**Índice de contenidos**

[**Objetivos** 4](#_Toc138841414)

[**Estado del arte** 5](#_Toc138841415)

[Android Studio 5](#_Toc138841416)

[Kotlin 6](#_Toc138841417)

[Jetpack Compose 10](#_Toc138841418)

[Firebase 19](#_Toc138841419)

[Firebase Authentication 20](#_Toc138841420)

[Firebase Cloud Firestore 21](#_Toc138841421)

[Firebase Cloud Storage 22](#_Toc138841422)

[**Aplicación** 23](#_Toc138841423)

[Interfaz de usuario 23](#_Toc138841424)

[Objetos 24](#_Toc138841425)

[Alimento 24](#_Toc138841426)

[Ejercicio 25](#_Toc138841427)

[Rutina 26](#_Toc138841428)

[Pantallas 29](#_Toc138841429)

[Login 29](#_Toc138841430)

[Registro 32](#_Toc138841431)

[Principal 35](#_Toc138841432)

[Rutinas 37](#_Toc138841433)

[verRutina 41](#_Toc138841434)

[añadirEjercicios 43](#_Toc138841435)

[Ejercicios 43](#_Toc138841436)

[verEjercicio 43](#_Toc138841437)

[Nutricion 43](#_Toc138841438)

[Alimento 43](#_Toc138841439)

[Calendario 43](#_Toc138841440)

[añadirRutinaCalendario. 43](#_Toc138841441)

[Otros detalles 43](#_Toc138841442)

[Tutorial de la aplicación 44](#_Toc138841443)

[**Conclusiones y mejoras** 45](#_Toc138841444)

[**Referencias** 46](#_Toc138841445)

[Ilustración 1. Android Studio 6](#_Toc138182754)

[Ilustración 2. Logotipo de Kotlin 7](#_Toc138182755)

[Ilustración 3. Logotipo de Firebase 8](#_Toc138182756)

[Ilustración 4. Pantalla de Login 12](#_Toc138182757)

[Ilustración 5. Pantalla de Registro 14](#_Toc138182758)

[Ilustración 6. Pantalla Principal 16](#_Toc138182759)

[Ilustración 7. Pantalla de Rutinas 18](#_Toc138182760)

# **Objetivos**

El objetivo principal de este Trabajo de Fin de Grado (TFG) es desarrollar una aplicación móvil de gimnasio utilizando Jetpack Compose y Kotlin, con el propósito de ofrecer a los usuarios una experiencia intuitiva y eficiente para llevar un seguimiento de sus entrenamientos y mejorar su rendimiento físico.

Para lograr este objetivo, se plantean los siguientes subobjetivos:

* Investigar y analizar a fondo las tecnologías Jetpack Compose y Kotlin, comprendiendo sus características, ventajas y su integración en el ecosistema de desarrollo de Android.
* Diseñar una interfaz de usuario atractiva y fácil de usar para la aplicación, centrándose en aspectos clave como la usabilidad, la estética y la funcionalidad.
* Utilizar Jetpack Compose como herramienta principal para crear interfaces de usuario dinámicas y declarativas, aprovechando sus capacidades para componer vistas de manera eficiente.
* Implementar las funcionalidades necesarias para el seguimiento de entrenamientos, como la programación de rutinas, el registro de progresos y el acceso a bibliotecas de ejercicios.

Al cumplir estos objetivos, se espera demostrar la viabilidad y las ventajas del uso de Jetpack Compose y Kotlin en el desarrollo de aplicaciones de gimnasio para dispositivos móviles. La aplicación resultante brindará a los usuarios una herramienta efectiva para mejorar su experiencia en el gimnasio, llevar un seguimiento de sus entrenamientos y alcanzar sus objetivos de fitness.

# **Estado del arte**

## Android Studio

Android Studio es el entorno de desarrollo oficial para Android, basado en IntelliJ IDEA, lo que significa que cuenta con un poderoso editor de código y herramientas para desarrolladores. Además, ofrece funciones adicionales para la compilación de aplicaciones Android, tales como:

* Sistema de compilación basado en Gradle, lo que permite personalizar y configurar el proceso de compilación, así como crear múltiples APK con diferentes funciones usando el mismo proyecto.
* Un emulador rápido.
* Un entorno de desarrollo integrado para todos los desarrollos Android.
* Integración de plantillas de código.
* Instant Run para aplicar cambios rápidamente mientras se ejecuta la aplicación sin la necesidad de generar un nuevo APK.
* Herramientas Lint para detectar problemas en cuanto a rendimiento, usabilidad, compatibilidad de versiones, etc.
* Herramientas de depuración que incluyen monitores de rendimiento y seguimiento de asignaciones de memoria.
* Compatibilidad con C++.



Ilustración 1. Logotipo de Android Studio

## Kotlin

Kotlin es un lenguaje de programación utilizado sobre todo para el desarrollo de aplicaciones Android que corre sobre la Máquina Virtual Java (JVM).

Su desarrollo primario es de un equipo de programadores de JetBrains, una compañía conocida por crear IntelliJ IDEA y Android Studio la cual tiene base en San Petersburgo, Rusia (el nombre proviene de la Isla de Kotlin, cerca de San Petersburgo). En mayo de 2017, Google anuncia Kotlin como lenguaje oficial para el desarrollo de Android.

El líder de desarrollo Andrey Breslav enuncio que Kotlin está diseñado para ser un lenguaje de programación orientado a objetos de calidad industrial, y para ser un lenguaje mejor que Java, pero que todavía debe ser plenamente interoperable con código Java, permitiendo a las compañías hacer una migración gradual de Java a Kotlin, y de estar forma se consiga un trasvase de conocimiento efectivo y regulado, sin ello repercutir en el código de los programadores

Como Kotlin y Android Studio fueron desarrollados por JetBrains, este lenguaje se encuentra perfectamente integrado en Android Studio, lo que pone a disposición de los desarrolladores las siguientes ventajas:

* **Compatibilidad**: Compatible con Java 6, lo que permite la compatibilidad con dispositivos antiguos.
* **Interoperabilidad**: Al ser interoperable con Java podemos seguir haciendo uso de las librearías de Java.
* **Performance**: Una aplicación desarrollada en Kotlin es igual de eficiente que una desarrollada en Java.
* **Curva** **de** **aprendizaje**: Es sencillo para un desarrollador Java empezar a desarrollar en Kotlin.
* **Seguro** **contra** **nulos**: Kotlin requiere que se verifique el valor no null de las variables en tiempo de compilación.
* **Programación funcional**: Kotlin incluye características de la programación funcional que se pueden combinar con la programación orientada a objetos.
* **Limpieza de código**: El código escrito en Kotlin es más legible y presenta características que permiten reducir la cantidad de código necesario.

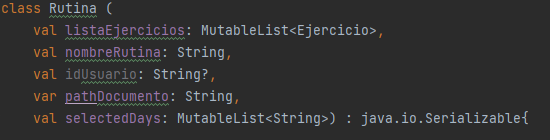


Ilustración 2. Logotipo de Kotlin

Las principales ventajas de kotlin sobre Java son las siguientes:

Seguridad ante nulos

Kotlin permite diferenciar entre aquellas variables que pueden tener valor nulo (nullable) y aquellas que no (por defecto) mediante el operador seguro **?**  tal como se puede ver en el siguiente ejemplo:



Si la variable nombreRutina fuera null, habría un error de compilación, el cual no se produce en caso de que la variable idUsuario fuera null, ya que al tener el operador **?**, Kotlin permite que esa variable sea nula.

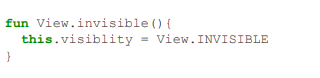
Para acceder a estas variables nullables, siempre será necesario añadir el operador seguro **?** con el fin de asegurar que esa llamada solo se produzca en caso de que no tenga valor nulo:



Funciones de extensión

Kotlin permite las funciones y variables de alto nivel, permitiendo así que sean accesibles desde cualquier parte de la aplicación sin necesidad de crear nuevas estancias o variables estáticas. Esto permite a los desarrolladores la posibilidad de crear las funciones de extensión, unas funciones que permiten ampliar la funcionalidad de las clases sin tocar su código.

Un ejemplo sencillo podría ser crear la función *invisible()* en una vista:



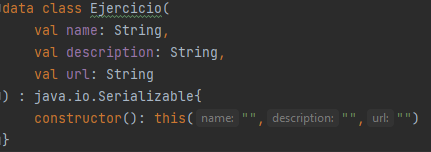
De esta forma, se puede añadir funcionalidad a cualquier clase y utilizar la funcionalidad añadida como si fueran propias de la clase.

