorporadas en Java 8, pero como éstas requieren una versión mínima de API de Android de 26, se ha optado por utilizar una librería que porta todas las utilidades de Java 8 a versiones anteriores: Threeten [22].

```
val dateTime = OffsetDateTime.now().toString()
```

Código 28: Obteniendo la fecha y hora actual como cadena de texto

Esta línea proporcionará una cadena con el siguiente formato: 2018-07-27T21:38:05.083+02:00.

Finalmente, la creación de un comentario listo para exportarse a la base de datos quedará así:

```
private fun createUserComment(content: String): UserComment {
    val currentUser = FirebaseAuth.getInstance().currentUser!!
    val uId = currentUser.uid
    val name = UserService.user?.name ?: ANONYMOUS
    val userPicturePath = currentUser.photoUrl.toString()
    val dateTime = OffsetDateTime.now().toString()
    return UserComment(uId, name, userPicturePath, content, dateTime)
}
```

Código 29: Creación de un comentario

Notar que se ha especificado explícitamente que el usuario actual (currentUser) no es nulo mediante el uso de la doble exclamación, ya que, si lo fuera, la aplicación navegaría automáticamente a la actividad de inicio de sesión. Este es un caso en el que se puede hacer la excepción de la que se habló previamente tras el bloque de ***código 2***.

En la **sección 5.3** se verá cómo se escribe esta información en la base de datos.

Para dar formato a la fecha de un comentario obtenido de la base de datos, es necesario usar *DateTimeFormatter*.

```
val format = DateTimeFormatter
        .ofLocalizedDateTime(FormatStyle.MEDIUM, FormatStyle.SHORT)
val postDate: String = OffsetDateTime.parse(postDate).format(format)
```

Código 30: Conversión de una cadena con fecha en formato ISO_8601 a un formato tipo "12 jun. 2018 10:21"

Donde

- ofLocalizedDateTime devolverá el formato de fecha y tiempo según los estilos especificados. El primer argumento será el estilo del formato de fecha y el segundo el de la hora.

El formato MEDIUM para la fecha, dependerá del idioma del sistema, siendo en castellano el día, seguido del mes abreviado y el año. El formato SHORT en el segundo parámetro, hará que se muestren horas y minutos, sin segundos.

Al final, el método que pasará la información del comentario a cada ítem de la lista queda así:

```
override fun onBindViewHolder(holder: UserCommentView, position: Int) {
    with(mCommentsList[position]) {
        val format = DateTimeFormatter
                .ofLocalizedDateTime(FormatStyle.MEDIUM, FormatStyle.SHORT)
        val postDate = OffsetDateTime.parse(postDate).format(format)
        holder.bind(userName, content, postDate, userPicturePath)
    }
}
```

Código 31: Pasando la información del modelo a la Vista de comentario

4.3 PLAYERACTIVITY

PlayerActivity es la actividad creada para contener el reproductor de video. Por ser consistente en la arquitectura, también implementa un *Fragment*.

Tiene sentido salirse de la arquitectura Single Activity en este caso, ya que esta pantalla tiene un estilo diferente, una presentación apaisada y no tiene *Toolbar*, a diferencia de *ApplicationActivity*.

Este Fragment (*PlayerControlsFragment)*, contiene un objeto *SimpleExoPlayerView*, elementos para definir los controles, información del estado de carga, mensajes de error y pantalla de fin de reproducción.

ExoPlayer [23] es un reproductor multimedia de código abierto desarrollado por Google con muchas funciones que no incorpora por defecto el reproductor nativo de Android. Para utilizarlo, basta con añadir en el fichero *build.gradle* la siguiente dependencia:

```
implementation 'com.google.android.exoplayer:exoplayer:2.x.x'
```

Código 32: Añadiendo la librería de ExoPlayer en Gradle

Donde '2.x.x' debe ser sustituido por la versión utilizada, siendo la última en este momento 2.8.4.

También ofrecen como alternativa añadir sólo los módulos necesarios. Para mantener el tamaño final de la aplicación lo más reducido posible, en esta aplicación se han implementado los tres módulos utilizados:

```
implementation "com.google.android.exoplayer:exoplayer-
core:$exoplayer_version"
implementation "com.google.android.exoplayer:exoplayer-ui:$
exoplayer_version"
implementation "com.google.android.exoplayer:exoplayer-dash:$
exoplayer_version"
```

Código 33: Módulos utilizados de ExoPlayer en la aplicación

Para mantener el código limpio, el número de versión de las dependencias de ExoPlayer, exoplayer_version, está referenciando un punto externo en el que se encuentran todas las versiones de las librerías utilizadas. La versión utilizada en el proyecto es la 2.7.3.

Los tres módulos empleados constan de la funcionalidad principal, los componentes visuales (para poder utilizar *SimpleExoPlayerView*), y el módulo de DASH.

DASH, o MPEG-DASH (Dynamic Adaptive Streaming over HTTP), es un formato de codificación que utiliza ancho de banda variable y permite streaming de alta calidad a través de servidores HTTP.

Por defecto, una Vista de ExoPlayer incorpora unos controles básicos que se muestran al tocar la pantalla:

Ilustración 20: Imagen de la aplicación de ejemplo de ExoPlayer

Estos controles por defecto no eran ideales para la aplicación. En la etapa de diseño se definió que las funciones que debe realizar el reproductor son:

- Pausar y reanudar la reproducción.
- Adelantar y retroceder una cantidad fija de tiempo (10 segundos).
- Poder silenciar la reproducción con un simple click.
- Homogeneidad con el tema de la aplicación.

Por lo que se ha optado por crear unos propios más apropiados. Para deshabilitar los controles por defecto se requiere especificarlo en el atributo *app:use_controller*:

```
<com.google.android.exoplayer2.ui.SimpleExoPlayerView
    app:use_controller="false"
    android:id="@+id/exoplayer_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Código 34: Componente del reproductor en un layout

Los controles personalizados se han definido en un *ConstraintLayout* utilizando gráficos vectoriales de Material Design que ya estaban predefinidos en Android Studio. El resultado final se aprecia en la ***ilustración 21***:

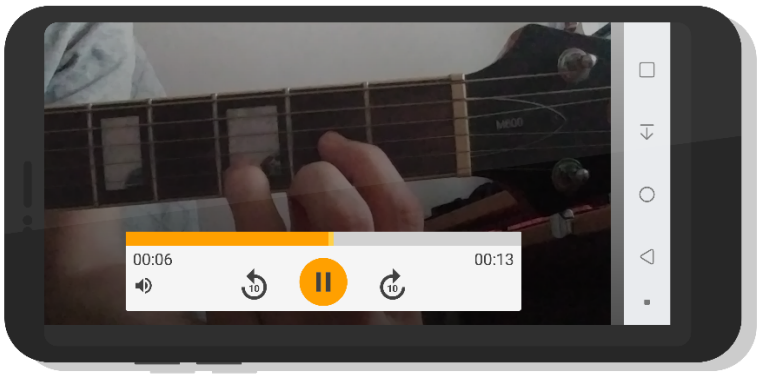


Ilustración 21: Controles personalizados

Para utilizar los controles personalizados el primer paso fue detectar el gesto táctil en la pantalla en *PlayerActivity* y notificarlo al *PlayerControlsFragment* a través de la función *onTouchEvent*.

```
override fun dispatchTouchEvent(ev: MotionEvent?): Boolean {
    ev?.let { event ->
        if (event.action == MotionEvent.ACTION_DOWN) {
            playerControlsFragment.onTouchEvent(event)
        }
    }

    return super.dispatchTouchEvent(ev)
}
```

Código 35: Gestión del evento táctil en la activity

```
override fun onTouchEvent(event: MotionEvent) {

    if (!isAnimating && !ended) { // Don't repeat this while the hide/show
animation is already performing

        if (!controlsVisible) { // Open up the controls when the screen is
touched
            setControlsVisible()
            startCountdownToHideControls()
        } else {
            if (areControlsOutsideOfTouchEvent(event)) {
                uiThreadHandler.removeCallbacks(hideControlsRunnable)
                setControlsInvisible()
            } else {
                // If the controls are touched, start the task again
                startCountdownToHideControls()
            }
        }
    }
}
```

Código 36: Gestión del evento táctil en el fragment de controles

Algunas aclaraciones sobre las variables comprobadas en las condicionales:

- isAnimating: para evitar que el código vuelva a ejecutarse mientras está sucediendo una animación de mostrar/ocultar.
- ended: cuando ExoPlayer notifica que el video ha terminado de reproducirse, guardo este estado en el *Fragment* y muestro una pantalla de fin de reproducción. Mientras esa pantalla se muestra, no quiero que se puedan mostrar los controles de reproducción.
- controlsVisible: verdadero si los controles son visibles.

El flujo del método es el siguiente:

- Se comprueba que no está en ejecución la animación y el video está en reproducción.
- Si los controles no son visibles, llama al método que ejecuta la animación de mostrarlos y comienza una tarea para ocultarlos automáticamente pasados unos instantes. La animación es una traslación desde la parte inferior de la pantalla hasta su posición final.
- Si son visibles, se comprueba la posición del click mediante colisiones con areControlsOutsideOfTouchEvent(event).
- Si el gesto ocurrió fuera del área de controles, los ocultará mediante una animación.
- Si el gesto ocurrió en el área de los controles, se reiniciará la tarea que los oculta automáticamente. De esta manera, se evita que los controles se oculten mientras están siendo utilizados.

Por otro lado, el método setupClickListeners() codifica la acción de cada botón:

```
override fun setupClickListeners() {
    playback_progress_seek_bar.setOnSeekBarChangeListener(object :
            SeekBar.OnSeekBarChangeListener {
        override fun onProgressChanged(seekBar: SeekBar?, progress: Int,
fromUser: Boolean) {}

        override fun onStartTrackingTouch(seekBar: SeekBar?) {
            isTouchingSeekbar = true
        }

        override fun onStopTrackingTouch(seekBar: SeekBar?) {
            isTouchingSeekbar = false
            val percentage = playback_progress_seek_bar.progress / 100f
            val newValue = percentage * player.duration
            player.seekTo(newValue.toLong())
        }
    })

    mute_button.setOnClickListener {
        player.volume = if (player.volume > 0) 0f else 1f
    }

    player_rewind_button.setOnClickListener {
        val newSeekTime = player.currentPosition - TEN_SECONDS
        player.seekTo(newSeekTime)
    }

    player_forward_button.setOnClickListener {
        player.seekTo(if (player.currentPosition + TEN_SECONDS >
player.duration) {
            player.duration
        } else {
            player.currentPosition + TEN_SECONDS
        })
    }

    player_playpause_button.setOnClickListener {
        if (controlsVisible) {
            if (!paused) pausePlayback() else resumePlayback()
        }
    }
}
```

Código 37: Código para los clicks de cada botón

Finalmente, para que la barra de progreso y los textos laterales se actualicen cada segundo, se ha requerido del siguiente código:

```
override fun initializeUIPeriodicTask() {
    updateTimeAndSeekRunnable = Runnable {
        if (player.currentPosition >= 0 && player.duration > 0) {
            val currentTime = player.currentPosition
            val totalTime = player.duration
            updateSeekTimes(currentTime, totalTime)
        }

        uiThreadHandler.postDelayed(updateTimeAndSeekRunnable,
uiRefreshTime) // post this runnable again to make this periodically
    }

    uiThreadHandler.postDelayed(updateTimeAndSeekRunnable, 0)
}
```

Código 38: Tarea periódica para actualizar progreso de la reproducción

```
private fun updateSeekTimes(currentTime: Long = 0, duration: Long = 1) {
    if (!isTouchingSeekbar) { // Avoid time updating if the user is
seeking manually
        current_time_text?.text = currentTime.parseTimeToMMSS()
        duration_text?.text = duration.parseTimeToMMSS()

        playback_progress_seek_bar?.progress =
                (currentTime * 100 / duration).toInt()
    }
}
```

Código 39: Actualiza tiempo de progreso desde la tarea periódica o desde un click

Además, hay dos estados más que son controlados. Cuando el video está cargando y cuando la reproducción finaliza:

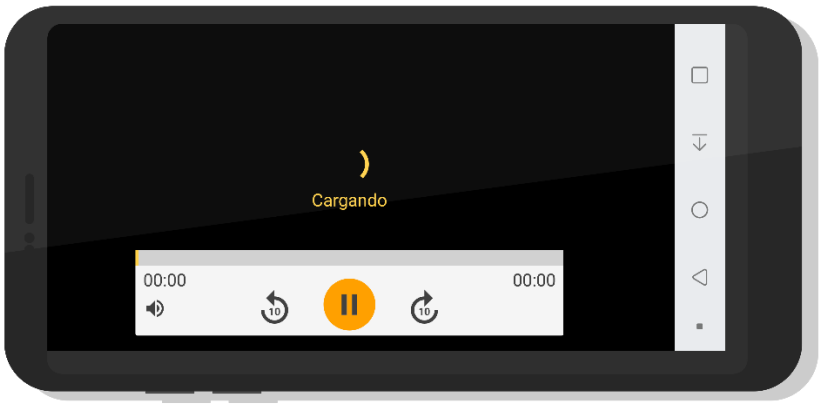


Ilustración 22: La barra de progreso se muestra cada vez que se está descargando

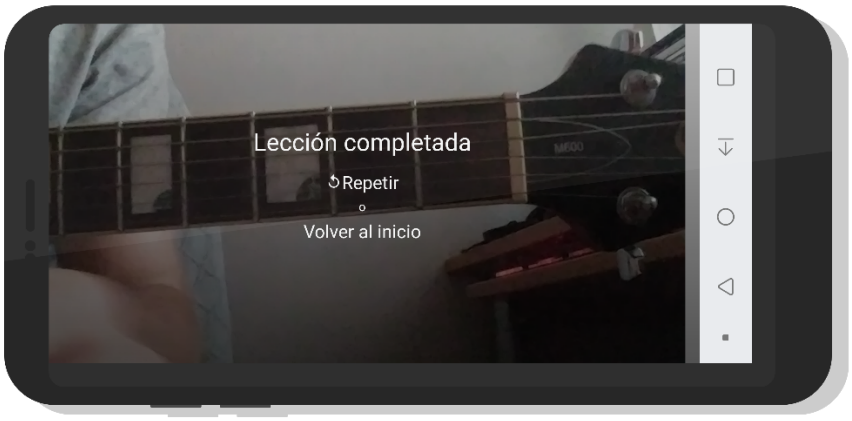


Ilustración 23: Pantalla de fin de reproducción

Para mostrar estas pantallas es necesario controlar el estado del Player:

```
override fun initializePlayerEventListener() {
    playerEventListener = object : Player.DefaultEventListener() {
        override fun onPlayerStateChanged(playWhenReady: Boolean,
playbackState: Int) {
            when {
                playbackState == Player.STATE_BUFFERING -> showBufferingView()
                playWhenReady && playbackState == Player.STATE_READY -> {
                    hideBufferingView()
                    startCountdownToHideControls()
                }
                playbackState == Player.STATE_ENDED -> {
                    if (!ended) {
                        presenter.onPlaybackEnded()
                    }
                    ended = true
                }
            }
        }

        override fun onPlayerError(error: ExoPlaybackException?) {
            showToast(getString(R.string.playback_error))
        }
    }
    player.addListener(playerEventListener)
}
```

Código 40: Listener para los cambios de estado del Player

Este método añade añade al Player un listener *DefaultEventListener* que es llamado cada vez que el estado de la reproducción cambia.

Cuando el estado playbackState es STATE_BUFFERING, se mostrarán los elementos de la ***ilustración 22***, mientras que en STATE_ENDED se notificará al *Presenter* que la reproducción ha terminado.

La llamada a *presenter.onPlaybackEnded()* realiza el siguiente bloque de código:

```
override fun onPlaybackEnded() {
    lesson.isCompleted = true
    updateWatchedLessonUseCase.updateLesson(lesson) {errorStr ->
        mView.showToast(errorStr)
    }
    mView.showFinishedView()
}
```

Código 41: Marca la lección como vista y notifica a la Vista que muestre la pantalla de fin de reproducción

Este actualiza el modelo de la lección, marcándola como completada. Acto seguido hace una llamada asíncrona al *Caso de Uso*, que mostrará un mensaje en la *Vista* en caso de error al comunicar con el servidor o descargar el fichero de vídeo. El método finaliza mostrando la pantalla de fin de la ***ilustración 23***.

5. BACKEND

5.1 CONFIGURACIÓN DE FIREBASE

El primer paso es añadir el proyecto en la consola de Firebase [24]:

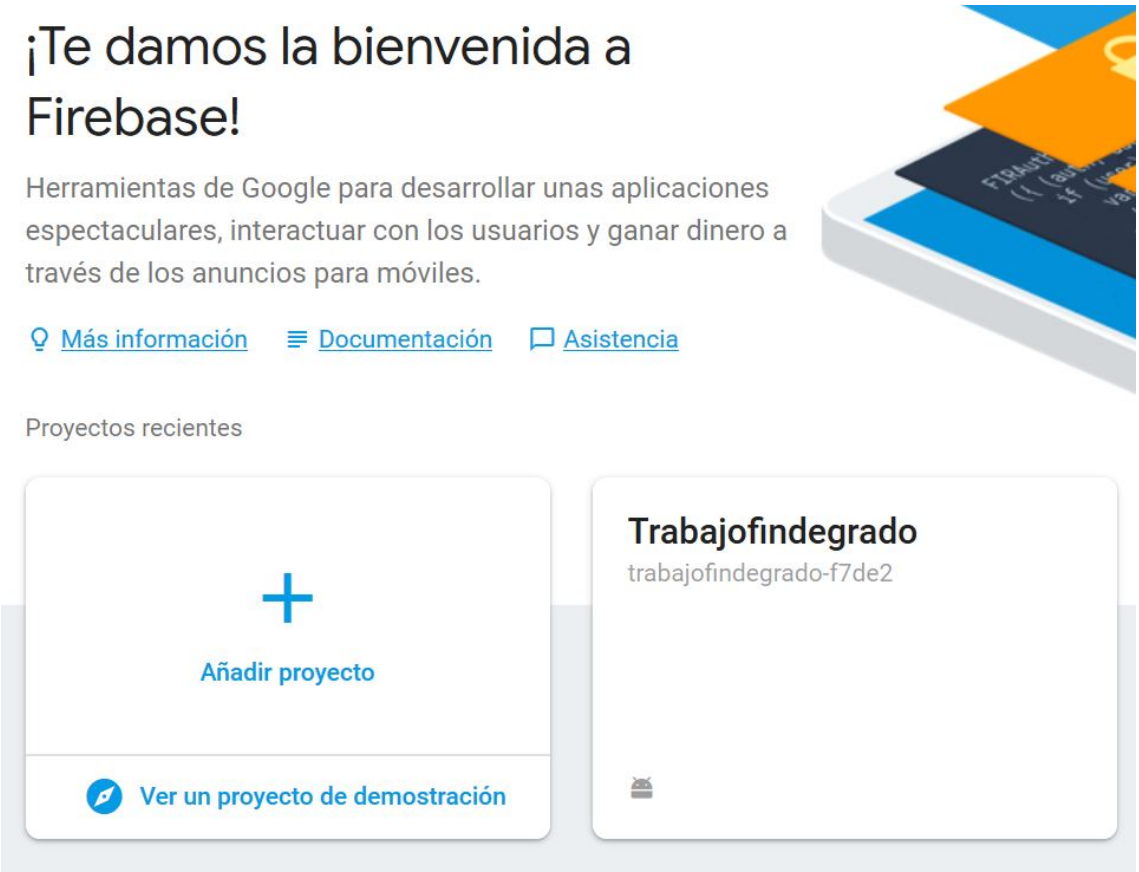


Ilustración 24: Vista principal de la consola de Firebase

Siguiendo el asistente, aparecerá un enlace de descarga, que también podrá visitarse en el futuro desde la pestaña de configuración del proyecto:

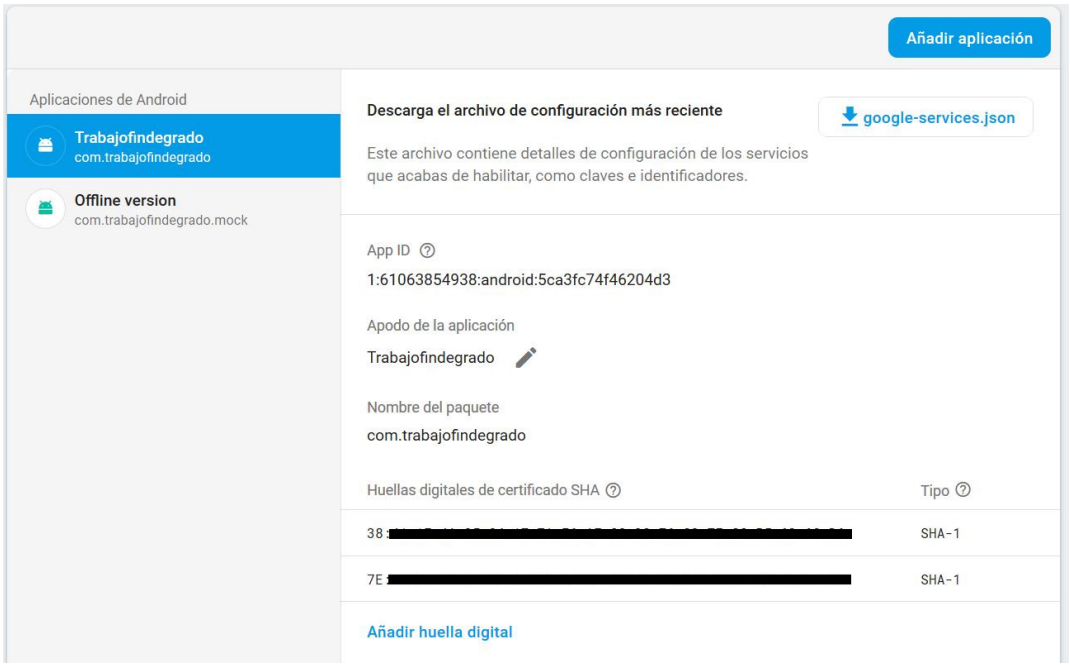


Ilustración 25: Página de configuración del proyecto

Notar que hay dos aplicaciones configuradas en el proyecto, "Offline version" es necesaria debido a que la variante con *mocks*[8] de la aplicación tiene un identificador diferente y Firebase necesita reconocerlo para compilar, aunque no se utilice.

Además de incluir las librerías en Gradle (**sección 6.7**), hay que descargar el archivo *google-services.json* e incluirlo en la raíz del proyecto de Android Studio:

[8] Objeto que simula el comportamiento de un objeto real. Se utiliza en test unitarios, para verificar el comportamiento.

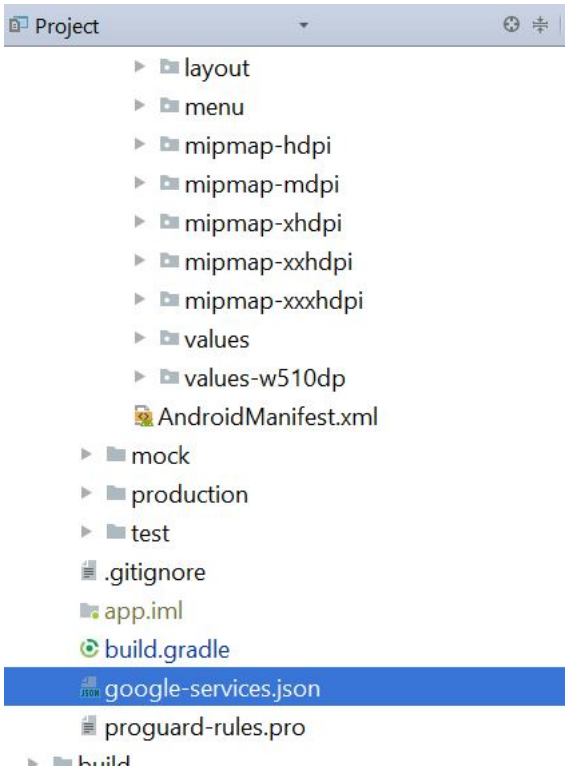


Ilustración 26: Ubicación de google-services.json desde la vista de Proyecto

5.2 AUTHENTICATION

Para configurar la autenticación, es necesario ir a la categoría “Authentication” desde la consola.

Cuando no hay usuarios, esta será la primera vista:

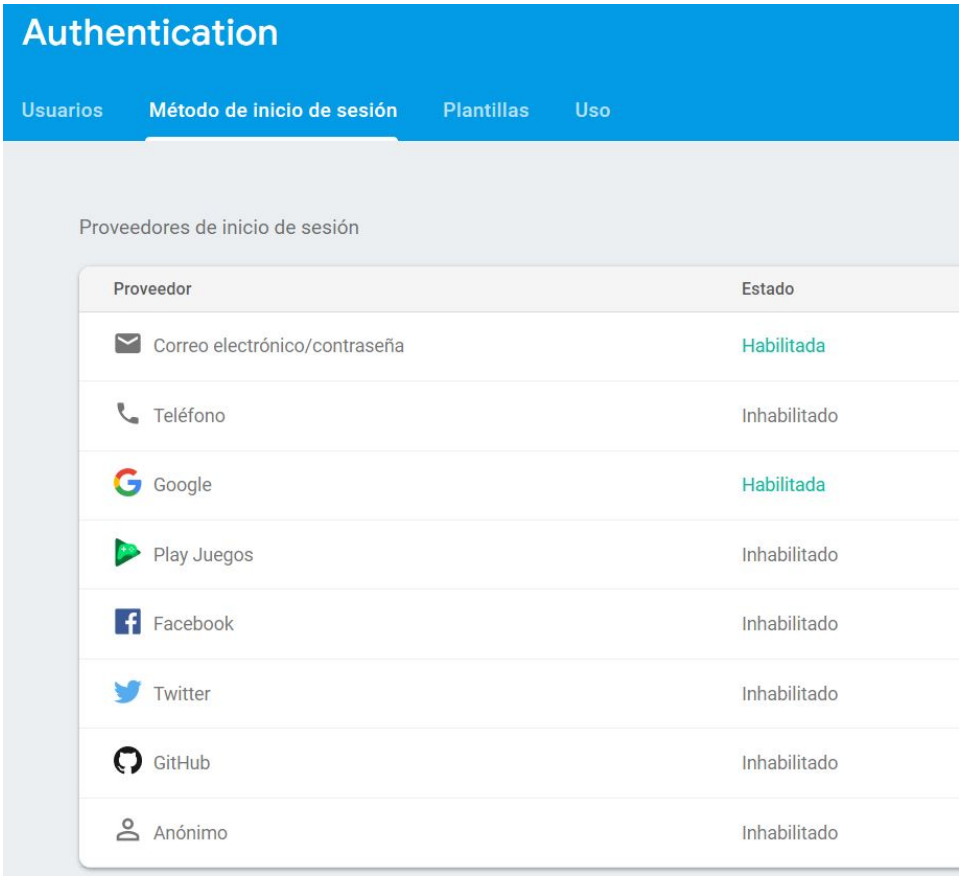


Ilustración 27: Pestaña de configuración de métodos de inicio de sesión

Una vez se hayan creado usuarios, se pueden ver en la pestaña “Usuarios”:

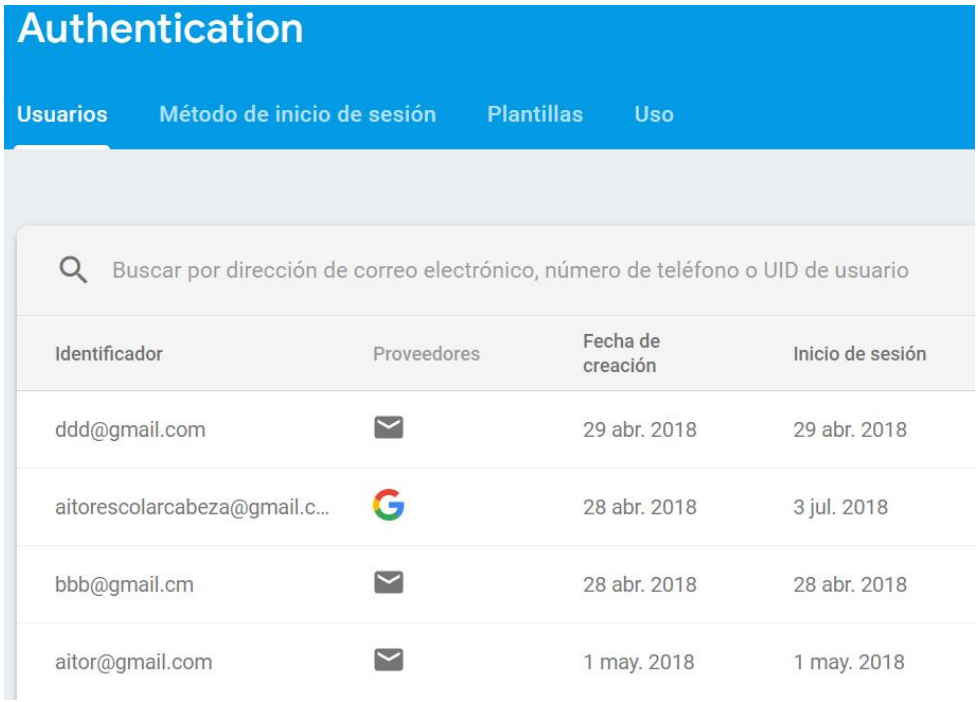


Ilustración 28: Lista de usuarios guardados

El siguiente bloque de código, muestra cómo sería el proceso de registro con email y contraseña de forma manual:

```
class RegisterWithEmailUseCase() : RegisterUserUseCaseInterface {

    private val mAuth: FirebaseAuth = FirebaseAuth.getInstance()

    override fun createUser(user: String,
                            password: String,
                            onRegisterUserCallback: OnRegisterUserCallback) {
        mAuth.createUserWithEmailAndPassword(user, password)
                .addOnCompleteListener { task ->
                    if (task.isSuccessful) {
                        onRegisterUserCallback.onRegisterSuccess()
                    } else {
                        val error = task.exception.toString()
                        onRegisterUserCallback.onRegisterError(error)
                    }
                }
    }
}
```

Código 42: Ejemplo de código para registrar un usuario con email y contraseña

Sin embargo, con Firebase-AuthUI, no es necesario añadir ninguna lógica para gestionarlo, simplemente lanzar la actividad:

```
val providers = listOf(
        AuthUI.IdpConfig.GoogleBuilder().build(),
        AuthUI.IdpConfig.EmailBuilder().build())

startActivityForResult(
        AuthUI.getInstance()
                .createSignInIntentBuilder()
                .setIsSmartLockEnabled(false)
                .setTheme(R.style.FirebaseTheme)
                .setAvailableProviders(providers)
                .build(),
        RC_SIGN_IN)
```

Código 43: Lanzando la actividad de registro e inicio de sesión con dos proveedores (Google y email)

AuthUI.IdpConfig contiene todas las factorías para crear distintos métodos de sesión.

Esta *Activity* tiene una estructura fija, pero permite usar un estilo personalizado. FirebaseTheme es un estilo creado dentro del fichero styles.xml para que la pantalla tenga la misma paleta de colores que el resto de la aplicación.

5.3 DATABASE

Firebase Realtime Database es una base de datos no relacional que permite importar y exportar ficheros JSON.

Desde la consola, se puede ver como una estructura jerárquica en árbol:

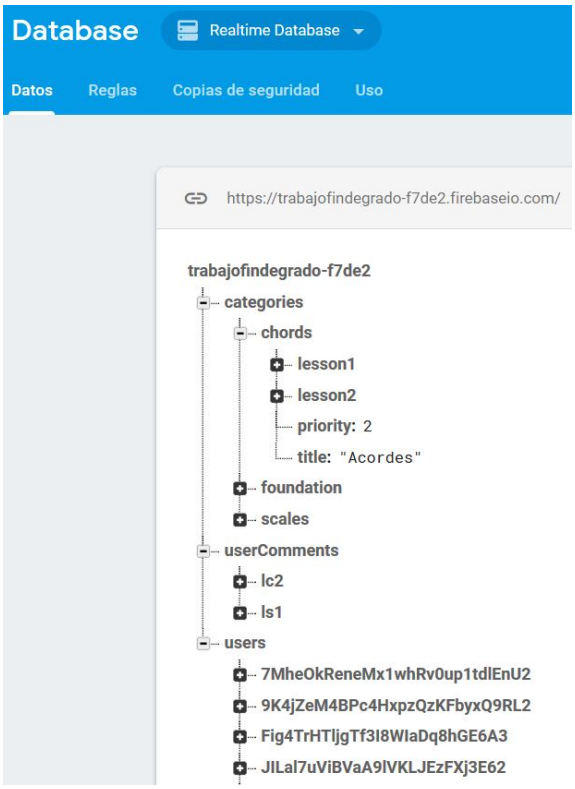


Ilustración 29: Vista global de la base de datos

Donde:

- categories: agrupa las categorías que contienen las lecciones.
- userComments: aquí, cada ítem corresponde al identificador único de una lección y almacena los comentarios.
- users: guarda estadísticas del usuario, como lecciones vistas o favoritas.

Los elementos finales desplegados son:

Ilustración 30: Campos de una lección

Ilustración 31: Comentarios de una lección

Aquí, el primer hijo de *userComments* es la id de la lección a la que corresponden los comentarios. Los hijos de la lección tienen un identificador único generado por Firebase.

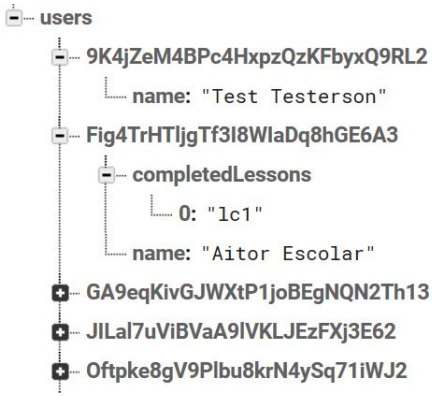


Ilustración 32: Entradas de usuarios

En este caso, dentro de *users* se encuentra la lista con el identificador único de cada usuario. El campo *completedLessons* tiene los identificadores de las lecciones que ha visto. Como mejora a futuro, también tendría que almacenar las lecciones favoritas.

En la pestaña “Reglas”, se pueden configurar los permisos. En esta aplicación, basta con que el usuario esté autenticado (auth != null) para que pueda leer todos los contenidos, salvo en el caso de las categorías, donde no tiene permitido escribir:

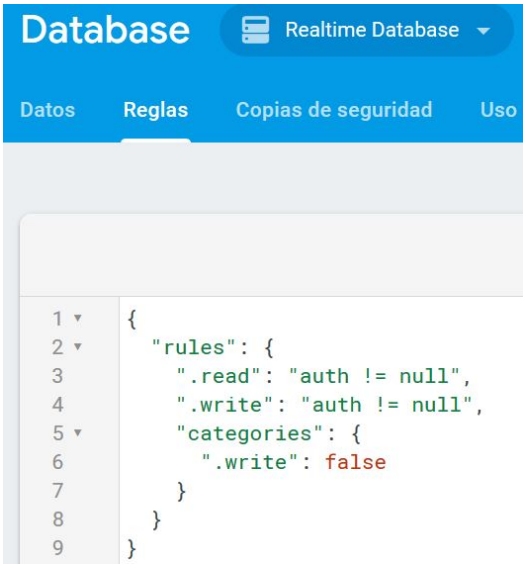


Ilustración 33: Permisos de la base de datos

A efectos prácticos, desde la aplicación, el usuario no podrá hacer nada que no permita la interfaz, pero es una medida de seguridad añadida.

A continuación, se adjunta el código de los casos de uso que realizan las operaciones relacionadas con la base de datos.

5.3.1 Lectura de categorías

```
interface ObtainCategoriesUseCase {
    interface ObtainCategoriesCallback {
        fun onContentLoaded(categoryContainers: List<CategoryContainer>)
        fun onError(error: String)
    }

    fun loadCategoryContainers(targetFilter: String? = null
                               callback: ObtainCategoriesCallback)

    fun stop()
}
```

Código 44: Interfaz del caso de uso

La implementación se encuentra en *ObtainCategoriesUseCaseImpl*.

```
override fun loadCategoryContainers(targetFilter: String?,
                                    callback:
ObtainCategoriesUseCase.ObtainCategoriesCallback) {
    val requestParams = getRequestParams(targetFilter)
    loadCategoryContainersFromFirebase(requestParams, callback)
}
```

Código 45: Comprobaciones antes de leer categorías desde Firebase

```
private fun loadCategoryContainersFromFirebase(databaseTag: String = DATABASE_TAG,
                                               callback: ObtainCategoriesCallback
) {

    reference = FirebaseDatabase.getInstance().getReference(databaseTag)

    valueEventListener = object : ValueEventListener {
        override fun onCancelled(p0: DatabaseError?) {
            callback.onError(p0.toString())
        }

        override fun onDataChange(contents: DataSnapshot?) {
            contents?.let {
                val categoryContainers = ArrayList<CategoryContainer>()
                for (child in contents.children) {
                    val category = parseCategoryContainer(child)
                    categoryContainers.add(category)
                }

                val categoriesListSortedByPriority = categoryContainers.
                        sortedWith(compareBy({it.priority}))

                callback.onContentLoaded(categoriesListSortedByPriority)
            }
        }
    }

    reference.addValueEventListener(valueEventListener)
}
```

Código 46: Código completo de lectura de categorías

Donde databaseTag será la ruta en la que se obtendrá la referencia, en este caso,

```
categories/
```

El método *onDataChange* del *ValueEventListener* se llamará la primera vez que se añada el *listener* a la referencia, o cuando se inserte, modifique o elimine algún contenido de la base de datos.

En caso de que el *DataSnapshot* no esté vacío, se itera entre los hijos (children). Cada hijo es también una instancia de *DataSnapshot*.

El método *parseCategoryContainer* crea la instancia de *CategoryContainer* a partir de la información del *DataSnapshot*.

5.3.2 Escritura de comentarios

```
interface WriteCommentUseCase {
    fun writeComment(content: String,
                     lesson: Lesson,
                     onSuccessCallback: () -> Unit = {},
                     onErrorCallback: ErrorCallback = {})
}
```

Código 47: Interfaz del caso de uso para escribir comentarios

```
class WriteCommentUseCaseImpl : WriteCommentUseCase {
    private val firebaseDatabase: FirebaseDatabase by lazy {
FirebaseDatabase.getInstance() }
    private val DATABASE_TAG = "userComments"
    private val ANONYMOUS = "Anonymous"

    override fun writeComment(content: String,
                              lesson: Lesson,
                              onSuccessCallback: () -> Unit,
                              onErrorCallback: ErrorCallback) {
        val reference =
firebaseDatabase.getReference("$DATABASE_TAG/${lesson.id}")
        val userComment = createUserComment(content)

        reference.push().setValue(userComment).addOnCompleteListener {
            if(!it.isSuccessful) {
                onErrorCallback(it.exception.toString())
            } else {
                onSuccessCallback()
            }
        }
    }

    private fun createUserComment(content: String): UserComment {
        val currentUser = FirebaseAuth.getInstance().currentUser!!
        val uId = currentUser.uid
        val name = UserService.user?.name ?: ANONYMOUS
        val userPicturePath = currentUser.photoUrl.toString()
        val dateTime = OffsetDateTime.now().toString()
        return UserComment(uId, name, userPicturePath, content, dateTime)
    }
}
```

Código 48: Implementación por defecto del caso de uso

Primero se obtiene la referencia con la ruta en la que se va a crear el comentario. Por ejemplo, un comentario en la lección con identificador “lc1” tendrá una referencia “userComments/lc1”.

En este caso, la operación es de escritura, por lo que no es necesario añadir un *ValueEventListener* a la referencia.

En primer lugar, se crea una instancia del modelo del comentario, *UserComment*. Después, se realiza una operación PUSH para añadir el contenido pasado en el método

setValue. Opcionalmente, se puede añadir un *OnCompleteListener* para recibir una notificación una vez la operación ha terminado en el servidor.

La referencia en este caso será "userComments" seguido del identificador único de la lección. Por ejemplo, para "lc1", un comentario se hallaría en la siguiente ruta:

```
userComments/lc1/
```

Firebase creará automáticamente una clave única para el comentario escrito en la base de datos.

5.3.3 Lectura de comentarios

```
interface ObtainCommentsUseCase {
    interface ObtainCommentsCallback {
        fun onLoaded(commentsList: List<UserComment>)
        fun onError(error: String)
    }

    fun start(lesson: Lesson, callback: ObtainCommentsCallback)

    fun stop()
}
```

Código 49: Caso de uso para obtener comentarios

```
class ObtainCommentsUseCaseImpl: ObtainCommentsUseCase {

    lateinit var valueEventListener: ValueEventListener
    private val firebaseDatabase: FirebaseDatabase by lazy {
FirebaseDatabase.getInstance() }
    private val DATABASE_TAG = "userComments"

    lateinit var reference: DatabaseReference

    override fun start(lesson: Lesson,
                       callback:ObtainCommentsCallback) {
        initValueEventListener(callback)
        reference = firebaseDatabase.getReference("$DATABASE_TAG/${lesson.id}")
        reference.addValueEventListener(valueEventListener)
    }

    override fun stop() {
        if (::valueEventListener.isInitialized) {
            reference.removeEventListener(valueEventListener)
        }
    }
}
```

Código 50: Implementación del caso de uso de obtener comentarios

```
private fun initValueEventListener(callback: ObtainCommentsCallback) {
    valueEventListener = object : ValueEventListener {
        override fun onCancelled(p0: DatabaseError?) {
            callback.onError(p0.toString())
        }

        override fun onDataChange(dataSnapshot: DataSnapshot?) {
            dataSnapshot?.let { snapshot ->
                val commentsList = ArrayList<UserComment>()
                for (comment in snapshot.children) {
                    val c = comment.getValue(UserComment::class.java)
                    commentsList.add(c!!)
                }
                callback.onLoaded(commentsList)
            }
        }
    }
}
```

Código 51: Escucha cambios en la base de datos para actualizar los comentarios

La lógica de este *Caso de Uso* es muy similar a la de lectura de categorías vista en el **apartado 5.3.1**.

Cada hijo del *dataSnapshot* será un comentario, que se añadirá a la lista de comentarios. Una vez terminado, se notifica al *Presenter*.

5.3.4 Lectura de información de usuario

```
interface ReadUserDataUseCase {
    interface ReadUserCallback {
        fun onUserDataRead(user: User)
        fun onError(error: String)
    }

    fun readUser(callback: ReadUserCallback)
    fun stop()
}
```

Código 52: Interfaz del caso de uso para leer los datos de usuario

```
class ReadUserDataUseCaseImpl : ReadUserDataUseCase {
    private val firebaseDatabase: FirebaseDatabase by lazy {
FirebaseDatabase.getInstance() }
    private val DATABASE_TAG = "users"

    lateinit var reference: DatabaseReference
    lateinit var valueEventListener: ValueEventListener

    override fun readUser(callback: ReadUserDataUseCase.ReadUserCallback) {
        if (UserService.user != null) {
            callback.onUserDataRead(UserService.user!!)
        } else {
            loadUserDataFromFirebase { user ->
                UserService.user = user
                callback.onUserDataRead(user)
            }
        }
    }
```

Código 53: Inicio de la implementación del caso de uso

```
private fun loadUserDataFromFirebase(onComplete: (User) -> Unit) {
    reference = firebaseDatabase.getReference(DATABASE_TAG)

    valueEventListener = object : ValueEventListener {
        override fun onCancelled(p0: DatabaseError?) {}

        override fun onDataChange(dataSnapshot: DataSnapshot) {
            FirebaseAuth.getInstance().currentUser?.apply {
                val user: User = if(!dataSnapshot.hasChild(uid)) {
                    val u = UserService.createEmptyUser(displayName!!)
                    reference.child(uid).setValue(u)
                    u
                } else {
                    dataSnapshot.child(uid).getValue(User::class.java)!!
                }
                onComplete(user)
            }
        }
    }
    reference.addListenerForSingleValueEvent(valueEventListener)
}
```

Código 54: Método loadUserDataFromFirebase

Este *Caso de Uso* tiene la particularidad de que, si no existe en la base de datos una entrada con el identificador del usuario, creará una entrada nueva y la almacenará. Esto ocurrirá la primera vez que un usuario entre en la aplicación.

En caso de que el usuario sí exista, se obtendrá y se almacenará en un modelo *User*.

```
data class User(var name: String = "",
                val completedLessons: ArrayList<String> = ArrayList(),
                val favoritedLessons: ArrayList<String> = ArrayList())
```

Código 55: Modelo de usuario

5.3.5 Actualización de lecciones vistas

```
interface UpdateWatchedLessonUseCase {
    fun updateLesson(lesson: Lesson,
                     onErrorCallback: ErrorCallback)
}
```

Código 56: Interfaz del caso de uso

```
class UpdateWatchedLessonUseCaseImpl : UpdateWatchedLessonUseCase {

    private val firebaseDatabase: FirebaseDatabase by lazy {
        FirebaseDatabase.getInstance()
    }
    private val DATABASE_TAG = "users"
    private val completedLessonsTag = "completedLessons"

    override fun updateLesson(lesson: Lesson, onErrorCallback: ErrorCallback) {
        UserService.user?.completedLessons?.let {
            if (!it.contains(lesson.id)) {
                it.add(lesson.id)
                insertInFirebase(it, onErrorCallback)
            }
        }
    }