Clases de datos

Kotlin permite crear clases denominadas data class para aquellas clases cuya única funcionalidad es almacenar datos.

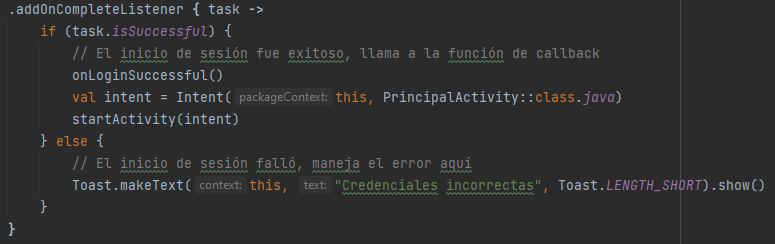
De esta forma, el compilador genera automáticamente las siguientes funciones:

* Equals() / hasCode().
* toString().
* Funciones componentN() correspondientes a las N propiedades del constructor y en el orden que se han declarado.
* Función copy() que permite copiar un objeto cambiando alguna de sus propiedades.



Funciones lambda

Estas funciones son una característica de Kotlin que permite tratas a las funciones como objetos de primera clase, es decir, se pueden almacenar en variables, pasar como argumentos y retornar como resultado en otras funciones.



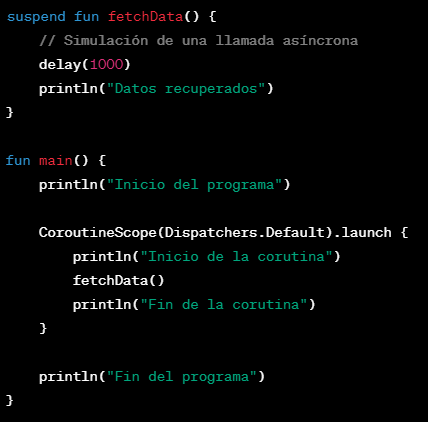
Las funciones lambda se expresan utilizando **{}** y pueden tener parámetros y una expresión de retorno opcional al final.

En este ejemplo, el método **addOnCompleteListener** recibe como parámetro una función lambda que toma un parámetro **task**, el cual representa la tarea de iniciar sesión y según el resultado de la tarea, se ejecutará una parte de código u otra.

Co-rutinas

Esta característica permite ejecutar de manera asíncrona código que se encuentra escrito secuencialmente. Esto se consigue suspendiendo la ejecución del hilo hasta que el resultado esté disponible.

Una de las formas mas comunes de ejecutar las co-rutinas es mediante el uso de la función launch, al cual se le pasa una función lambda con el bloque de código que se ejecutará de forma asíncrona.



En este ejemplo, aunque pareciera que el resultado de ejecución fuera en el orden “Inicio programa” – “Inicio corutina” – “Fin corutina” – “Fin programa”, como el hilo de ejecución del launch se detiene durante un segundo, la frase “Fin programa” se ejecuta antes de que acabe el launch, por lo cual el resultado es el siguiente:

Inicio del programa

Fin del programa

Inicio de la corutina

Fin de la corutina

Inicio de la corutina y fin de la corutina se mantienen en orden, pero detrás de Fin del programa, ya que ambas se muestran simultáneamente una vez ha terminado el launch.

## Jetpack Compose

Jetpack Compose es un moderno kit de herramientas declarativas de IU para Android que simplifica y acelera el desarrollo de interfaces de usuario nativas en Android. Fue desarrollado por Google como parte del conjunto de bibliotecas y herramientas de Jetpack. A continuación, se mencionan algunas de las principales ventajas y características de Jetpack Compose:

Programación declarativa

La programación declarativa es un paradigma de programación en el que se describe el qué se desea lograr, en lugar de especificar cómo se debe hacer y que en Jetpack Compose se aplica a la construcción de interfaces de usuario (UI) en Android.

En lugar de utilizar el enfoque tradicional de construcción de interfaces de usuario basado en vistas, en el que se manipulan y actualizan directamente los elementos de la interfaz, Jetpack Compose adopta un enfoque declarativo. En este enfoque, defines la estructura y el estado de la interfaz de usuario en un conjunto de funciones llamadas componibles. Estas funciones toman datos como entrada y emiten elementos de la interfaz de usuario como resultado.

Menos código

Jetpack Compose permite escribir menos código en comparación con el sistema de vistas tradicional de Android. Con Compose, se puede lograr más con menos código, lo que facilita el desarrollo, la revisión y el mantenimiento del código. Además, el código se escribe únicamente en Kotlin, eliminando la necesidad de alternar entre Kotlin y XML.

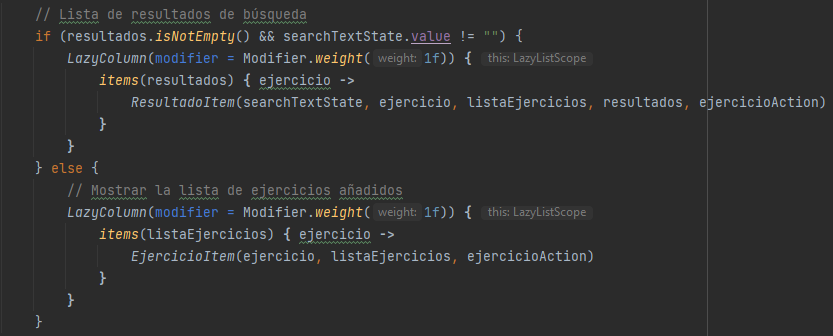
Funciones de extensión

Al igual que Kotlin, Jetpack Compose admite funciones de extensión. Esto permite extender la funcionalidad de las clases existentes sin modificar su código original. Las funciones de extensión en Jetpack Compose pueden ser utilizadas para agregar comportamiento o personalizar componentes de IU sin la necesidad de crear nuevas clases o utilizar variables estáticas.

Composición de componentes

Jetpack Compose facilita la creación de interfaces de usuario mediante la composición de componentes reutilizables como bloques de construcción. Los componentes son funciones que definen como se visualiza y se comporta un elemento de la interfaz de usuario y se pueden combinar y anidar componentes más grandes con componentes más pequeños para construir interfaces de usuario complejas de manera más sencilla y legible.

Esta composición se realiza de manera declarativa, declarando como se debe comportar la interfaz de usuario en diferentes estados o situaciones



Compatibilidad con Kotlin

Jetpack Compose está totalmente integrado con Kotlin y hace uso de características avanzadas del lenguaje, como las funciones lambda y las corutinas. Esto permite aprovechar el poder de Kotlin y su interoperabilidad con el código Java existente en el ecosistema de Android.

Actualizaciones eficientes

Jetpack Compose utiliza un mecanismo de recomposición eficiente para actualizar solo las partes de la interfaz de usuario que han cambiado. En lugar de actualizar manualmente una jerarquía de vistas completa, Compose identifica los cambios y realiza actualizaciones incrementales, lo que mejora el rendimiento y reduce el consumo de recursos.

Componentes

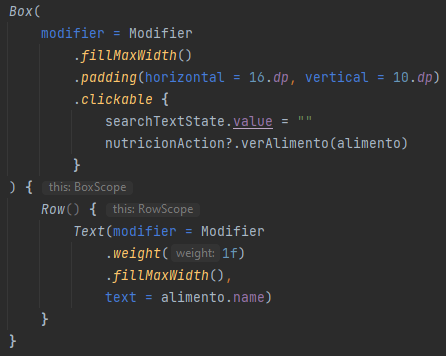
Los principales componentes de Jetpack Compose que se han utilizado en este proyecto son:

* Scaffold: Se trata de un componible que proporciona una estructura de diseño básica siguiendo los principios de Material Design y se utiliza como base para construir la interfaz de usuario. Ofrece las siguientes características:  
  + topBar: Permite agregar una barra superior a la pantalla.
  + floatingActionButton: Permite agregar un botón de acción flotante a la pantalla.
  + drawerContent: Permite agregar un menú deslizante a la pantalla.
  + Content: Es el contenido principal de la pantalla.
  + bottomBar: Permite agregar una barra inferior de navegación a la pantalla.