    private fun insertInFirebase(it: ArrayList<String>,
                                 onErrorCallback: ErrorCallback) {
        FirebaseAuth.getInstance().currentUser!!.uid.apply {
            val reference = firebaseDatabase.getReference("$DATABASE_TAG/$this")
            reference
                    .child(completedLessonsTag)
                    .setValue(it)
                    .addOnFailureListener { exception ->
                        onErrorCallback(exception.toString())
                    }
        }
    }
}
```

Código 57: Implementación del caso de uso

Tiene una lógica muy similar al procedimiento de escritura de comentarios, *6.3.2*, con una particularidad: En este caso, al añadir la entrada a la base de datos, no se llama al método *push()*, esto evitará que se cree una entrada con una clave única generada por Firebase. En lugar de eso, se añadirá directamente el valor sin ninguna clave.

Por ejemplo, para la lección “lc1”, la entrada en la base de datos sería:

users/<id_usuario>/completedLessons/lc1

5.4 STORAGE

Para realizar pruebas, se han añadido dos vídeos a Firebase Storage:

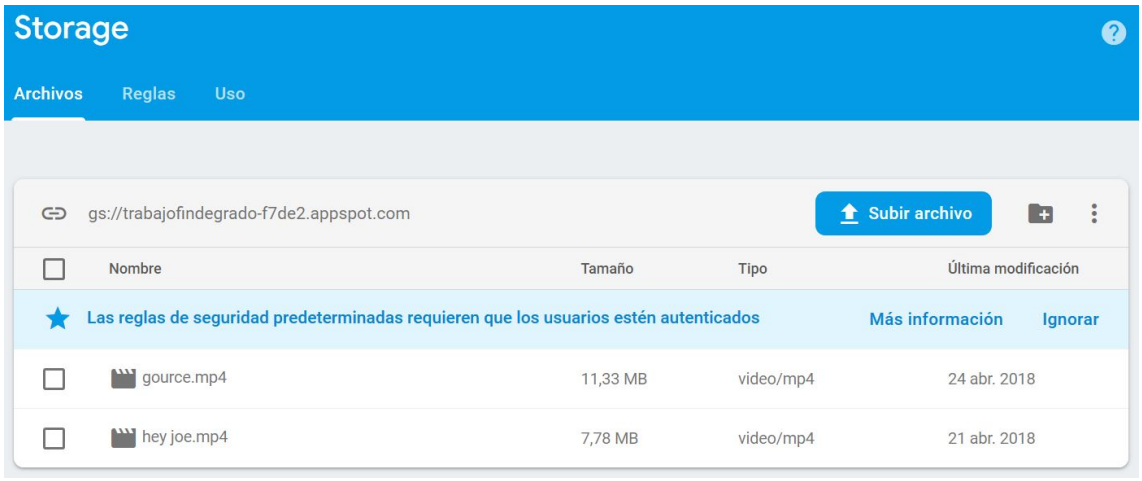


Ilustración 34: Pestaña de Storage en la consola de Firebase

El primero, es un montaje creado con la herramienta Gource [25], que muestra de forma visual la evolución de proyecto. El segundo, una versión propia de una canción popular de los años 60 [26].

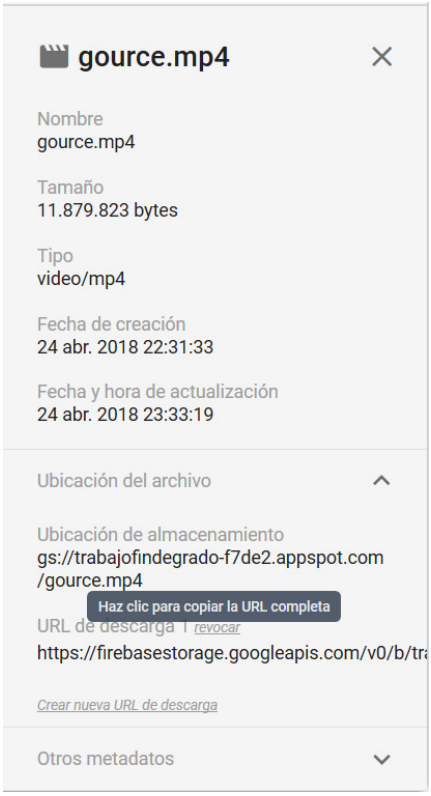


Ilustración 35: Obtención del enlace de descarga

El enlace de descarga de la ***ilustración 35*** se almacena en el campo video_path de la lección en la base de datos.

5.5 GOOGLE DRIVE

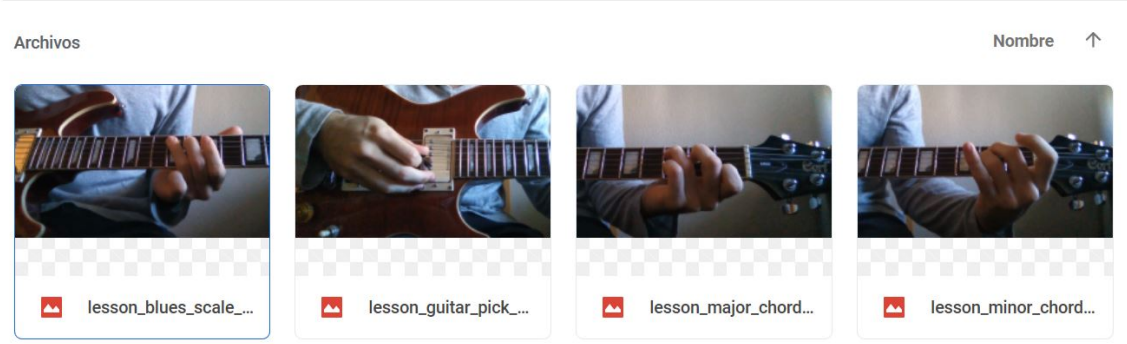


Ilustración 36: Imágenes de las lecciones

Debido a que Firebase Storage tiene limitaciones en almacenamiento y tráfico en la versión gratuita, se ha optado por alojar las imágenes en Google Drive.

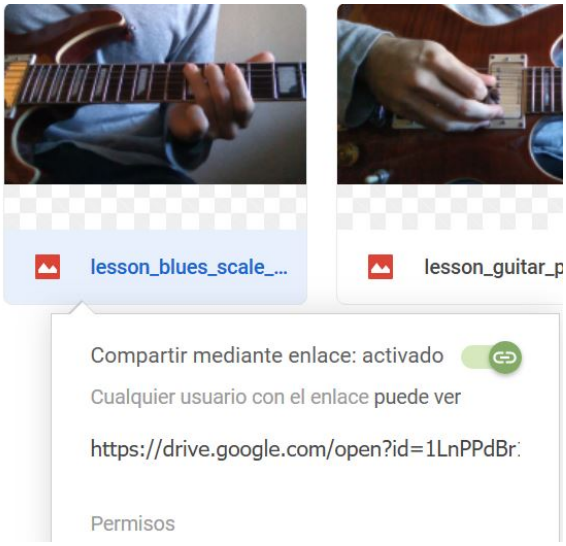


Ilustración 37: Enlace para compartir

El enlace de descarga de Drive se genera concatenando

```
https://drive.google.com/uc?export=download&id=
```

Con el campo id del enlace de compartir (***ilustración 37***).

Desde la aplicación, el enlace de descarga a las imágenes de Google Drive se obtiene del campo snapshot_src de la lección en la base de datos.

5.6 CRASHLYTICS

Crashlytics es una herramienta que permite recoger registros de errores ocurridos en instancias de la aplicación. Esto es, cualquier usuario que experimente un error -crash- en la aplicación, enviará dicha información al servidor.

Esto es especialmente útil de cara al mantenimiento de una aplicación tras salir a producción.

Para configurar Crashlytics en el proyecto, basta con ir a la sección en Firebase, optar por la característica y añadir las dependencias de librerías. No es necesario añadir código para comenzar a registrar datos.

Una vez añadida, se recomienda probarla. En caso de que la aplicación no tenga errores fáciles de reproducir, la forma más sencilla es forzándolo:

```
Crashlytics.getInstance().crash()
```

Código 58: Forzar un crash en la aplicación

Tras lanzar la aplicación, se cerrará y quedará notificado en la plataforma:

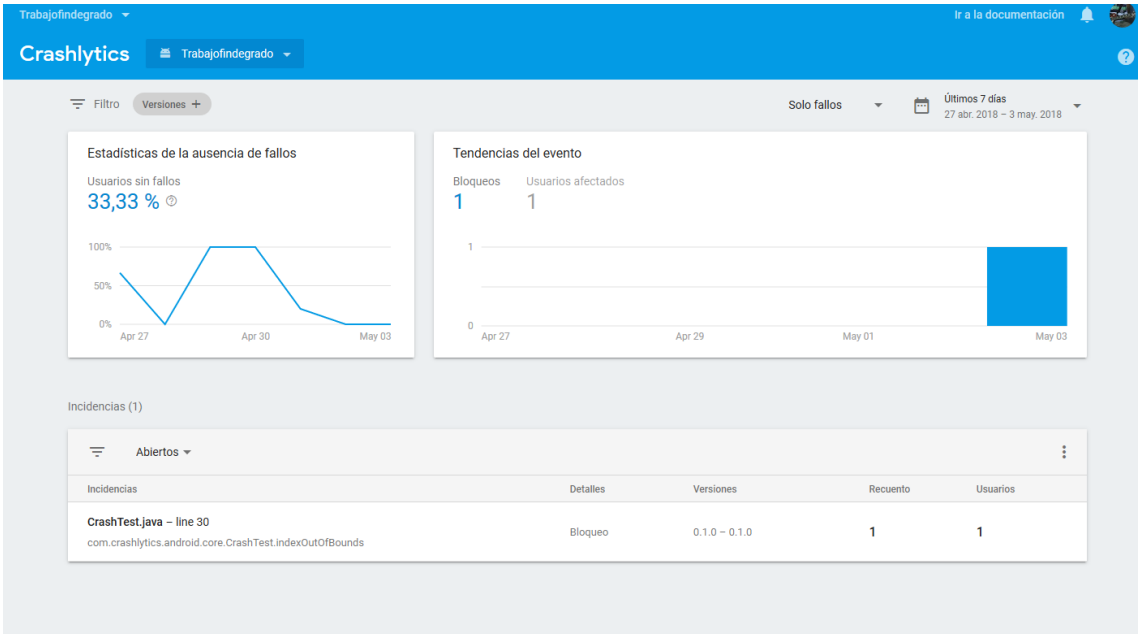


Ilustración 38: Pantalla de Crashlytics mostrando un crash

6. ARQUITECTURA

6.1 MODELO VISTA PRESENTADOR

Google no fuerza una estructuración rígida del código, por lo que a menudo se encuentran proyectos con toda la lógica dentro de las *Actividades* o *Fragments*. Esto hace que el código no sea verificable y difícil de mantener a largo plazo.

Modelo Vista Presentador o MVP es un patrón de arquitectura muy utilizado en aplicaciones con interfaces de usuario. La idea es separar la lógica de presentación de la de negocio y de los modelos de datos, teniendo como resultado tres capas:

- *Modelos* que definen los datos y la lógica de negocio.
- *Vistas* pasivas que notifican al *Presenter* las interacciones del usuario.
- *Presenters* que actúan como intermediarios y obtienen los datos (modelo) del *Repositorio* o *Casos de Uso* para procesarlos y actualizar la *Vista*.

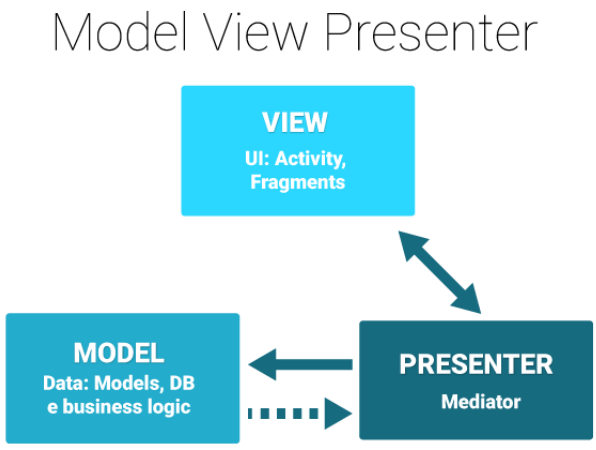


Ilustración 39: Diagrama de flujo MVP

Mediante *Contratos* (*Contracts*), se definen las interfaces de cada capa, lo cual permitirá inyectar diferentes implementaciones con facilidad y crear *mock* de la *Vista* independientes del contexto de Android para realizar tests unitarios.

```
interface BasePresenter: LifecycleObserver {
    @OnLifecycleEvent(Lifecycle.Event.ON_START)
    fun start()

    @OnLifecycleEvent(Lifecycle.Event.ON_STOP)
    fun stop()
}
```

Código 59: Presenter base con métodos dependientes del ciclo de vida

BasePresenter, utiliza anotaciones[9] del Arquitecture Component *lifecycle* para ejecutarse automáticamente en los eventos *onStart()* y *onStop()*.

En cuanto a un ejemplo de contrato, este es el que define la pantalla de lista de lecciones:

```
interface LessonsListContract {
    interface View {
        fun showProgressBar()
        fun hideProgressBar()
        fun setSectionTitle(sectionTitle: String)
        fun initializeCategoriesView()
        fun showMessage(message: String)
        fun navigateTo(intent: Intent)
    }

    interface Presenter: BasePresenter {
        fun setCategoryToRequest(categoryToRequest: String)
        fun setCategoryTitle(categoryTitle: String)
    }
}
```

Código 60: Ejemplo de contrato para Vista y Presenter

6.2 Casos de Uso

Siguiendo el esquema sacado del libro Clean Architecture [27] (***ilustración 40***) y complementando al diagrama de la ***ilustración 39***, están los casos de uso, o repositorios.

[9] https://docs.oracle.com/javase/tutorial/java/annotations/

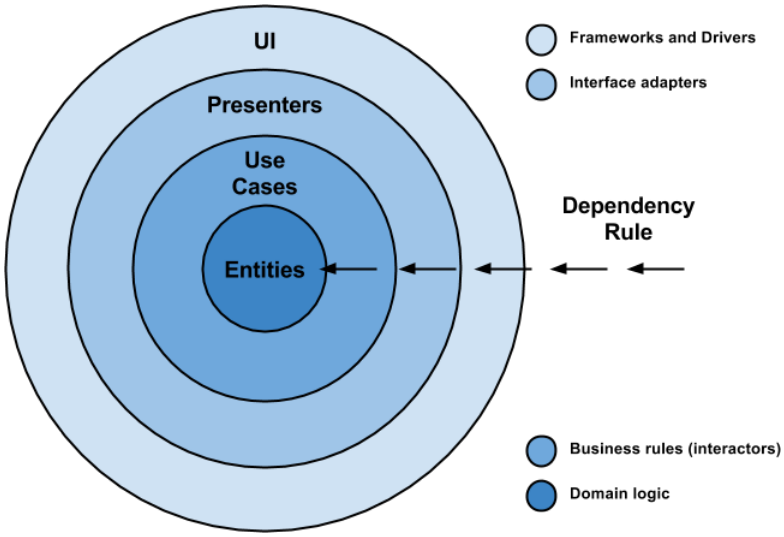


Ilustración 40: Capas arquitectura limpia

Los casos de uso realizan la lógica de negocio, esto es, lectura y escritura en base de datos, interacción con el backend, etcétera.

Del mismo modo que con los *Contracts*, se han usado interfaces para poder crear *mocks* de los mismos y de este modo poder realizar implementaciones ficticias y tests unitarios.

En la aplicación desarrollada, no se han utilizado bases de datos locales, por lo que todos los casos de uso interactúan con Firebase, a excepción de las implementaciones *mock*.

6.2.1 Ejemplo de Caso de Uso: Inicialización de la fuente de vídeo

El *Caso de Uso ObtainVideoSourceUseCase* se encarga de inicializar el stream del reproductor de video.

Éste recibe como parámetro la ruta del archivo a descargar y utiliza un *Listener* para devolver los objetos necesarios, siendo *mediaSource* el contenedor del stream y *bandwidthMeter* el estimador de ancho de banda.

```
interface ObtainVideoSourceUseCase {

    interface ObtainVideoCallback {
        fun onVideoSourceLoaded(videoSource: MediaSource, bandwidthMeter:
DefaultBandwidthMeter)
    }

    fun loadVideoSource(source: String, obtainVideoCallback:
ObtainVideoCallback)

}
```

Código 61: interfaz que define el caso de uso de descargar la fuente de vídeo

La implementación por defecto descargará el fichero .mp4 del backend (Firebase) a partir de la url:

```
class ObtainVideoSourceUseCaseImpl : ObtainVideoSourceUseCase {

    override fun loadVideoSource(
            source: String,
            obtainVideoCallback: ObtainVideoSourceCallback) {

        val uri = Uri.parse(source)
        val bandwidthMeter = DefaultBandwidthMeter()

        val dataSourceFactory =
DefaultHttpDataSourceFactory("exoplayer_agent")

        val videoSource = ExtractorMediaSource.Factory(
                dataSourceFactory).createMediaSource(uri)

        obtainVideoCallback.onVideoSourceLoaded(videoSource, bandwidthMeter)
    }
}
```

Código 62: Implementación por defecto de ObtainVideoSourceUseCase

Por otro lado, la implementación *mock* no hace uso de Firebase y utiliza un stream adaptativo DASH obtenida de un banco de vídeos de código abierto [28]. El video en concreto que se ha utilizado es Big Buck Bunny, de la fundación Blender [29].