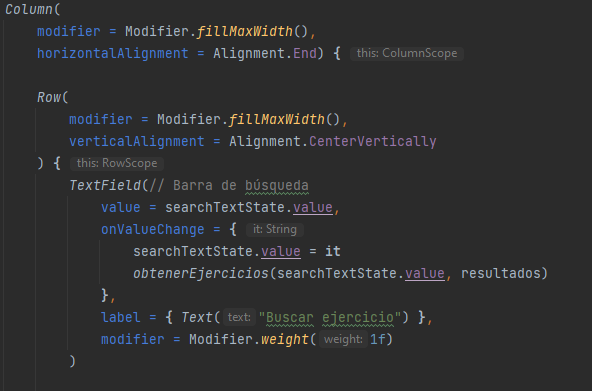
Además, se le puede pasar el contenido de la pantalla como una lambda tal que: Scaffold(){ “Contenido” }



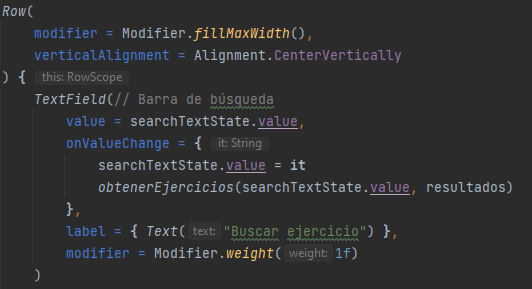
* Box: Se trata de un componible que se utiliza para crear un contenedor rectangular flexible que puede contener otros componentes y cuenta con los siguientes parámetros:
  + Modifier: Permite aplicar modificadores de diseño y controlar aspectos como tamaño, posición, espaciado, etc. Una función importante del modifier puede ser Modifier.clickable, ya que permite definir una acción que se realizará cuando el usuario clique en el box.
  + contentAlignment: Define la alineación del contenido dentro del Box.
  + Content: Especifica el contenido del Box mediante una lambda. Se puede utilizar tal que content = { “Contenido” } o Box(){ “Contenido” }, al igual que el Scaffold.
  + backgroundColor: Define el color del fondo del Box.
  + Border: Permite agregar bordes al Box.



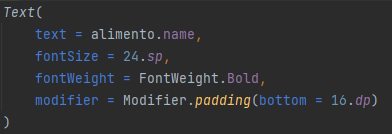
* Column: Se trata de un componible que se utiliza para crear un contenedor vertical que puede contener otros componentes y cuenta con los siguientes parámetros:
  + Modifier: Permite aplicar modificadores al Column para cambiar su apariencia o comportamiento.
  + verticalArrangement: Define como se alinearán verticalmente los elementos secundarios dentro de la columna.
  + horizontalAlignment: Define como se alinearán horizontalmente los elementos secundarios dentro de la columna.
  + Content: Especifica el contenido del Column mediante una lambda. Se puede utilizar tal que content = { “Contenido” } o Column(){ “Contenido” }, al igual que el Scaffold.



* Row: Se trata de un componible que se utiliza para crear un contenedor horizontal que puede contener otros componentes y cuenta con los siguientes parámetros:
  + Modifier: Permite aplicar modificadores al Row para cambiar su apariencia o comportamiento.
  + verticalArrangement: Define como se alinearán verticalmente los elementos secundarios dentro de la fila.
  + horizontalAlignment: Define como se alinearán horizontalmente los elementos secundarios dentro de la fila.
  + Content: Especifica el contenido del Row mediante una lambda. Se puede utilizar tal que content = { “Contenido” } o Row(){ “Contenido” }, al igual que el Scaffold.



* Text: Se trata de un componible que se utiliza para mostrar texto en la interfaz de usuario y cuenta con los siguientes parámetros:
  + Text: Permite definir el texto que se mostrará en el componente.
  + Modifier: Permite aplicar modificaciones al Text para cambiar su apariencia o comportamiento.
  + Color: Define el color del texto.
  + fontSize: Establece el tamaño del texto.
  + fontStyle: Permite establecer el estilo de fuente del texto como negrita o cursiva.
  + textAlign: Define la alineación horizontal del texto dentro del componente.
  + maxLines: Especifica el número máximo de líneas que se mostrarán para el texto.

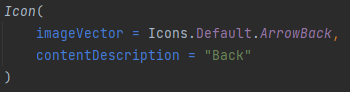


* TextField: Se trata de un componible que permite capturar la entrada de texto del usuario y cuenta con los siguientes parámetros:
  + Value: Representa el valor actual del Texto. Puede ser una variable mutable (**MutableState**) o una variable que se actualice mediante la llamada a la función **onValueChange**.
  + onValueChange: Es una función lambda que se llama siempre que el usuario actualice el texto del componente.
  + Modifier: Permite aplicar modificaciones al TextField para cambiar su apariencia o comportamiento.
  + Label: Especifica la etiqueta que se muestra en la parte superior del componente.
  + Placeholder: Define un texto que se muestra únicamente cuando no hay ningún valor ingresado en el componente.
  + keyboardActions: Permite especificar las acciones que se deben realizar cuando el usuario interactúa con el teclado.
  + singleLine: Se trata de un parámetro booleano que indica si el componente debe aceptar solo una línea de texto (**true**) o varias (**false**).
  + Enabled: Se trata de un parámetro booleano que indica si el componente está habilitado y el usuario puede interactuar o editar el texto (**true**) o no (**false**).

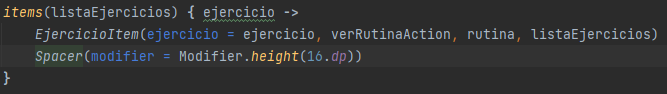
Similar al componente TextField podemos encontrar OutlinedTextField, cuya principal diferencia es que OutlinedTextField cuenta con un cuadro delimitador visible alrededor del campo de entrada. Además, cuenta con el parámetro **visualTransformation**, el cual permite aplicar transformaciones al texto, como por ejemplo ocultar las contraseñas con asteriscos.



* Icon: Se trata de un componible que se utiliza para mostrar un icono en la interfaz y cuenta con los siguientes parámetros:
  + imageVector: Especifica el icono como un vector de imagen predefinido de Material Design si no usas tus propios recursos de icono.
  + Painter: Permite definir el icono como un “Painter”, es decir, una imagen dibujable. Se utiliza para especificar tus propios recursos de icono añadidos en la carpeta res.
  + contentDescription: Se trata de un texto que describe la función del icono.
  + Modifier: Se trata de un modificador que se utiliza para aplicar estilos, comportamientos o diseños personalizados al icono.
  + Tint: Permite especificar un color de tinte para el icono.
  + Size: Define el tamaño del icono. Puede especificarse como un valor constante predefinido (**IconSize.Small**) o un valor personalizado en pixeles.
  + contentScale: Especifica la escala del contenido del icono.



* Spacer: Se trata de un componible que se utiliza para agregar espacios vacíos o separaciones entre elementos. No tiene parámetros propios, pero se puede configurar mediante modificadores para ajustar el tamaño de espacio.



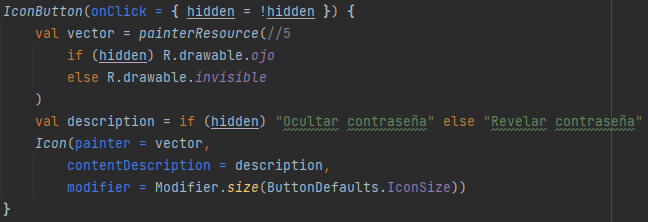
* LazyColumn: Se trata de un componible utilizado para crear una lista de elementos de manera eficiente que renderiza únicamente los elementos visibles en pantalla y los que están cerca de la ventana visible de tal forma que puedes arrastrar la pantalla para ver los siguientes elementos. LazyColumn cuenta con los siguientes parámetros:
  + Modifier: Un modificador que se utiliza para aplicar estilos, comportamientos o diseños personalizados al componente.
  + verticalArrangement: Especifica cómo se colocan verticalmente los elementos en el LazyColumn.
  + horizontalAlignment: Especifica cómo se alinean horizontalmente los elementos en el LazyColumn.
  + contentPadding: Especifica el espacio de relleno alrededor del contenido del LazyColumn.
  + items: Especifica la lista de elementos que se mostrarán en el LazyColumn. Puedes proporcionar una lista de datos y una función lambda que define cómo se debe representar cada elemento en la lista.
  + itemContent: Define el contenido visual y el comportamiento de cada elemento en la lista. Puede ser una lambda que toma el elemento de la lista y crea la interfaz de usuario correspondiente.
  + reverseLayout: Un valor booleano que indica si el orden de los elementos en el LazyColumn debe invertirse. Si se establece en true, el último elemento se colocará en la parte superior y viceversa.

También existe **LazyRow**, el cual es similar a LazyColumn pero creando una lista de elementos de forma horizontal, lo cual no ha sido necesario en este proyecto.



* Button: Se trata de un componible que se utiliza para crear un botón interactivo en la interfaz de usuario y cuenta con los siguientes parámetros:
  + onClick: Permite definir la acción a ejecutar una vez el botón es pulsado. Para ello, se puede proporcionar una función lambda.
  + Modifier: Un modificador que se utiliza para aplicar estilos, comportamientos o diseños personalizados al componente.
  + Enabled: Se trata de un valor booleano que indica si el botón está habilitado (true) o deshabilitado (false).
  + Colors: Define los colores del botón en diferentes estados (normal, presionado, desactivado…).
  + Shape: Permite definir la forma del botón.
  + contentPadding: Especifica el espacio de relleno alrededor del botón.
  + Content: Especifica el contenido del botón mediante una lambda. Se puede utilizar tal que content = { “Contenido” } o Button(){ “Contenido” }, al igual que el Scaffold.

Similar a Button podemos encontrar IconButton, el cual es un subtipo específico de Button que se utiliza para representar un botón cuyo único contenido es un Icon; o OutlinedButton, el cual se diferencia únicamente de button en que este último muestra un borde alrededor del contenido en lugar de tener un fondo sólido.



Respecto a los componentes, es importante especificar que los parámetros relativos a la posición como contentAlignment, verticalArrangement y horizontalAlignment resultarán en error si el componente se encuentra dentro de otro componente, de tal forma que solo será correcto utilizar estos parámetros en el primer componente anidado.

En resumen, Jetpack Compose es un kit de herramientas moderno para el desarrollo de interfaces de usuario en Android. Con su enfoque declarativo, capacidad de reutilización de componentes, soporte total para Kotlin y eficiencia en las actualizaciones, Jetpack Compose ofrece beneficios significativos para simplificar y mejorar el desarrollo de aplicaciones Android.

## Firebase

Firebase es una plataforma de desarrollo de aplicaciones móviles y web desarrollada por Google. Proporciona una amplia gama de servicios y herramientas que permiten a los desarrolladores crear aplicaciones de alta calidad de manera más rápida y eficiente. Firebase se basa en la infraestructura de Google Cloud Platform y ofrece una integración perfecta con otras tecnologías de Google.

Los principales componentes de Firebase incluyen:

* Firebase Authentication: Un servicio de autenticación que permite a los usuarios registrarse e iniciar sesión en aplicaciones utilizando diferentes métodos, como correo electrónico y contraseña, proveedores de identidad externos y autenticación basada en números de teléfono.
* Firebase Cloud Firestore: Una base de datos en tiempo real y orientada a documentos que permite almacenar y sincronizar datos de manera eficiente entre clientes y servidores. Firestore ofrece una estructura de datos flexible y escalable, y proporciona potentes consultas y capacidades de sincronización en tiempo real.
* Firebase Cloud Storage: Un servicio de almacenamiento en la nube que permite a los desarrolladores almacenar y servir archivos estáticos, como imágenes, videos y documentos. Firebase Storage ofrece una alta disponibilidad y escalabilidad, y facilita la carga y descarga de archivos desde aplicaciones.

Firebase se destaca por su facilidad de uso, escalabilidad, confiabilidad y flexibilidad. Permite a los desarrolladores centrarse en la creación de características y experiencias de usuario excepcionales, ya que se encarga de la infraestructura subyacente y de la administración de servidores. Con Firebase, los desarrolladores pueden acelerar el desarrollo de aplicaciones, reducir la complejidad y ofrecer aplicaciones de alta calidad en menos tiempo.

En este proyecto nos hemos centrado en utilizar Firebase Authentication para la autenticación de usuarios y tanto Firebase Cloud Firestore como Firebase Cloud Storage para el almacenamiento de los datos en tiempo real.

Texto, Logotipo

Descripción generada automáticamente

Ilustración 3. Logotipo de Firebase

### Firebase Authentication

Firebase Authentication es un servicio de autenticación en la nube proporcionado por Google. Permite a los desarrolladores agregar fácilmente funcionalidades de autenticación segura a sus aplicaciones web y móviles, sin tener que desarrollar y mantener un sistema de autenticación personalizado.

Firebase Authentication ofrece una variedad de métodos de autenticación para adaptarse a las necesidades de diferentes aplicaciones y usuarios. Entre los métodos de autenticación disponibles se incluyen:

* Autenticación por correo electrónico y contraseña: Permite a los usuarios registrarse y acceder a la aplicación utilizando una combinación de correo electrónico y contraseña. Firebase maneja el almacenamiento seguro de las contraseñas y la verificación de las credenciales.
* Autenticación con proveedores de identidad externos: Firebase Authentication admite la autenticación con proveedores populares como Google, Facebook, Twitter y GitHub. Esto permite a los usuarios iniciar sesión en la aplicación utilizando sus cuentas existentes en estos servicios.
* Autenticación mediante números de teléfono: Firebase Authentication proporciona la capacidad de autenticar a los usuarios mediante la verificación de números de teléfono. Esto es especialmente útil para aplicaciones que se centran en la autenticación basada en SMS.
* Autenticación con proveedores de identidad personalizados: Firebase Authentication permite a los desarrolladores integrar proveedores de identidad personalizados utilizando OpenID Connect o SAML para autenticar a los usuarios con sistemas externos.

Una vez que los usuarios se autentican en la aplicación, Firebase Authentication proporciona una capa de seguridad para proteger los recursos y las acciones en la aplicación. Los desarrolladores pueden utilizar reglas de seguridad personalizadas para definir quién tiene acceso a qué datos y acciones, y Firebase se encargará de hacer cumplir esas reglas.

En resumen, Firebase Authentication es un servicio de autenticación en la nube que simplifica la implementación de la autenticación segura en aplicaciones web y móviles. Ofrece varios métodos de autenticación, protección de recursos y acciones con reglas de seguridad personalizadas, y funcionalidades adicionales para administrar usuarios. Con Firebase Authentication, los desarrolladores pueden garantizar la autenticación segura de los usuarios y concentrarse en la construcción de una experiencia de usuario excelente.

En este proyecto se ha utilizado para autenticar y almacenar usuarios por correo electrónico y contraseña o a través de Google.

### Firebase Cloud Firestore

Firebase Cloud Firestore es un servicio de base de datos en la nube proporcionado por Google. Se basa en un modelo de datos flexible y escalable, diseñado para almacenar y sincronizar datos en tiempo real en aplicaciones web y móviles.

Firestore utiliza un enfoque de base de datos NoSQL, lo que significa que no sigue el modelo tradicional de bases de datos relacionales. En lugar de eso, Firestore utiliza un modelo de datos basado en documentos y colecciones. Un documento es una unidad básica de almacenamiento que contiene datos en formato JSON. Estos documentos se agrupan en colecciones, lo que permite organizar los datos de manera lógica y jerárquica.

La escalabilidad y el rendimiento son características fundamentales de Firestore. El servicio está diseñado para manejar grandes volúmenes de datos y cargas de trabajo intensivas. Puede distribuir automáticamente los datos en diferentes servidores y escalar horizontalmente según sea necesario para garantizar un rendimiento rápido y constante.

Una de las ventajas clave de Firestore es su capacidad de sincronización en tiempo real. Esto significa que los cambios realizados en un dispositivo se propagan automáticamente a otros dispositivos conectados a la misma base de datos en tiempo real. Esto es especialmente útil para aplicaciones colaborativas o en tiempo real, donde múltiples usuarios pueden ver y actualizar los datos simultáneamente.

Firestore también ofrece capacidades de consulta y filtrado de datos flexibles. Puedes realizar consultas basadas en campos específicos, ordenar y filtrar resultados, y combinar múltiples condiciones de consulta para obtener los datos que necesitas de manera eficiente.

Además, Firestore se integra perfectamente con otras herramientas y servicios de Firebase, lo que amplía aún más su funcionalidad. Puedes utilizar la autenticación de usuarios, el almacenamiento de archivos, las notificaciones push y el análisis de Firebase junto con Firestore para construir aplicaciones completas y escalables.