```
class MockedObtainVideoSourceUseCaseImpl : ObtainVideoSourceUseCase {
    override fun loadVideoSource(source: String, obtainVideoCallback:
ObtainVideoSourceCallback) {

        val uri =
Uri.parse("http://www.bok.net/dash/bunny/cleartext/stream.mpd")
        val bandWidthMeter = DefaultBandwidthMeter()
        val manifestDataSourceFactory = DefaultHttpDataSourceFactory("ua")
        val dashChunkSourceFactory = DefaultDashChunkSource.Factory(
                DefaultHttpDataSourceFactory("ua", bandWidthMeter))
        val videoSource = DashMediaSource.Factory(dashChunkSourceFactory,
                manifestDataSourceFactory).createMediaSource(uri)

        obtainVideoCallback.onVideoSourceLoaded(videoSource, bandWidthMeter)
    }
}
```

Código 63: Implementación para pruebas del caso de ObtainVideoSourceUseCase

6.3 INYECCIÓN DE DEPENDENCIAS

La *Vista* tiene un *Presenter*, y éste tiene casos de uso. Estas relaciones se llaman dependencias.

Sin inyección de dependencias, el patrón normal a seguir sería que la *Vista* instanciase su propio *Presenter* y el *Presenter* sus *Casos de Uso*. Esto hace que el código sea poco modificable y muy difícil, de testear. El principio de inversión de dependencias [30], define que cualquier clase con dependencias, debe obtenerlas desde un contexto externo.

La librería más popular para inyectar dependencias en Android es Dagger [31], aunque en el caso de esta aplicación, se ha desarrollado un sistema propio más simple pero que permite justo la flexibilidad necesaria.

En el método de inyección de dependencias elaborado, las *Vistas* sólo conocen la interfaz del *Presenter*, y éstos, la interfaz de sus casos de uso, la implementación concreta la provee un objeto al que he llamado *Injection*.

En el caso de la *Vista*, la inyección es realizada de la siguiente forma:

```
class CategorySelectFragment : Fragment(), CategorySelectContract.View {

    private val presenter by lazy {
        Injection.provideCategoriesPresenter(this)
    }
    //...
```

Código 64: Obtención del Presenter desde la Vista

La inicialización *lazy* hará que el bloque de código se ejecute sólo cuando sea necesario. De este modo, si el *Fragment* se destruye en algún momento del ciclo de vida, el *Presenter* volverá a inicializarse. Esto elimina posibilidad de nulos e inicializaciones innecesarias.

Además, también es necesario añadir al *Presenter* como observador al ciclo de vida del *Fragment* para que pueda escuchar los eventos necesarios como se mencionó en la **sección 6.1**:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    lifecycle.addObserver(presenter)
}
```

Código 65: Se añade el Presenter como observador del ciclo de vida de la Vista

De esta forma, se evita tener que controlar el ciclo de vida del *Fragment* para llamar al *Presenter* en cada evento.

En cuanto a los casos de uso, son inyectados al *Presenter* desde el propio método que los crea:

```
object Injection {

    fun provideCategoriesPresenter(view: CategorySelectContract.View):
CategorySelectContract.Presenter =
            CategorySelectPresenter(view, ObtainCategoriesUseCaseImpl(),
ReadUserDataUseCaseImpl())

    //...
```

Código 66: Creación de un Presenter

Para ver en detalle cómo probar distintas implementaciones de *Presenter*, o con diferentes implementaciones de los casos de uso, ver la **sección 7.1**.

6.4 NAVEGACIÓN

Como se desveló ligeramente en el **apartado 4.3**, es necesario llamar a la función *showNewFragment* que fue añadida como extensión a la clase *FragmentActivity*. Pero los *Fragment* no pueden acceder a este de forma natural. Para evitar el acoplamiento y que los *Fragment* puedan ser agnósticos de la implementación de *Activity* en la que está, se han utilizado *LocalBroadcast*[10].

Para ello el primer paso es crear un receiver que realice la acción de navegar cuando recibe un mensaje.

```
private inner class NavigationReceiver : BroadcastReceiver() {

    override fun onReceive(context: Context, intent: Intent) {
        navigate(intent.action, intent.extras)
    }
}
```

Código 67: Navigation Receiver. Una implementación de BroadcastReceiver

El método *navigate* se encuentra definido a continuación:

[10] Permite transmitir Intents entre objetos de la aplicación.

```
private fun navigate(action: String?, args: Bundle? = null) {
    when (action) {
        CategorySelectFragment.TAG -> {
            navigateToFragment(
                    CategorySelectFragment.newInstance(),
                    false)
        }
        LessonsListFragment.TAG -> {
            navigateToFragment(LessonsListFragment.newInstance(args))
        }
        ExtendedInfoFragment.TAG -> {
            navigateToFragment(ExtendedInfoFragment.newInstance(args))
        }
        PlayerActivity.TAG -> {
            val intent = Intent(this, PlayerActivity::class.java)
                    .putExtra(Lesson.LESSON_KEY, args)
            startActivity(intent)
        }
        ProfileFragment.TAG -> {
            navigateToFragment(
                    ProfileFragment.newInstance(),
                    false)
        }
    }
}
```

Código 68: Lógica de navegación en la actividad

Donde *navigateToFragment* es una función intermedia que llama a *showNewFragment* con la id del contenedor *main_screen_container*.

El segundo parámetro de la función *navigateToFragment*, indica que no debe añadirse al *BackStack* y reiniciar la navegación, esto hará que se vacíe la cola y la pantalla mostrada se vuelva la raíz. De este modo, tras navegar a *CategorySelectFragment* o a *ProfileFragment* una pulsación del botón Back del dispositivo terminarán la ejecución de la aplicación.

Acto seguido, se crea un filtro de *intents* para registrar todas las acciones que llamarán al *Receiver* del bloque de ***código 67***. Por simplicidad, la acción para cada pantalla es un *TAG* con el nombre de la clase:

```
private fun setupNavigation() {
    val intentFilter = IntentFilter()
    intentFilter.addAction(ExtendedInfoFragment.TAG)
    intentFilter.addAction(CategorySelectFragment.TAG)
    intentFilter.addAction(LessonsListFragment.TAG)
    intentFilter.addAction(PlayerActivity.TAG)
    intentFilter.addAction(ProfileFragment.TAG)

    LocalBroadcastManager.getInstance(this)
            .registerReceiver(NavigationReceiver(), intentFilter)
}
```

Código 69: Añade el TAG de cada pantalla como acción al filtro

Tanto el *TAG* como los métodos necesarios para construir cada pantalla están definidos de forma estática (companion object en Kotlin) en cada *Fragment*:

```
class PlayerActivity : AppCompatActivity() {

    companion object {
        val TAG = PlayerActivity::class.java.simpleName

        fun getStartIntent(lesson: Lesson): Intent {
            val args = Bundle()
            args.putSerializable(Lesson.LESSON_KEY, lesson)
            return Intent(TAG).putExtras(args)
        }
    }
```

Código 70: companion object en PlayerActivity

Desde un *Fragment* cualquiera, se puede navegar a éste de la siguiente forma:

```
override fun onCardButtonClick() {
    val intent = PlayerActivity.getStartIntent(lesson)

    view.sendPlayerBroadcast(intent)
}
```

Código 71: Código ejecutado al notificar un click en Reproducir en LessonDetailsPresenter

```
override fun sendPlayerBroadcast(intent: Intent) {
    LocalBroadcastManager.getInstance(activity!!)
            .sendBroadcast(intent)
}
```

Código 72: La navegación se realiza desde LessonDetailsFragment al ser el que tiene dependencia de Activity

6.5 CREACIÓN DE LISTAS

Para crear listas, se ha utilizado el componente *RecyclerView* [32] de la librería de soporte. Éste es especialmente útil para gestionar de forma eficiente listas muy largas, ya que al hacer scroll, va reciclando los elementos que salen de la pantalla para generar los que entran y así se evitar tener instanciados en memoria más de los visibles. Los elementos básicos para utilizar un *RecyclerView* son:

- *Adapter*: provee las vistas que representan a cada ítem de una lista de datos.
- *Viewholder*: describe la vista de un ítem y los metadatos sobre su ubicación en el *RecyclerView*.
- *LayoutManager*: responsable de medir y posicionar las vistas y determinar cuándo reciclar.

Para evitar que el *adapter* conozca los modelos de datos, se ha creado una interfaz que permitirá mantener los modelos en el *Presenter* sin hacer que éste último tenga dependencias del contexto de Android:

```
interface RecyclerAdapterDelegate<E> {
    fun onBindViewHolder(holder: E, position: Int)
    fun getListSize(): Int
}
```

Código 73: Interfaz genérica para Presenters vinculados con Adapters de RecyclerView

Los *ViewHolder* implementarán una interfaz para bindear la información, por ejemplo, esta es la interfaz que implementa el *ViewHolder* de la lista de comentarios:

```
interface UserCommentView {
    fun bind(userName: String,
             messageBody: String,
             postDate: String,
             userPicturePath: String?)
}
```

Código 74: Ejemplo de interfaz a implementar en un ViewHolder

Los *Presenter* cuya *Vista* tengan una lista quedarán de la siguiente forma:

```
class CommentsPresenter(val view: CommentsContract.View,
                        private val obtainCommentsUseCaseImpl:
ObtainCommentsUseCase,
                        private val writeCommentUseCase:
WriteCommentUseCase) :
        CommentsContract.Presenter,
        RecyclerAdapterDelegate<UserCommentView>,
        LifecycleObserver {

    private lateinit var lesson: Lesson
    private val commentsList = arrayListOf<UserComment>()

    // ...

    override fun onBindViewHolder(holder: UserCommentView, position: Int) {
        with(mCommentsList[position]) {
            val parser = ISODateTimeFormat.dateTimeParser()
            val date = parser.parseDateTime(timeStamp)
            val postDate = date.toString()
            holder.bind(userName, content, postDate, userPicturePath)
        }
    }

    override fun getListSize(): Int = mCommentsList.size
}
```

Código 75: Ejemplo de Presenter vinculado a un RecyclerView

Y el código de la *Vista*:

```
class CommentsFragment : Fragment(), CommentsContract.View {
    private val presenter by lazy {
        Injection.provideCommentsPresenter(this, arguments)
    }

    // ...

    override fun initializeCommentsView() {
        comments_recyclerview.apply {
            adapter = CommentsRecyclerAdapter(
                    presenter as RecyclerAdapterDelegate<UserCommentView>
            )
            layoutManager = LinearLayoutManager(
                    context,
                    LinearLayoutManager.VERTICAL,
                    false)
        }
    }
}
```

Código 76: Inyectando dependencias del Presenter en el Adapter

El *adapter* hará uso del *Delegate* dentro de los métodos *getItemCount()* y *onBindViewHolder()*:

```
class CommentsRecyclerAdapter(
        private val commentsListDelegate:
RecyclerAdapterDelegate<UserCommentView>) :
        RecyclerView.Adapter<UserCommentViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
UserCommentViewHolder {
        val view = parent.inflate(R.layout.user_message)
        return UserCommentViewHolder(view)
    }

    override fun getItemCount(): Int =
            commentsListDelegate.getListSize()

    override fun onBindViewHolder(holder: UserCommentViewHolder, position:
Int) {
        commentsListDelegate.onBindViewHolder(holder, position)
    }
}
```

Código 77: Resultado final de un Adapter

Y el *ViewHolder* del comentario actualizará la *Vista* con los argumentos pasados por el *Presenter*:

```
class UserCommentViewHolder(itemView: View) :
        RecyclerView.ViewHolder(itemView),
        UserCommentView {

    override fun bind(userName: String, messageBody: String, postDate:
String, userPicturePath: String?) {

        with(itemView) {
            user_picture.loadProfileImageFromGlide(userPicturePath ?: "")
            text_author.text = userName
            text_content.text = messageBody
            text_date.text = postDate
        }
    }
}
```

Código 78: Resultado final de un ViewHolder

De esta manera, ni el *Presenter* tiene dependencias de Android, ni el *adapter*/*viewholder* tienen dependencias del modelo, por lo que se cumplen los principios de la arquitectura limpia.

6.6 Funciones de extensión

Kotlin permite añadir funciones a clases. Esto es especialmente útil para evitar herencia en ciertas situaciones en las que quedaría forzado o para añadir métodos sólo en algunos casos.

Muchas de las librerías de utilidades desarrolladas para Kotlin como KTX [33] o Anko [34] se basan en el uso de funciones de extensión.

En esta aplicación, se han desarrollado varias, y se han agrupado en cuatro ficheros: *ViewExtensions.kt*, para extender las *Vistas*; *TypeExtensions.kt*, con un método infijo para añadir elementos únicos a un *ArrayList*; y *ActivityExtensions.kt*, que contiene extensiones dependientes del contexto de Android.

6.6.1 ViewExtensions

```
fun ViewGroup.inflate(layoutRes: Int): View {
    return LayoutInflater.from(context).inflate(layoutRes, this, false)
}

fun ImageView.loadProfileImageFromGlide(url: String) {
    loadImageFromGlide(url, RequestOptions().circleCrop())
}

fun ImageView.loadImageFromGlide(url: String, options: RequestOptions =
RequestOptions()) {
    if(url.isNotEmpty() && url.isNotBlank()) {
        Glide.with(context).load(url).apply(options).into(this)
    }
}

fun hideViews(vararg views: View?) {
    views.forEach {
        it?.visibility = View.GONE
    }
}
```

Código 79: Funciones de extensión en ViewExtensions.kt

6.6.2 TypeExtensions

```
infix fun <E>ArrayList<E>.addUnique(t: E) {
    if(!contains(t)) add(t)
}
```

Código 80: Función infija para añadir elementos no duplicados

El uso de funciones infijas queda así:

```
commentsList.forEach {
    mCommentsList addUnique it
}
```

Código 81: Uso de addUnique

No tienen ventajas respecto a usar funciones normales, pero hacen las sentencias más fáciles de leer y añaden expresividad al evitar puntos y paréntesis.

6.6.3 ActivityExtensions

```
un FragmentActivity.setHomeButtonEnabled(enabled: Boolean) {
    (this as AppCompatActivity).supportActionBar?.apply {
        setHomeButtonEnabled(enabled)
        setDisplayHomeAsUpEnabled(enabled)
    }
}

fun FragmentActivity.setFullScreenOn() {
    window?.apply {
        clearFlags(WindowManager.LayoutParams.FLAG_FORCE_NOT_FULLSCREEN)
        addFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN)
    }
}

fun FragmentActivity.setFullScreenOff() {
    window?.apply{
        addFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN)
        addFlags(WindowManager.LayoutParams.FLAG_FORCE_NOT_FULLSCREEN)
    }
}

fun FragmentActivity.signOut() {
    AuthUI.getInstance().signOut(this)
}

fun FragmentActivity.broadcastTo(intent: Intent) {
    LocalBroadcastManager.getInstance(this)
            .sendBroadcast(intent)
}

fun FragmentActivity.showToast(str: String) {
    Toast.makeText(this, str, Toast.LENGTH_SHORT).show()
}

inline fun FragmentManager.replaceContent(placeHolder: Int,
                                          fragment: Fragment,
                                          transactionBody:
FragmentTransaction.() -> Unit = {}) {
    beginTransaction()
            .apply(transactionBody)
            .replace(placeHolder, fragment)
            .commit()
}
```

Código 82: ActivityExtensions.kt

6.6.4 Typealiases

Además, Kotlin permite añadir alias para facilitar el uso de lambdas[11] o acortar nombres y hacer algunas llamadas más legibles:

[11] Función anónima, sirve para definir argumentos que son pasados a otras funciones o para manejar el resultado obtenido de llamar a otra función.

```
typealias ClickListener = () -> Unit

typealias ErrorCallback = (String) -> Unit

typealias ObtainCategoriesCallback =
ObtainCategoriesUseCase.ObtainCategoriesCallback

typealias ObtainCommentsCallback =
ObtainCommentsUseCase.ObtainCommentsCallback

typealias ObtainVideoSourceCallback =
ObtainVideoSourceUseCase.ObtainVideoCallback

typealias BottomSheetCallback = BottomSheetBehavior.BottomSheetCallback

typealias ReadUserCallback = ReadUserDataUseCase.ReadUserCallback
```

Código 83: TypeAliases.kt

6.7 LIBRERÍAS

A lo largo del proyecto, se han utilizado ciertas librerías para añadir funcionalidades.