En resumen, Firebase Cloud Firestore es un servicio de base de datos en la nube que ofrece un modelo de datos flexible, escalabilidad, sincronización en tiempo real y capacidades de consulta avanzadas. Es una opción poderosa para almacenar y sincronizar datos en aplicaciones web y móviles, y se integra estrechamente con otras herramientas de Firebase para brindar una solución completa de desarrollo de aplicaciones.

En este proyecto se ha utilizado Firebase Cloud Firestore para almacenar los datos referentes a ejercicios y rutinas.

### Firebase Cloud Storage

Firebase Cloud Storage es un servicio de almacenamiento en la nube ofrecido por Firebase, la plataforma de desarrollo de aplicaciones móviles y web de Google. Proporciona a los desarrolladores una solución escalable y confiable para almacenar y servir archivos estáticos, como imágenes, videos, documentos y otros recursos multimedia, en la infraestructura de almacenamiento en la nube de Google.

Algunas características clave de Firebase Cloud Storage incluyen:

* Almacenamiento seguro y duradero: Firebase Cloud Storage utiliza la infraestructura de almacenamiento de Google Cloud Platform, que ofrece altos niveles de seguridad y durabilidad. Los archivos se almacenan en ubicaciones geográficas redundantes para garantizar la disponibilidad y la integridad de los datos.
* Fácil integración: Firebase Cloud Storage se integra perfectamente con otros servicios de Firebase, lo que permite a los desarrolladores utilizarlo en conjunto con otros componentes, como Firebase Authentication y Firebase Firestore. Esto facilita el acceso y la administración de los archivos almacenados según las necesidades de autenticación y permisos de la aplicación.
* Carga y descarga de archivos sencilla: Los desarrolladores pueden cargar archivos en Firebase Cloud Storage desde aplicaciones móviles y web utilizando una API sencilla y fácil de usar. También pueden descargar archivos de manera eficiente para mostrarlos en la interfaz de usuario de la aplicación.
* Almacenamiento y entrega de contenido estático: Firebase Cloud Storage permite a los desarrolladores almacenar y servir contenido estático, como imágenes o archivos HTML, directamente desde la nube. Esto ayuda a mejorar el rendimiento de la aplicación al proporcionar una entrega rápida y eficiente de los recursos estáticos a los usuarios.

En resumen, Firebase Cloud Storage es una solución completa de almacenamiento en la nube que ofrece a los desarrolladores una forma sencilla y segura de almacenar y servir archivos estáticos en sus aplicaciones. Con su integración con otros servicios de Firebase y su escalabilidad inherente, Firebase Cloud Storage proporciona una base sólida para la gestión eficiente de archivos y recursos multimedia en aplicaciones móviles y web.

En esta aplicación se ha utilizado para el almacenamiento de las imágenes relacionadas a los ejercicios.

# **Aplicación**

## Interfaz de usuario

La interfaz de usuario consta de 12 pantallas: **Login, Registro, Principal, Rutinas, verRutina, añadirEjercicios, Ejercicios, verEjercicio, Nutricion, Alimento, Calendario y añadirRutinaCalendario.**

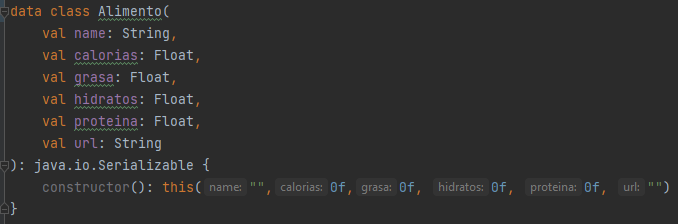
Con el fin de utilizar correctamente cada pantalla, también se han creado 3 clases de objetos denominadas: **Rutina, Alimento y Ejercicio.**

## Objetos

Con el fin de una mayor claridad a la hora de explicar las pantallas, se explicarán primero los objetos que se pueden encontrar en la aplicación:

### Alimento

La clase Alimento se ha creado principalmente con el objetivo de recuperar datos de la base de datos de Firebase relativa a los alimentos y su estructura es la siguiente:



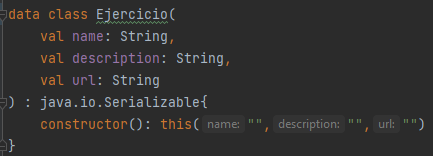
Esta clase contiene las propiedades necesarias para la visualización del Alimento en las diversas pantallas de la aplicación. Estas son:

* Name: Propiedad utilizada para representar el nombre del alimento.
* Calorías: Propiedad utilizada para representar el valor energético del alimento.
* Grasa: Propiedad utilizada para representar la cantidad de grasa que contiene el alimento (en gramos) por cada ración de 100 gramos.
* Hidratos: Propiedad utilizada para representar la cantidad de hidratos de carbono que contiene el alimento (en gramos) por cada ración de 100 gramos.
* Proteína: Propiedad utilizada para representar la cantidad de proteína que contiene el alimento (en gramos) por cada ración de 100 gramos.
* Url: Propiedad utilizada para almacenar la url de la imagen del alimento. Esta imagen se encuentra en Firebase Cloud Storage, desde donde se puede obtener un enlace. Una vez obtenido el enlace, se añade manualmente al alimento en la base de datos de Firebase Cloud Firestore.

Además, se utiliza la biblioteca Serializable para poder recuperar estos alimentos de la base de datos de Firebase Cloud Firestore y utilizarlos en la aplicación.

### Ejercicio

La clase Ejercicio se ha creado principalmente con el objetivo de recuperar datos de la base de datos de Firebase relativa a los ejercicios y su estructura es la siguiente:



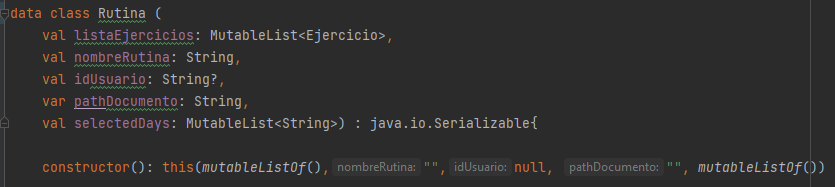
Esta clase contiene las propiedades necesarias para la visualización del Ejercicio en las diversas pantallas de la aplicación. Estas son:

* Name: Propiedad utilizada para representar el nombre del Ejercicio.
* Description: Propiedad utilizada para proporcionar una descripción sobre el ejercicio, como realizarlo y diversos consejos.
* Url: Propiedad utilizada para almacenar la url de la imagen del ejercicio. Esta imagen se encuentra en Firebase Cloud Storage, desde donde se puede obtener un enlace. Una vez obtenido el enlace, se añade manualmente al alimento en la base de datos de Firebase Cloud Firestore.

Además, se utiliza la biblioteca Serializable para poder recuperar estos ejercicios de la base de datos de Firebase Cloud Firestore y utilizarlos en la aplicación.

### Rutina

La clase Rutina se ha creado principalmente con el objetivo de interactuar con la base de datos de Firebase relativa a las Rutinas y su estructura es la siguiente:



Esta clase contiene las propiedades necesarias para la visualización de la Rutina en las diversas pantallas de la aplicación. Estas son:

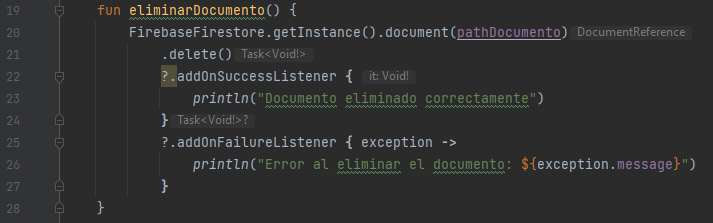
* ListaEjercicios: Propiedad utilizada para representar los ejercicios que conforman una rutina.
* NombreRutina: Propiedad utilizada para representar el nombre con el cual se visualizará la rutina.
* idUsuario: Propiedad utilizada para almacenar el id del usuario actualmente logeado.
* PathDocumento: Propiedad utilizada para almacenar la ruta de la Rutina en la base de datos de Firebase Cloud Firestore.
* SelectedDays: Propiedad utilizada para almacenar los días seleccionados a la hora de crear una Rutina.

Se utiliza la biblioteca Serializable para poder recuperar las rutinas desde la base de datos de Firebase Cloud Firestore y utilizarlas en la aplicación.