En primer lugar, la de soporte, para asegurar que la aplicación sea compatible con versiones más antiguas de Android. Los módulos utilizados han sido los siguientes:

```
implementation "com.android.support:appcompat-v7:$support_version"
implementation "com.android.support:cardview-v7:$support_version"
implementation "com.android.support.constraint:constraint-layout:$
constraint_layout_version"
```

Código 84: Librerías de soporte

Donde

- **appcompat** incluye la clase de la que extienden las activities utilizadas, *AppCompatActivity*.
- **cardview** es el componente utilizado para dar estilo a las listas de categorías y lecciones.
- **constraint-layout** permite el uso de *ConstraintLayout*, que es un tipo de layout más eficiente que *RelativeLayout* y permite crear vistas complejas sin necesidad de anidar múltiples layouts.

Kotlin:

```
implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
```

Código 85: Soporte para Kotlin

Para descargar imágenes el backend y mostrarlas en un *ImageView*, se ha hecho uso de **Glide** [35], cuya implementación puede verse en la **sección 6.6.1**:

```
implementation "com.github.bumptech.glide:glide:$glide_version"
kapt "com.github.bumptech.glide:compiler:$glide_version"
```

Código 86: Dependencias de Glide

Threeten [22], para poder convertir cadenas de texto con fechas siguiendo el ISO_8601:

```
implementation "org.threeten:threetenbp:$threeten_version"
```

Código 87: Librería Threeten

Como ya se vio en la **sección 4.3**, los tres módulos utilizados de **ExoPlayer**:

```
implementation "com.google.android.exoplayer:exoplayer-core:$
exoplayer_version"
implementation "com.google.android.exoplayer:exoplayer-
ui:$exoplayer_version"
implementation "com.google.android.exoplayer:exoplayer-dash:$
exoplayer_version"
```

Código 88: Dependencias de ExoPlayer

Servicios de Google Play necesarios para utilizar Firebase, **módulos de Firebase** utilizados y **FirebaseUI-auth**:

```
implementation "com.google.android.gms:play-services:$google_version"
implementation "com.google.firebase:firebase-appindexing:$google_version"
implementation "com.google.firebase:firebase-database:$google_version"
implementation "com.google.firebase:firebase-core:$google_version"
implementation "com.google.firebase:firebase-auth:$google_version"
// Firebase UI
implementation "com.firebaseui:firebase-ui-auth:$firebase_ui_version"
```

Código 89: PlayServices y Firebase

Para utilizar **Crashlytics**, es necesario incluir también:

```
implementation
"com.crashlytics.sdk.android:crashlytics:$crashlytics_version"
```

Código 90: SDK de Crashlytics

Por último, para testing, se han incluido varias librerías, tanto para tests unitarios como de instrumentación:

```
// Local unit tests
testImplementation "org.mockito:mockito-all:$mockitoVersion"
testImplementation "junit:junit:$junitVersion"
testImplementation "org.hamcrest:hamcrest-all:$hamcrestVersion"
testImplementation "org.powermock:powermock-module-junit4:$powerMockito"
testImplementation "org.powermock:powermock-api-mockito:$powerMockito"

// Android Testing Support Library's runner and rules
androidTestImplementation "com.android.support.test:runner:$runnerVersion"
androidTestImplementation "com.android.support.test:rules:$rulesVersion"

// Espresso UI Testing.
androidTestImplementation "com.android.support.test.espresso:espresso-
contrib:$espressoVersion"
androidTestImplementation "com.android.support.test.espresso:espresso-
intents:$espressoVersion"
androidTestImplementation("com.android.support.test.espresso:espresso-
core:$espressoVersion", {
    exclude group: 'com.android.support', module: 'support-annotations'
})
```

Código 91: Librerías para testing

El número de versión compilado de las librerías se encuentra externamente, en el fichero de configuración gradle a nivel de proyecto:

```
buildscript {
    ext.kotlin_version = '1.2.41'
    ext.google_version = '12.0.1'
    ext.support_version = '27.1.1'
    ext.exoplayer_version = '2.7.3'
    ext.junitVersion = '4.12'
    ext.espressoVersion = '2.2.2'
    ext.mockitoVersion = '1.10.19'
    ext.powerMockito = '1.6.2'
    ext.hamcrestVersion = '1.3'
    ext.runnerVersion = '0.5'
    ext.rulesVersion = '0.5'
    ext.glide_version = '4.3.1'
    ext.lifecycle_version = "2.0.0-alpha1"
    ext.constraint_layout_version = '1.1.2'
    ext.firebase_ui_version = '3.3.0'
    ext.crashlytics_version = '2.9.3'
    ext.threeten_version = '1.3.6'
    // ...
}
```

Código 92: Número de versión de las librerías utilizadas

Esto tiene varias ventajas respecto a definir el número en directamente en el gradle del módulo de aplicación. Primero, permite obtener una visión general de las librerías utilizadas de forma mucho más rápida. Segundo, en caso de tener una aplicación multi-modular con dependencias compartidas, se unifica la versión y se evitan posibles problemas por actualizar un módulo y olvidar el otro.

7. TESTING

7.1 GRADLE Y BUILD VARIANTS

Gradle es un sistema de automatización para construir proyectos y gestionar dependencias.

En una aplicación desarrollada en Android Studio hay al menos dos ficheros de configuración de Gradle: uno a nivel de proyecto, y otro a nivel de módulo.

En el que está a nivel de proyecto se definen opciones de configuración utilizadas en todos los subproyectos/módulos, los repositorios utilizados para importar librerías y las dependencias de los scripts de Gradle.

En el que está definido a nivel de aplicación, se declaran configuraciones de Android. Entre ellas, la mínima versión de SDK compatible con la aplicación, el código de versión, los tipos de *build*[12], las dependencias de librerías de la aplicación, los *flavors*[13], etcétera.

En esta aplicación, se ha hecho uso de estos para definir una configuración de producción y una de test (*mock*) y así poder cambiar todo el entorno de la aplicación con un de click. Ésto se ha conseguido gracias a la inyección de dependencias: las dependencias se obtienen del objeto *Injection*, y basta con replicar este objeto con diferente código en cada flavor.

Para utilizar flavors, el primer paso es crearlos en el gradle del módulo:

```
productFlavors {

    production {
    }

    mock {
        applicationIdSuffix = ".mock"
    }
}
```

Código 93: Declaración de Flavors en Gradle

[12] Configuraciones del proyecto con diferentes parámetros. Por defecto hay una de debug y otra de release. Lo habitual es añadir trazas de consola para obtener información durante el desarrollo en la configuración de debug y desactivarlas en release.
[13] Variantes de compilación para configurar, por ejemplo, una versión gratuita y otra de pago.

Donde el applicationIdSuffix es un campo para diferenciar la id de la aplicación y poder identificarla por separado en Firebase.

En segundo lugar, se crean directorios con los nombres de cada *flavor* bajo la carpeta "src", junto a 'main' (directorio por defecto). Tienen la misma estructura que main: tienen un directorio 'java' para código y uno 'res' para recursos de Android.

Cualquier clase que se encuentre en el directorio de un *flavor*, sustituirá a una clase del mismo nombre que se encuentre en 'main', mientras que ficheros como el *AndroidManifest*, se combinan aplicando sólo las diferencias del *flavor*.

El código de un *flavor* sólo está disponible cuando éste es activado desde la pestaña de Build Variants.

El número total de Build Variants será la combinación de cada tipo de configuración con cada *flavor*. Para evitar tener combinaciones inservibles, se puede utilizar la siguiente función:

```
variantFilter { variant ->
    if (variant.buildType.name.equals('release')
            && variant.getFlavors().get(0).name.equals('mock')) {
        variant.setIgnore(true);
    }
}
```

Código 94: Uso de la función variantFilter

Así, se limitan las opciones mostradas. En este caso, se elimina la posibilidad de generar por error una versión de producción (release) con *mocks*, ya que el propósito de este *flavor* es el de realizar pruebas.

El resultado final de la estructura del proyecto tras aplicar los pasos anteriores es el siguiente:

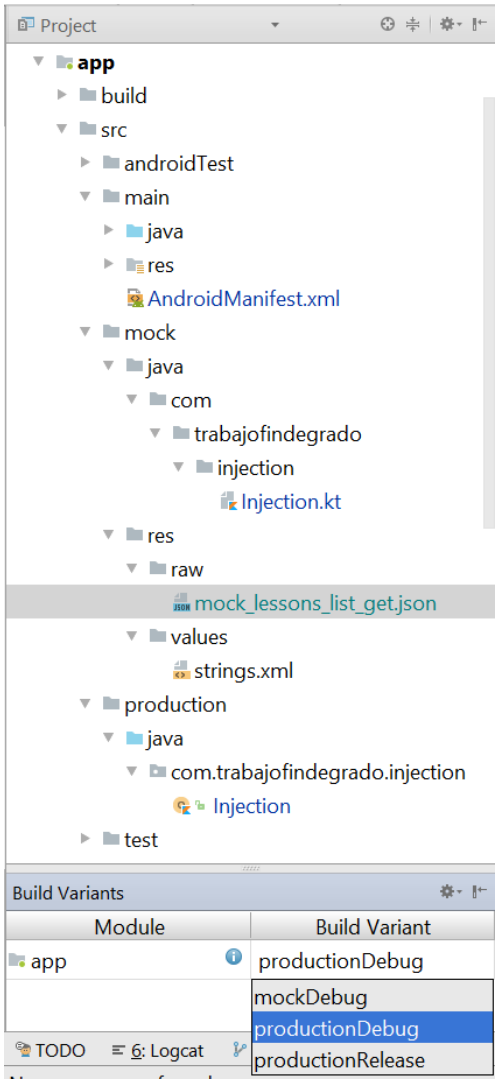


Ilustración 41: Vista de proyecto y pestaña para elegir BuildVariant

Al seleccionar una variante de tipo "mock" o "production", se usará el objeto *Injection* de cada flavor respectivamente.

7.2 TESTS UNITARIOS: JUNIT Y MOCKITO

Por un lado, están los **tests unitarios**, esto es, tests locales, que corren directamente en la JVM y son independientes del contexto de Android. Para garantizar esta independencia es necesaria la librería Mockito [36] y tener una arquitectura con las capas correctamente separadas.

Este tipo de tests es útil porque evitan la carga de arrancar una máquina física o emulador, construir el instalable e instalarlo, pueden ejecutarse localmente.

Mockito es una librería muy popular de Java para crear *mocks* y permitir un desarrollo orientado a tests (TDD [37]). En la propia documentación de Android [38] la incluyen como librería opcional.

En la aplicación, se han desarrollado varios tests. A modo de ejemplo, se explicará el funcionamiento de los tests y las ventajas de incluir Mockito con el *Presenter* de la pantalla de lecciones, *LessonsListPresenter*.

Al iniciar la *Vista* de lecciones, se muestra una barra circular de carga (*ProgressBar*), se inicializan los componentes de UI y se realiza una llamada asíncrona al backend para descargar la lista de lecciones para la categoría de la pantalla.

Una vez termina de descargar los datos, oculta la barra de carga y muestra la lista, o en su defecto un mensaje de error si hubo algún problema descargando el contenido.

En caso de hacer click en una lección de la lista, se navegará a la pantalla de lección.

El contrato de la pantalla es el siguiente:

```
interface LessonsListContract {

    interface View {
        fun showProgressBar()
        fun hideProgressBar()
        fun setSectionTitle(sectionTitle: String)
        fun initializeCategoriesView()
        fun showMessage(message: String)
        fun navigateTo(intent: Intent)
    }

    interface Presenter: BasePresenter {
        fun setCategoryToRequest(categoryToRequest: String)
        fun setCategoryTitle(categoryTitle: String)
    }
}
```

Código 95: Contract de LessonsList

El código del *Presenter* relevante para la prueba se encuentra a continuación:

```
class LessonsListPresenter(private val mView: LessonsListContract.View,
                           private val obtainCategoriesUseCase:
ObtainCategoriesUseCase) :
        LessonsListContract.Presenter,
        RecyclerAdapterDelegate<LessonView> {

    private lateinit var categoryToRequest: String
    private lateinit var categoryTitle: String
    private val lessonsList = arrayListOf<Lesson>()

    override fun start() {
        mView.setSectionTitle(categoryTitle)
        mView.showProgressBar()
        mView.initializeCategoriesView()
        loadContents()
    }

    override fun stop() {
        obtainCategoriesUseCase.stop()
    }

    override fun setCategoryToRequest(categoryToRequest: String) {
        this.categoryToRequest = categoryToRequest
    }

    override fun setCategoryTitle(categoryTitle: String) {
        this.categoryTitle = categoryTitle
    }


    //...
```

Código 96: Código de la clase LessonsListPresenter

El *Caso de Uso* que utiliza el *Presenter* tiene la siguiente interfaz:

```
interface ObtainCategoriesUseCase {
    interface ObtainCategoriesCallback {
        fun onContentLoaded(categoryContainers: List<CategoryContainer>)
        fun onError(error: String)
    }

    fun loadCategoryContainers(targetFilter: String? = null,
                               callback: ObtainCategoriesCallback)

    fun stop()
}
```

Código 97: La interfaz ObtainCategoriesUseCase

Si se quisiera testear que todo funciona correctamente sin necesidad de implementar la *Vista*, bastaría con crear un *mock* a partir del *Contract* de la misma. Como se muestra en el siguiente bloque de código:

```
public class LessonsListPresenterTest {
    @Mock private LessonsListContract.View mView;
    @Mock private ObtainCategoriesUseCase obtainCategoriesUseCase;
    @Captor private
ArgumentCaptor<ObtainCategoriesUseCase.ObtainCategoriesCallback>
            callbackArgumentCaptor;

    private LessonsListPresenter lessonsListPresenter;
    private String categoryTitle = "CATEGORY";
    private String categoryToRequest = "cat";

    @Before
    public void setupLessonsMainPresenter() {
        MockitoAnnotations.initMocks(this);

        lessonsListPresenter = new LessonsListPresenter(mView,
obtainCategoriesUseCase);
        lessonsListPresenter.setCategoryTitle(categoryTitle);
        lessonsListPresenter.setCategoryToRequest(categoryToRequest);
    }
    //...
```

Código 98: Declaración de variables e inicialización previa del test de LessonsListPresenter

Donde:

- @Mock: anota una interfaz que va a funcionar como *mock*.
- @Captor: anota una interfaz que va a capturar una llamada para ser manipulada.
- @Before: será llamado antes de cada @Test, ya que puede haber múltiples test en una clase. En este caso, inicializa los *mocks* y crea el *Presenter*.

```
@Test
public void categoriesRead() {
    ArrayList<Lesson> lessons = new ArrayList<>();
    lessons.add(new Lesson());
    lessons.add(new Lesson());
    Category category = new Category(categoryToRequest, categoryTitle, 0, 0);
    List<CategoryContainer> categoryContainers =
            Collections.singletonList(new CategoryContainer(lessons,
category, 1));

    lessonsListPresenter.start();

    verify(mView).showProgressBar();
    verify(obtainCategoriesUseCase)
            .loadCategoryContainers(eq(categoryToRequest),
                    callbackArgumentCaptor.capture());

    callbackArgumentCaptor.getValue().onContentLoaded(categoryContainers);

    verify(mView).initializeCategoriesView();
    verify(mView).hideProgressBar();
}
```

Código 99: Test que comprueba la lectura de categorías

Este test, prueba que las categorías se leen correctamente. Para ello, crea una categoría falsa con dos lecciones.

Inicia el *Presenter* y comprueba, mediante la función "verify", que los métodos *showProgressBar* y *loadCategoryContainers* se han llamado. En caso de que algún "verify*"* devolviera "false", el test fallaría.

En la llamada a *loadCategoryContainers*, se llama al método *capture()* del ArgumentCaptor. Al capturarlo, se puede llamar a uno de sus métodos, en este caso, onContentLoaded con la lista creada al inicio.

Finalmente, se comprueba con otros dos "verify" que se llaman los métodos que se deberían llamar en caso de que al *Presenter* le llegase un *onContentLoaded*.

El otro test, comprueba lo que ocurre cuando se produce un error:

```
@Test
public void emptyCategoryContainers() {
    lessonsListPresenter.start();

    verify(mView).showProgressBar();
    verify(obtainCategoriesUseCase)
            .loadCategoryContainers(eq(categoryToRequest),
                    callbackArgumentCaptor.capture());

    callbackArgumentCaptor.getValue().onError("error");

    verify(mView).showMessage(eq("error"));
}
```

Código 100: Test de una lista vacía

En el código completo de la aplicación, también se puede estudiar un test de la inicialización del player, *PlayerControlsPresenterTest*.

7.3 TEST DE INSTRUMENTACIÓN: ESPRESSO

Por otro lado, están las pruebas de UI, o **de instrumentación**, que corren directamente en un dispositivo Android o emulador y sirven para comprobar que los elementos de las pantallas funcionan, que determinados elementos se muestran o no, o que la navegación es correcta.

Para este apartado, se hará uso de código antiguo, anterior a la integración de la librería Firebase-UI. Así, se verá al mismo tiempo la librería y la utilización de un *Caso de Uso mock* con el propósito de probar la UI sin depender de Firebase.

Las pruebas de instrumentación son útiles para, tras un cambio importante en la lógica de la aplicación, probar de forma automatizada que el comportamiento esperado de la aplicación no ha sido alterado. En los test de ejemplo a continuación, se comprueba que

distintas interacciones con el formulario de registro e inicio de sesión devuelven el resultado esperado.

La pantalla que se va a probar es la siguiente:

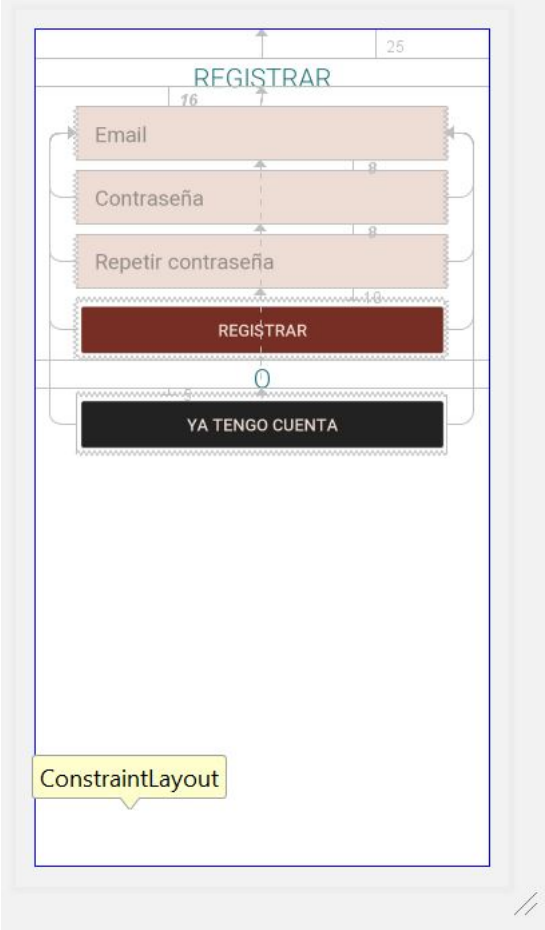


Ilustración 42: Pantalla de registro vista desde el editor de AndroidStudio

Cuando se pulsa el botón *Registrar*, se realiza una serie de comprobaciones.

- El primer campo es un email.
- Los campos de contraseña no están vacíos.
- La contraseña tiene al menos 6 caracteres.
- El campo de repetir contraseña tiene le mismo contenido que el de contraseña.
- El email introducido no existe en la base de datos de firebase.

En caso de que alguno de estos puntos no se cumpla, aparecerá una ventana indicando el tipo de error. Los mensajes de error son los siguientes:

```
<!-- Error messages -->
<string name="error_empty_email">Email vacío</string>
<string name="error_not_an_email">No es una dirección de correcto
válida</string>
<string name="error_password_empty">Debes introducir una contraseña</string>
<string name="error_password_incorrect">Contraseña incorrecta</string>
<string name="error_user_not_exist">El usuario no existe</string>
<string name="error_other">Se ha producido un error en el inicio de
sesión</string>
<string name="error_other_register">Se ha producido un error en el proceso de
registro</string>
<string name="error_password_short">La contraseña es demasiado corta. Por
favor introduce una contraseña con mínimo 6 caracteres</string>
<string name="error_password_missmatch">Las contraseñas no coinciden</string>
<string name="error_user_already_exists">Esa dirección ya está
registrada</string>
```

Código 101: Cadenas de error en el fichero strings.xml

Todas las cadenas de texto de la aplicación están externalizadas en el fichero strings.xml. Así, localizar la aplicación a diferentes idiomas es tan sencillo como añadir este fichero traduciendo el contenido en el correspondiente directorio values-*xx* [39].

Para simular el comportamiento de la pantalla sin necesitar Firebase, se ha creado una implementación *mock* del *Caso de Uso*:

```
class MockRegisterWithEmailUseCase()
    : RegisterUserUseCaseInterface {

    companion object {
        const val RIGHT_USER = "rightUser@gmail.com";
        const val RIGHT_PASSWORD = "rightPassword";
        const val WRONG_USER = "wrongUser"
        const val REPEATED_USER = "repeatedUser@gmail.com";
        const val WRONG_PASSWORD = "wrongPassword"
    }

    override fun createUser(user: String,
                            password: String,
                            onRegisterUserCallback: OnRegisterUserCallback) {
        if (user == RIGHT_USER && password == RIGHT_PASSWORD) {
            onRegisterUserCallback.onRegisterSuccess()
        } else {
            onRegisterUserCallback.onRegisterError(if(user == REPEATED_USER) {
                RegisterUserUseCaseInterface.ERROR_ALREADY_IN_USE
            } else {
                "Unknown error"
            })
        }
    }
}
```

Código 102: Caso de uso mock con constantes para el test

Éste se limita a comparar las cadenas recibidas con las constantes conocidas para notificar el éxito o devolver distintos mensajes de error.

El flujo de un test de Espresso [40] es el siguiente:

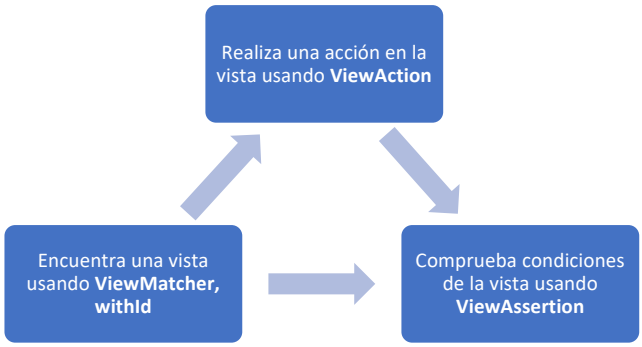


Ilustración 43: Flujo de acciones en un test de Espresso

Para crear test de UI, en primer lugar, creo una clase base de la que extenderán los demás. Aquí, se especifica la actividad en la que se ejecutarán las pruebas y defino las constantes necesarias.

```
@RunWith(AndroidJUnit4.class)
public class BaseInstrumentationTest {
    @Rule
    public ActivityTestRule<SigninActivity> mActivityRule =
            new ActivityTestRule<>(SigninActivity.class);

    public final String RIGHT_USER =
MockedRegisterWithEmailUseCase.Companion.getRIGHT_USER();
    public final String WRONG_USER =
MockedRegisterWithEmailUseCase.Companion.getWRONG_USER();
    public final String REPEATED_USER =
MockedRegisterWithEmailUseCase.Companion.getREPEATED_USER();
    public final String PASSWORD =
MockedRegisterWithEmailUseCase.Companion.getRIGHT_PASSWORD();
    public final String WRONG_REPEAT_PASSWORD =
MockedRegisterWithEmailUseCase.Companion.getWRONG_PASSWORD();
}
```

Código 103: Test base con la regla de la Activity de registro y constantes

Donde @Rule es una anotación que hará que se ejecuten los métodos *onCreate*, *onStart* y *onResume* del ciclo de vida de dicha *Activity*. Se ejecuta antes de @Before.

ActivityTestRule define la *Activity* de inicio, por lo que se puede testear cualquier *Activity* de forma independiente ignorando la navegación. Esto agiliza los test de instrumentación en aplicaciones de múltiples actividades.

El primer test de *RegisterTest*, probará introducir un valor incorrecto en el campo de email:

```
@RunWith(AndroidJUnit4.class)
public class RegisterTest extends BaseInstrumentationTest {

    @Test
    public void testNotEmail() {
        // Email incorrecto
        onView(withId(R.id.edit_email_register))
                .perform(typeText(WRONG_USER), closeSoftKeyboard());
        onView(withId(R.id.button_register))
                .perform(click());
        onView(withText(R.string.error_not_an_email))
                .check(matches(isDisplayed()));
    }
    //...
```

Código 104: Primer test de la pantalla de registro

- onView: busca una vista en el layout actual.
- withId: matcher para buscar una vista a partir del campo id.
- perform: realiza una serie de N interacciones, en orden.
- typeText: escribe una cadena en el campo.
- closeSoftKeyboard: cierra el teclado. De este modo se evita que oculte otros elementos de la pantalla.
- click: realiza un click.
- withText: matcher para buscar una vista que contenga el texto del argumento.
- check: comprueba una aserción. En este caso, que la vista encontrada sea visible (matches(isDisplayed()).

El test fallará tanto si se intenta realizar una acción sobre una vista no encontrada como si una interacción *check* no se cumple.

A continuación, se muestra un test un poco más elaborado que comprueba que la confirmación de contraseña es correcta:

- Escribe los campos correctos en email y contraseña.
- No escribe nada en la comprobación de contraseña.
- Realiza click en el botón de registro.
- Comprueba que se muestra el popup de error y tiene el texto correcto.
- Cierra el popup.
- Escribe el campo incorrecto en la confirmación de contraseña.
- Realiza click en el botón de registro.
- Comprueba de nuevo que se muestra el popup de error correcto.

```
@Test
public void testPasswordMissmatch() {
    // Validación password vacía
    onView(withId(R.id.edit_email_register))
            .perform(clearText(), typeText(RIGHT_USER), closeSoftKeyboard());
    onView(withId(R.id.edit_password_register))
            .perform(clearText(), typeText(PASSWORD), closeSoftKeyboard());
    onView(withId(R.id.button_register))
            .perform(click());
    onView(withText(R.string.error_password_missmatch))
            .check(matches(isDisplayed()));
    onView(withText("OK"))
            .perform(click());

    // Validación password incorrecta
    onView(withId(R.id.edit_repeat_password_register))
            .perform(clearText(), typeText(WRONG_REPEAT_PASSWORD),
closeSoftKeyboard());
    onView(withId(R.id.button_register))
            .perform(click());
    onView(withText(R.string.error_password_missmatch))
            .check(matches(isDisplayed()));
}
```

Código 105: Test de validación de contraseña incorrecta

8. PLANIFICACIÓN Y PRESUPUESTO

8.1 PRESUPUESTO

Para la elaboración de este proyecto, se ha hecho uso de los siguientes equipos, con su correspondiente precio:

Ordenador	**Precio**
Portátil Asus UX430UA i5 7300U, 8GB RAM	744€
Sobremesa AMD Ryzen 1400, 16GB RAM	648€
Total:	**1392€**

Para pruebas, se han utilizado varios dispositivos Android:

Dispositivo	**Precio**
Motorola Moto G1	140€
BQ Aquaris U Plus	175€
LG G6	329€
Total:	**644€**

El tiempo total de desarrollo ha sido de aproximadamente 9 meses, compaginados con un trabajo a jornada completa. En promedio, he dedicado 2 horas al día al proyecto. Redondeando un mes a 30 días, el número total es de 540h.

El salario base para un ingeniero con mi experiencia es de aproximadamente 29.000,0€ brutos al año, o lo que es lo mismo, 14€/h.

Por otro lado, se sumarán 35€/h por cada hora de tutelaje de la tutora de este proyecto, María Celeste Campo Vázquez.

Personal	**Tiempo (horas)**	**Salario/hora (bruto)**	**Coste**
Programador Junior con 2 años de experiencia	540h	14€	7.560€
Tutora	20h	35€	700€
Total:	-	-	**8260€**

Por tanto, el coste total del proyecto hasta la fecha ha sido de 10296€.

8.2 EXPLOTACIÓN Y MONETIZACIÓN

La aplicación desarrollada sólo es una prueba de concepto, sin embargo, se han estudiado distintas posibilidades para dar rentabilidad al proyecto en una versión definitiva.

La forma más habitual de monetizar aplicaciones Android es mediante marcos publicitarios en los márgenes o ventanas emergentes. Esta opción queda descartada por ser demasiado intrusiva y estropear la experiencia de usuario.

Otro método es el de dividir entre una versión gratuita con limitaciones y una de pago. En cuanto a las limitaciones, hay dos posibilidades; limitar características, o limitar contenidos.

Una alternativa más sería la de partir de un contenido base de partida y ofrecer packs de contenido totalmente opcionales.

La opción de ofrecer un contenido básico ampliable con packs de contenido parece la óptima por varios motivos:

- Permite atraer a más usuarios a adquirir contenido de pago gracias a que cada pack tendría un precio menor que el de una aplicación completa.
- En la arquitectura de la aplicación desarrollada, es posible añadir contenido nuevo en el lado del servidor sin necesidad de cambiar el código del cliente, por lo que se evitaría forzar a los usuarios a actualizar.
- En el caso de usuarios que adquieran contenido de forma regular, los ingresos a largo plazo serían mayores que si la aplicación se adquiriese mediante pago único.

9. CONCLUSIONES

9.1 OBJETIVOS CUMPLIDOS

El objetivo técnico era que la aplicación fuera lo suficientemente madura para servir como prueba de concepto. Para ello, basta con comparar si los requisitos definidos en el capítulo **Diseño y Requisitos** han sido conseguidos a lo largo de los capítulos posteriores.

Requisito	Implementado
Sección de lecciones ordenadas por categoría y dificultad	Sí
Las lecciones tienen explicación teórica y video	Sí
Se ha implementado un reproductor de video	Sí
Los usuarios pueden comentar en las lecciones	Sí
Sección de práctica con algoritmo de detección de acordes a través del micrófono	No
Sección de perfil con estadísticas personales	Parcialmente*
Gestión de usuarios y autenticación para guardar sus progresos	Sí
Utilización de principios de arquitectura limpia	Sí
Basado en un patrón de diseño arquitectural	Sí
Compatible con tecnología de streaming adaptativo	Sí
Sigue las pautas de Material Design	Sí
Tests de funcionamiento de código	Sí

* ha sido desarrollada la pantalla, pero no muestra las estadísticas.

En la tabla se aprecia que se han cumplido la mayoría de los requisitos.

Por otro lado, los objetivos profesionales se han cumplido por completo.

En conclusión, podría decirse que los resultados satisfacen los objetivos.