Además de las propiedades de la clase, la clase Rutina implementa también varias funciones para interactuar con la base de datos relativa a las Rutinas:

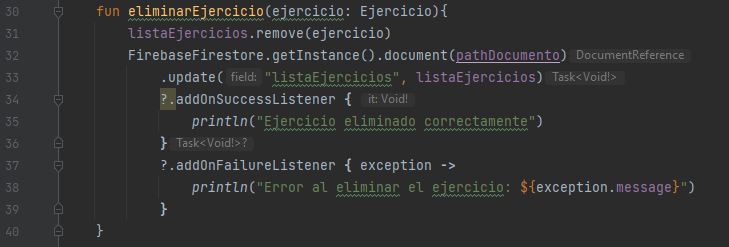
#### eliminarDocumento()

Esta función se encarga de eliminar una Rutina completa de la base de datos utilizando la API de FirebaseFirestore. Para ello, se utiliza el **pathDocumento** para navegar hasta el documento en cuestión y la función **.delete().**



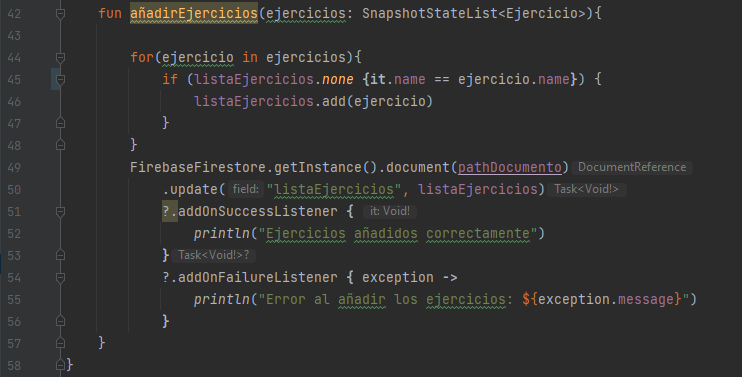
#### eliminarEjercicio()

Esta función se encarga de eliminar un ejercicio que se pasa por parámetro de **listaEjercicios**, la lista de ejercicios de la rutina. Para ello, se elimina el ejercicio de **listaEjercicios** de la aplicación, se accede al documento mediante la variable **pathDocumento** y se actualiza el array existente en la base de datos por la nueva **listaEjercicios** (la cual ya no contiene el ejercicio indicado).



#### añadirEjercicios()

Esta función se encarga de añadir todos los ejercicios deseados a la **listaEjercicios** de rutina, tanto en la aplicación como en la base de datos de Firebase Cloud Firestore. Para ello, se le pasa como parámetro la lista de ejercicios a añadir y se comprueba si alguno de estos ejercicios ya se encontraba añadido a la rutina (para evitar duplicados). Una vez se ha realizado la comprobación, accede al documento deseado a través de la propiedad **pathDocumento** y se actualiza el campo **listaEjercicios** del documento.



## Pantallas

### Login

La pantalla de inicio de sesión (**LoginScreen**) implementada en esta aplicación tiene como objetivo permitir a los usuarios acceder a sus cuentas existentes para utilizar las funcionalidades de la aplicación.



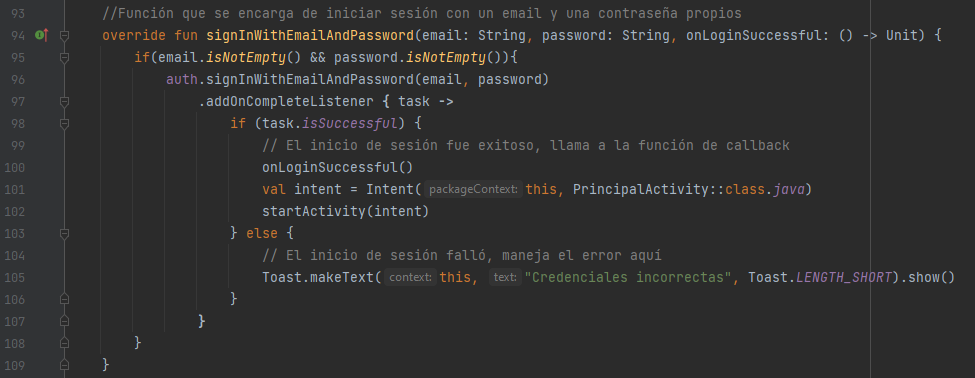
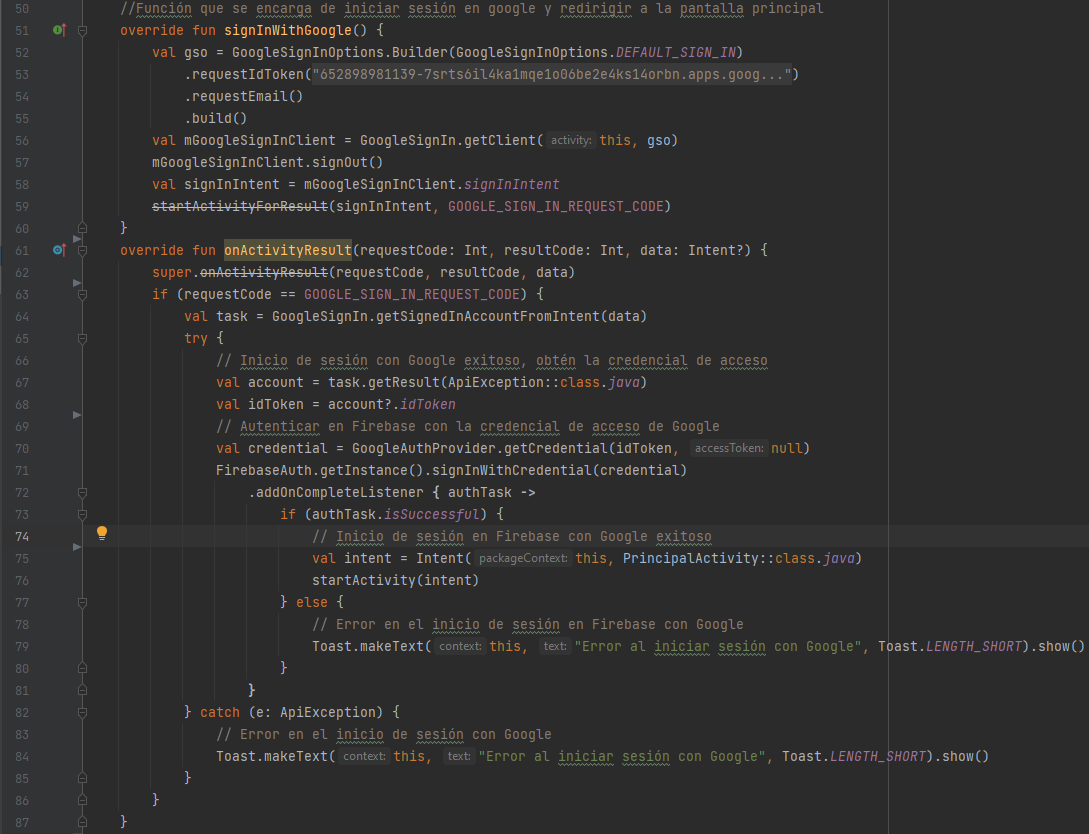
Ilustración 4. Pantalla de Login

Esta pantalla se compone de 3 clases:

1. LoginActivity: Se encarga de implementar los métodos de la interfaz loginActions, mostrar LoginScreen y de comunicarse con otras pantallas mediante intents.
2. LoginActions: Se encarga de definir las funciones necesarias y comunicar loginActivity con LoginScreen.
3. LoginScreen: Se encarga de mostrar los elementos visuales creados con Jetpack Compose e invocar los métodos de LoginActivity a través de LoginActions cuando sea necesario.

A continuación, se describe cómo funciona esta pantalla:

La pantalla se compone de varios elementos que facilitan la interacción del usuario con el formulario de inicio de sesión:

* Campo de correo electrónico: Este campo permite al usuario ingresar su dirección de correo electrónico.
* Campo de contraseña: Aquí, el usuario puede ingresar su contraseña para la autenticación. Además, se incluye un icono que permite alternar entre mostrar u ocultar la contraseña.
* Botón de inicio de sesión: Al hacer clic en este botón, se invoca la función **signInWithEmailAndPassword** del objeto **loginActions**. Esta función se encarga de autenticar al usuario utilizando el correo electrónico y la contraseña proporcionados. Si la autenticación es exitosa, se muestra un mensaje de "Inicio de sesión exitoso".   
    
  
* Botón de inicio de sesión con Google: Al hacer clic en este botón, se invoca la función **signInWithGoogle** del objeto **loginActions**. Esta función inicia el proceso de inicio de sesión con Google utilizando la API de autenticación de Google. Si el inicio de sesión es exitoso, el usuario es redirigido a la pantalla principal de la aplicación.   
    
  
* Texto y botón de registro: Se muestra un texto que indica "¿No tienes una cuenta?" y un botón "Regístrate". Al hacer clic en este botón, se invoca la función **navRegister** del objeto **loginActions**, que redirige al usuario a la pantalla de registro ([Registro](#_Registro)).

Además de estos elementos, la pantalla también utiliza el componente **Scaffold** de Jetpack Compose para estructurar el diseño general. La función **setContent** se encarga de establecer el contenido de la actividad utilizando la función **LoginScreen** como punto de entrada.

En la clase **LoginActivity**, que actúa como controlador de la pantalla, se implementan las acciones definidas en la interfaz **LoginActions**. Estas acciones se comunican con la pantalla de inicio de sesión para realizar las operaciones de inicio de sesión correspondientes, como el inicio de sesión con correo electrónico y contraseña, y el inicio de sesión con Google. Además, se manejan las respuestas de los eventos, como la autenticación exitosa o los errores de inicio de sesión.

### Registro

La pantalla de registro (**RegisterScreen**) implementada en esta aplicación tiene como objetivo permitir a los usuarios crear nuevas cuentas para acceder a las funcionalidades de la aplicación. A continuación, se describe cómo funciona esta pantalla:

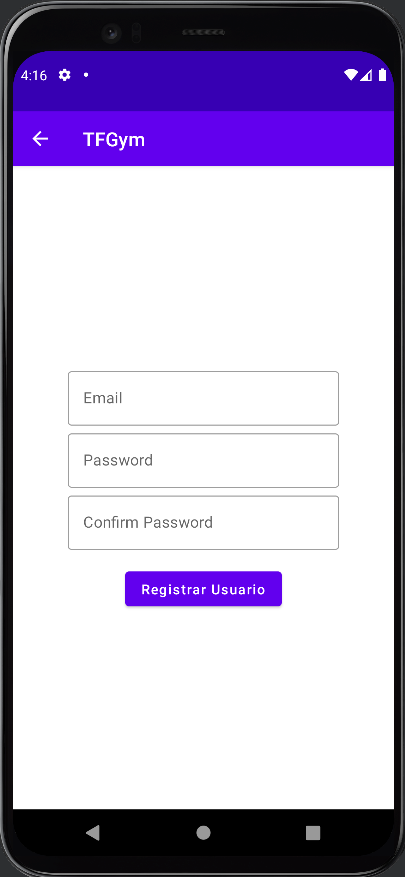


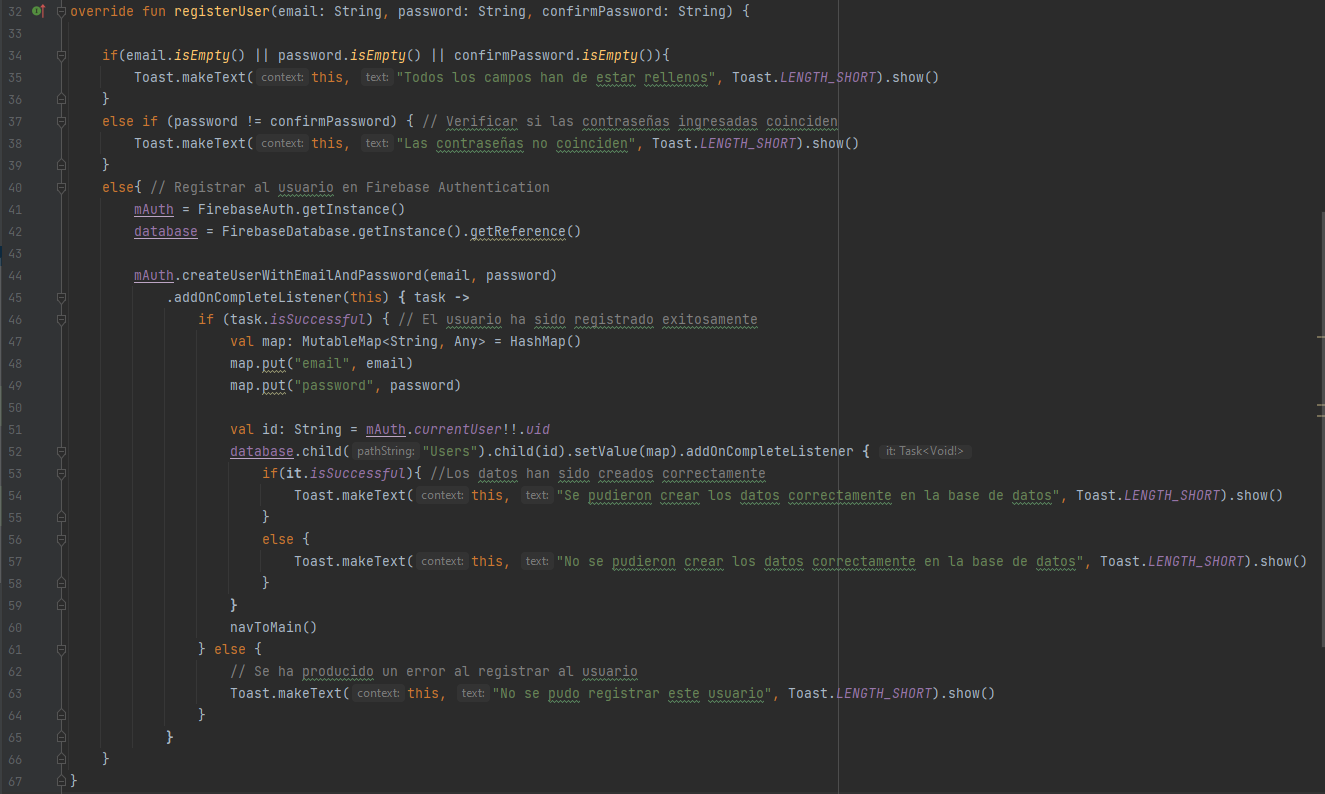
Ilustración 5. Pantalla de Registro

Esta pantalla se compone de 3 clases:

1. RegisterActivity: Se encarga de implementar los métodos de la interfaz RegisterActions, mostrar RegisterScreen y de comunicarse con otras pantallas mediante intents.
2. RegisterActions: Se encarga de definir las funciones necesarias y comunicar RegisterActivity con RegisterScreen.
3. RegisterScreen: Se encarga de mostrar los elementos visuales creados con Jetpack Compose e invocar los métodos de RegisterActivity a través de RegisterActions cuando sea necesario.

La pantalla de registro se compone de los siguientes elementos:

* Campo de correo electrónico: Este campo permite al usuario ingresar su dirección de correo electrónico.
* Campo de contraseña: Aquí, el usuario puede ingresar su contraseña para la creación de la cuenta. Al igual que en la pantalla de inicio de sesión, se incluye un ícono que permite alternar entre mostrar u ocultar la contraseña.
* Campo de confirmación de contraseña: Este campo permite al usuario ingresar nuevamente la contraseña para confirmarla y asegurarse de que coincide con la contraseña anteriormente ingresada.
* Botón de volver a la pantalla de inicio de sesión: Se muestra un botón con la opción de "Volver a inicio de sesión". Al hacer clic en este botón, se invoca la función **volverLogin** del objeto **registerActions**, que redirige al usuario a la pantalla de inicio de sesión.
* Botón de registro: Al hacer clic en este botón, se invoca la función **registerUser** del objeto **registerActions**. Esta función se encarga de procesar los datos ingresados por el usuario y realizar el registro en la base de datos. Si el registro es exitoso, se muestra un mensaje de "Registro exitoso" y se redirige al usuario a la pantalla principal de la aplicación.



Al igual que en la pantalla de inicio de sesión, la pantalla de registro utiliza el componente **Scaffold** de Jetpack Compose para estructurar el diseño general. La función **setContent** se encarga de establecer el contenido de la actividad utilizando la función **RegisterScreen** como punto de entrada.