9.2 MEJORAS A FUTURO

Sobre las características desarrolladas, hay varias funcionalidades que podrían añadirse para mejorar la experiencia de usuario sin necesidad de invertir mucho tiempo en modificar la plataforma:

- Cambiar la vista de categorías por una con todas las lecciones y poder aplicar filtros: por categoría, por dificultad, mostrar sólo favoritas, por popularidad (cantidad de comentarios), etc.
- Android proporciona una API llamada MediaSession que se puede integrar con ExoPlayer. Esta permite controlar eventos de reproducción como Play, Pausa, Pista Anterior/Siguiente, o Reiniciar. Estos eventos se podrían controlar desde Android Wear o Google Assistant (voz) entre otras. Sería un complemento interesante para las sesiones de práctica, dado que el usuario no tendría que apartar las manos de la guitarra.
- La sección de estadísticas quedó parcialmente implementada, pero para completarla, bastaría con añadir variables en el modelo de *Usuario* que se almacena en la base de datos y actualizarlas cada vez que se terminase de reproducir una lección.

La sección de práctica quedó sin implementar debido a las dificultades que presentó en la etapa de diseño: no había librerías pensadas para ello y la creación de un algoritmo de detección de acordes no es trivial (hay Trabajos de Fin de Grado [41] sobre ello). Sin embargo, el 8 de mayo lanzaron en fase beta una característica nueva para Firebase: ML Kit.

ML Kit prorciona utilidades para Machine Learning. Por defecto, incorpora funciones de detección de texto, rostros, paisajes y etiquetado de imágenes, pero también permite incluir modelos de TensorFlow Lite desarrollados en Python. Bohumír Zámečník tiene un proyecto publicado en GitHub bajo la licencia MIT basado en la detección de acordes con TensorFlow [42].

Otras mejoras que podrían conseguirse implementando funciones de Firebase son:

- Utilizar Firebase Predictions para personalizar de forma dinámica el contenido mostrado a diferentes grupos de usuario, filtrando por tipo de interés. Así, en la ordenación de la lista de lecciones, tendrían prioridad las que tienen más probabilidad de gustar más al usuario.
- Aproximadamente en la mitad de la fase de desarrollo, se publicó Cloud Firestore, otra implementación de base de datos más escalable que Realtime Database. Sería interesante migrar la plataforma para aprovechar las ventajas que proporciona.

10. BIBLIOGRAFÍA

[1] «Creative Commons,» [En línea]. Disponible en: https://creativecommons.org/licenses/.

[2] «Apache License,» [En línea]. Disponible en: https://www.apache.org/licenses/LICENSE-2.0.

[3] «Reglamento (UE) 2016/679 del Parlamento Europeo y del Consejo,» [En línea]. Disponible en: https://eur-lex.europa.eu/legal-content/ES/TXT/?uri=CELEX%3A32016R0679.

[4] «Kotlin,» [En línea]. Disponible en: https://kotlinlang.org.

[5] «Ionic,» [En línea]. Disponible en: https://ionicframework.com/.

[6] Facebook, «React Native,» [En línea]. Disponible en: https://facebook.github.io/react-native/.

[7] «IDC,» [En línea]. Disponible en: https://www.idc.com/promo/smartphone-market-share/os.

[8] «JetBrains,» [En línea]. Disponible en: https://www.jetbrains.com/.

[9] Android Developers, «Activity,» [En línea]. Disponible en: https://developer.android.com/reference/android/app/Activity.

[10] Google, «Fragmentos,» [En línea]. Disponible en: https://developer.android.com/guide/components/fragments.

[11] Android Developers, «Intents y filtros de intents,» [En línea]. Disponible en: https://developer.android.com/guide/components/intents-filters.

[12] Google, «Firebase,» [En línea]. Disponible en: https://firebase.google.com/.

[13] «Geared,» [En línea]. Disponible en: http://www.geared.rocks.

[14] «Solo Learn,» [En línea]. Disponible en: https://www.sololearn.com/.

[15] Google, «Material Design,» [En línea]. Disponible en: https://material.io/design/.

[16] Ubisoft, «Rocksmith,» [En línea]. Disponible en: https://rocksmith.ubisoft.com/rocksmith/en-us/home/.

[17] Google, «FirebaseUI-Android,» [En línea]. Disponible en: https://github.com/firebase/FirebaseUI-Android.

[18] F. Devos, «Single Activity Architecture,» [En línea]. Disponible en: https://github.com/wealthfront/magellan/wiki/Single-Activity-Architecture.

[19] «Buttons: floating action button,» [En línea]. Disponible en: https://material.io/design/components/buttons-floating-action-button.html.

[20] «Bottom Sheets,» [En línea]. Disponible en: https://material.io/develop/android/components/bottom-sheet-behavior/.

[21] «ISO-8601,» [En línea]. Disponible en: https://www.iso.org/iso-8601-date-and-time-format.html.

[22] «Threeten,» [En línea]. Disponible en: http://www.threeten.org/threetenbp/.

[23] Google, «ExoPlayer,» [En línea]. Disponible en: https://google.github.io/ExoPlayer/.

[24] Google, «Firebase Console,» [En línea]. Disponible en: https://console.firebase.google.com/.

[25] «Gource,» [En línea]. Disponible en: http://gource.io/.

[26] Songfacts, «Hey Joe,» [En línea]. Disponible en: http://www.songfacts.com/detail.php?id=2241.

[27] R. C. Martin, Clean architecture: a craftsman's guide to software structure and design, Prentice Hall Press, 2017.

[28] «Bok,» [En línea]. Disponible en: http://www.bok.net/dash/.

[29] «Peach,» [En línea]. Disponible en: https://peach.blender.org/.

[30] A. Leiva, «Devexperto,» [En línea]. Disponible en: https://devexperto.com/principio-de-inversion-de-dependencias/.

[31] Google, «Dagger,» [En línea]. Disponible en: https://github.com/google/dagger.

[32] Android Developers, «RecyclerView,» [En línea]. Disponible en: https://developer.android.com/reference/android/support/v7/widget/RecyclerView.

[33] Android Developers, «Ktx,» [En línea]. Disponible en: https://developer.android.com/kotlin/ktx.

[34] «Anko,» [En línea]. Disponible en: https://github.com/Kotlin/anko.

[35] «Glide,» [En línea]. Disponible en: https://github.com/bumptech/glide.

[36] «Mockito,» [En línea]. Disponible en: http://site.mockito.org/.

[37] S. Wambler, «Agile Data,» [En línea]. Disponible en: http://www.agiledata.org/essays/tdd.html.

[38] Android Developers, «Local Unit Tests,» [En línea]. Disponible en: https://developer.android.com/training/testing/unit-testing/local-unit-tests.

[39] Android Developers, «Supporting Devices,» [En línea]. Disponible en: https://developer.android.com/training/basics/supporting-devices/languages.

[40] Android Developers, «Espresso,» [En línea]. Disponible en: https://developer.android.com/training/testing/espresso.

[41] J. M. Macías, «DISEÑO E IMPLEMENTACIÓN DE UN DETECTOR AUTOMÁTICO DE ACORDES,» 2012. [En línea]. Disponible en: https://orff.uc3m.es/bitstream/handle/10016/15588/PFC%20Julio%20Martin.pdf.

[42] B. Zámečník, «Chord Recognition,» [En línea]. Disponible en: https://github.com/bzamecnik/ml/tree/master/chord-recognition.

Extended Abstract

1. Introduction, motivation, and objectives

Guitar learning methods share that all of them deflect some facets in the learning process because either they are too rigid, or too ambiguous.

People that travel regularly make great use of smartphones as they provide accessible ways of learning or consuming multimedia content.

Using a smartphone to learn how to play an instrument is more convenient than a computer, as the latter not always feature a microphone.

The purpose of this project is to create a platform for music learning with a scalable architecture.

Kotlin is the new official language for Android application development.

Firebase is a platform owned by Google that offers cloud services to simplify development time.

ExoPlayer is an open source media player developed by Google that supports more functionalities than the default Android player. One of those functionalities is the reproduction of adaptive streamings, that detect the available bandwidth and chose different bitrates to optimize buffering time and data usage.

The **motivation** behind this project is combining my passion for music and programming using knowledge acquired during the degree and my professional career.

The **technical objective** is for the application to meet a mature state enough to be a proof of concept.

On the other hand, **professional objectives** are:

- Being able to combine knowledge acquired during the degree and software development.
- Dominating ExoPlayer and DASH encoding.
- Achieving an advanced level with Kotlin language.
- Learning a backend technology.
- Applying a clean architecture so the application can escalate and be easy to test.

Talking about the **regulatory framework,** there are three points that must be taken into acount:

- *Software licenses*: If the application uses resources of third-party libraries, it's creators retain certain rights over them. The application must reference all of them and quote its license.

- *Permissions:* In case the application needs access to device information or component such as the phone state or the camera, it has to prompt the user for them. Since Android 6.0, the user can opt-in only for the permissions he wants.

- *The General Data Protection Regulation*: An application that uses user data, must consider the Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). This also applies if the application does not directly use it, but third-party software used does.

2. State of the art

Android: Android is an operating system developed by Google. Developing native Android apps requires the Android Software Development Kit (SDK) and the Android Studio development environment.

There are two characteristics that make Android a great OS for developers:

First, it's easy for distributing applications via the app store (Play Store). Second, it has approximately 85% of the market share in smartphones.

Although Android P or 9.0 is the last version available for developers, more than 90% of the current devices feature Android 6.0 or lower versions.

Alternatives: Geared is a web page that allows registered users to watch lessons organized by technique and difficulty. Solo learn is a web platform with an Android application used to learn programming languages. Rocksmith is a video game with a robust search recognition algorithm.

Application	**Strenghts**	**Weaknesses**
Geared	• Good distribution of information • Plenty of content	• It doesn't have a community • Not compatible with mobile devices
SoloLearn	• Intuitive learning methodology • Big user community	• It's not a music learning application

	• Available for mobile devices	
Rocksmith 2014	• Chord detection algorithm • Practice oriented	• Only available for computer and consoles • No user community

Technology:

- *Kotlin*: programming language developed by JetBrains. It runs in the Java Virtual Machine so it can be used everywhere Java is. It allows for a more expressive coding, and it's interoperable with Java code. As a primary advantage, it's null-safe, which is the greatest source of errors in an Android application.

- *Firebase*: it is a platform owned by Google that provides web services for mobile and web development. On this project, the following features were used:

- Firebase authentication: backend services to authenticate users using different methods such as facebook, Twitter, Google or email address.
- Firebase Realtime Database: non-relational database using a JSON-like format.
- Storage: store objects like text, images, and video.
- Crashlytics: crash reports in real time to allow tracking stability issues.

3. Design and requirements

Requirements can be split in two kinds, functional and technical.

Functional specifications of the application are:

- Lessons section ordered by category and difficulty.
- Each lesson must have a text-based introduction and an explicative video.
- User management and authentication to store user progress.
- Users must have the possibility to give feedback on each lecture.
- Practice section using the smartphone's microphone and a chord detection algorithm.
- User profile section to display personal statistics such as completed lessons for each category and progress in different techniques.

Technical requirements are:

- Make use of clean architecture principles.
- Use an architectural design pattern (such as Model View Presenter).
- Use an adaptive streaming technology to optimize bandwidth usage during video reproduction. For this, Dynamic Adaptive Streaming over HTTP (DASH) was used.

- User Interface must follow the rules defined by Material Design.
- Implement tests that assert the right behaviour of the code in the long term.

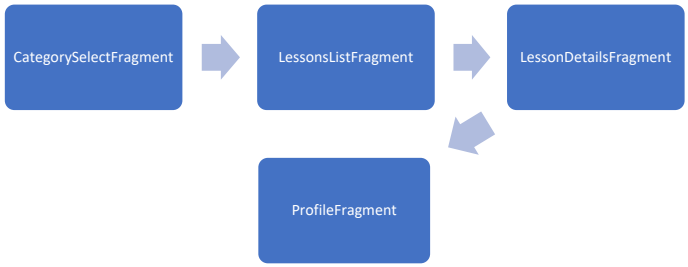
4. Application development

Application development is divided in three chapters: Screens, Backend, and Architecture.

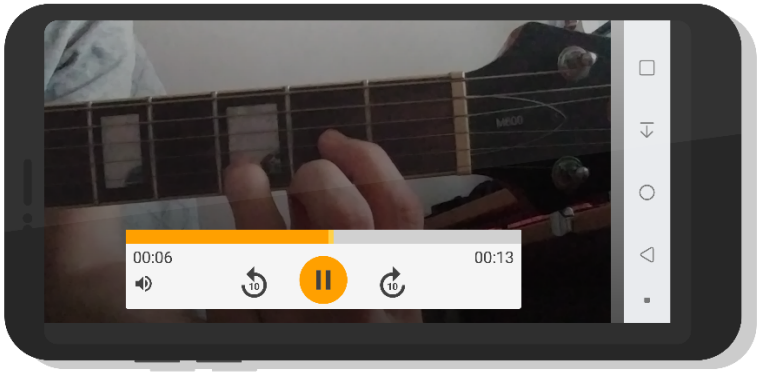
Screens: the application splits into three main screens.

- *AuthUI*: for authentication, it provides register and login interfaces via email or Google. As it's a Firebase library, it's completely integrated with the application and there is no need for further development Authentication wise.

- *ApplicationActivity*: It starts with a view of the different categories, clicking one will navigate to a list with the lessons of that category. The next screen is a detailed view of the lesson with a comments section to post and display user feedback. Finally, clicking the lesson's Play button will open the player.



- *PlayerActivity*: it's a fullscreen screen showing a Player and customized controls. It has the possibility to play DASH content. The customized controls allow the user to pause and resume, mute, seek and jump ten seconds forward or backwards.



Backend: Different modules of Firebase have been used to achieve the specified requirements. User credentials in Authentication. Realtime Database to store contents. Google Drive for images. FirebaseStorage for videos.

Architecture covers the lower layers of the project. Model View Presenter was used as an architectural pattern.

5. Testing

For the application, two kinds of test were developed: instrumentation tests, and local or unit tests.

Instrumentation tests are those that run on an emulator or a physical device and make use of the Android framework.

Unit tests run on the local machine, in this case, the Java Virtual Machine, to minimize execution time. These tests are used to verify the logic of specific code that is not related to the Android framework or which dependencies can be filled out with mocks.

6. Project management

Budget: A PC and a laptop were used in the development for a total of 1392€. Three Android devices were used to deploy the application, for a total of 644€.

29.000€/year salary was set for a Junior developer with 2 years of experience in the field and 35€/h for the tutor of the project, María Celeste Campo Vázquez. In total, the development consumed 540h from me and 20h from the tutor, which makes a total of 8260€.

Therefore, the total cost of this project is 10296€.

Monetization: To monetize the project, a free version with optional downloadable paid content was the best option. This monetization model offers several advantages:

- It allows more users to acquire paid content as it would be cheaper than a complete paid app.
- As the content in the application comes from the backend, there is no need for the user to update the client application to include new content.
- It provides higher long-term revenues thanks to users that pay for content on a regular basis.

7. Conclusions

The following chart checks if the requirements set in Design and Requirements were met:

Specification	**Implemented**
Lessons section ordered by category and difficulty	Yes
Lectures contain theoretical explanation and video	Yes
Create media player	Yes
Users can comment in lessons	Yes
Practice section with chord detection	No
Sección de perfil con estadísticas personales	Partially*
User management and authentication to store progress	Yes
Clean architecture principles	Yes

An adaptive streaming technology was used	Yes
Follows Material Design guidelines	Yes
Tests that verify the code	Yes

Overall, the results were successful.

Future improvements:

Some functionalities could be added with not much effort:

- Display all lessons in one list ordered them by relevance to the user or popularity (number of comments, for example).
- Use MediaSession API to handle events like Play and Pause so the player can be used with the voice.
- Statistics section wasn't implemented but only some variables in the User model would be needed for it.

Practice section was not implemented due to its technical difficulties at the moment. However, Firebase recently released the beta version of ML Kit.

ML Kit provides utilities for Machine Learning, and it can include TensorFlow models. For example, Bohumír Zámečník created a Python program for chord detection based on TensorFlow published under the MIT license.

Other improvements that could be achieved using Firebase services are:

- Use Firebase Predictions to personalize dynamically content offered to different groups of users, filtering by the type of interest.
- A new database alternative was launched lately, Cloud Firestore, which is more scalable than Realtime Database.