En la clase **RegisterActivity**, que actúa como controlador de la pantalla, se implementan las acciones definidas en la interfaz **RegisterActions**. Estas acciones se comunican con la pantalla de registro para realizar las operaciones de registro correspondientes, como la creación de la cuenta en Firebase Authentication y el almacenamiento de los datos en Firebase Realtime Database. Además, se manejan las respuestas de los eventos, como el registro exitoso o los errores durante el proceso de registro.

En resumen, la pantalla de registro permite a los usuarios crear nuevas cuentas en la aplicación de gimnasio proporcionando su correo electrónico, contraseña y confirmación de contraseña. Proporciona una interfaz intuitiva y fluida para facilitar el proceso de registro y la integración con Firebase para almacenar los datos de usuario.

### Principal

La clase **PrincipalScreen** representa la pantalla principal de la aplicación de gimnasio, donde se muestran varias opciones de navegación para acceder a diferentes funcionalidades. A continuación, describiré cómo funciona esta pantalla:



Ilustración 6. Pantalla Principal

Esta pantalla se compone de 3 clases:

1. PrincipalActivity: Se encarga de implementar los métodos de la interfaz MainNav, mostrar PrincipalScreen y de comunicarse con otras pantallas mediante intents.
2. MainNav: Se encarga de definir las funciones necesarias y comunicar PrincipalActivity con PrincipalScreen.
3. RegisterScreen: Se encarga de mostrar los elementos visuales creados con Jetpack Compose e invocar los métodos de PrincipalActivity a través de MainNav cuando sea necesario.

La pantalla se compone de los siguientes elementos:

* Barra de navegación superior: Utiliza el componente **TopAppBar** de Jetpack Compose para mostrar el título de la aplicación y un ícono de flecha hacia atrás. Al hacer clic en el ícono de flecha hacia atrás, se invoca la función **cerrarSesion** del objeto **mainNav** para cerrar la sesión actual y redirigir al usuario a la pantalla de inicio de sesión.
* Columna de opciones de navegación: Utiliza el componente **Column** para organizar las opciones de navegación en forma de lista vertical. Cada opción está envuelta en un **Box** que proporciona un área de pulsación amplia y un fondo de color correspondiente.
* Opción “Rutinas": Al hacer clic en esta opción, se invoca la función **navRutinas** del objeto **mainNav**, que redirige al usuario a la pantalla de rutinas.
* Opción "Nutrición": Al hacer clic en esta opción, se invoca la función **navNutricion** del objeto **mainNav**, que redirige al usuario a la pantalla de nutrición.
* Opción "Calendario de ejercicios": Al hacer clic en esta opción, se invoca la función **navCalendario** del objeto **mainNav**, que redirige al usuario a la pantalla de calendario de ejercicios.

En la clase **PrincipalActivity**, que actúa como controlador de la pantalla, se implementan las acciones definidas en la interfaz **MainNav**. Estas acciones se comunican con la pantalla principal para realizar las operaciones de navegación correspondientes, como redirigir al usuario a las pantallas de rutinas, nutrición y calendario de ejercicios, o cerrar la sesión actual. Además, se maneja la respuesta del evento de cierre de sesión, donde se utiliza FirebaseAuth para cerrar la sesión y se redirige al usuario a la pantalla de inicio de sesión.

En resumen, la pantalla principal (**PrincipalScreen**) muestra opciones de navegación para acceder a diferentes funcionalidades de la aplicación de gimnasio. Proporciona una interfaz de usuario intuitiva y fluida para facilitar la navegación y la interacción con las diferentes secciones de la aplicación.

### Rutinas

La clase **RutinaScreen** representa la pantalla de visualización de rutinas en la aplicación de gimnasio. A continuación, describiré cómo funciona esta pantalla:



Ilustración 7. Pantalla de Rutinas

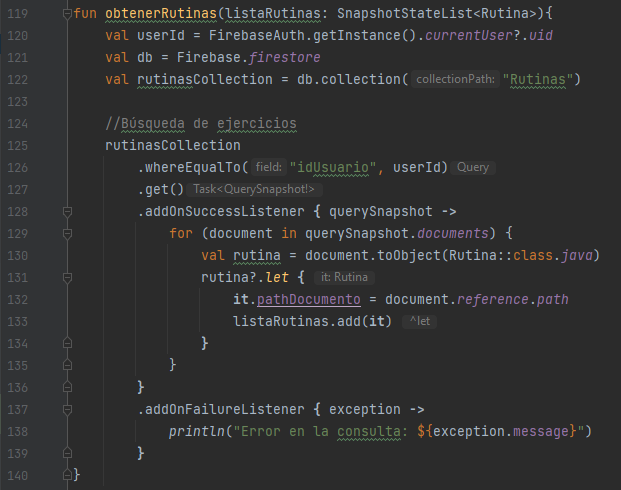
Esta pantalla se compone de 3 clases:

1. RutinasActivity: Se encarga de implementar los métodos de la interfaz RutinasActions, mostrar RutinasScreen y de comunicarse con otras pantallas mediante intents.
2. RutinasActions: Se encarga de definir las funciones necesarias y comunicar RutinasActivity con RutinasScreen.
3. RutinasScreen: Se encarga de mostrar los elementos visuales creados con Jetpack Compose e invocar los métodos de RutinasActivity a través de RutinasActions cuando sea necesario.

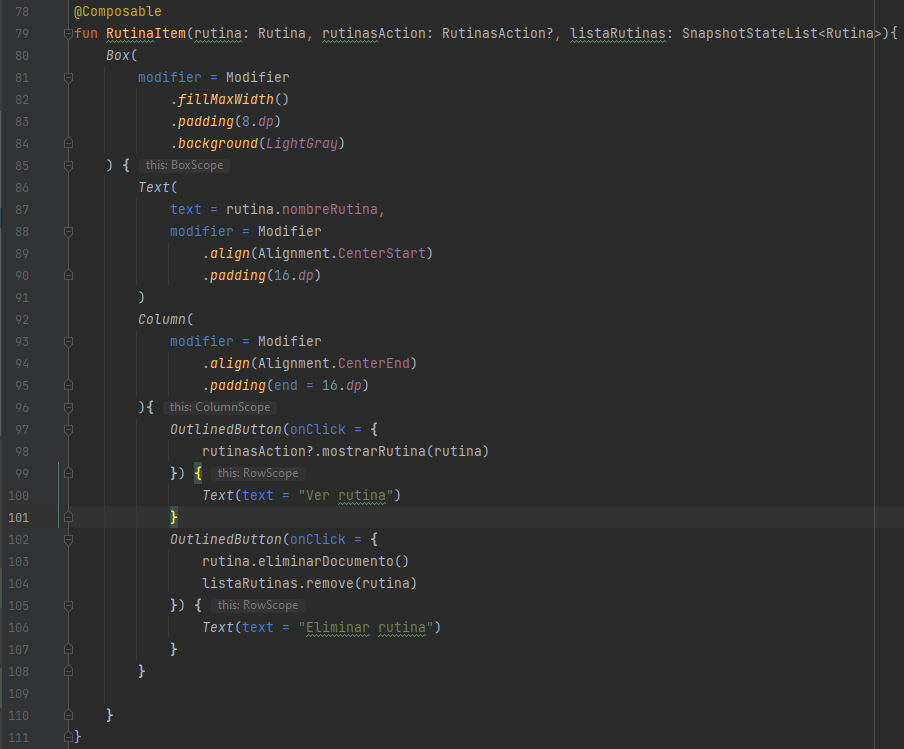
La pantalla se compone de los siguientes elementos:

* Lista de rutinas: Utiliza el componente **LazyColumn** de Jetpack Compose para mostrar una lista vertical de rutinas. Cada rutina se representa mediante el componente **RutinaItem**, que muestra el nombre de la rutina y proporciona opciones para ver la rutina y eliminarla.
* Botón flotante para crear una nueva rutina: Utiliza el componente **FloatingActionButton** para mostrar un botón flotante con un ícono de "+" que representa la acción de crear una nueva rutina. Al hacer clic en este botón, se invoca la función **crearRutina()** de la interfaz **RutinasAction** para redirigir al usuario a la pantalla de selección de ejercicios [Ejercicios](#_Ejercicios).
* Barra de navegación superior: Utiliza el componente **TopAppBar** para mostrar el título de la aplicación. Además, incluye un ícono de flecha hacia atrás que permite al usuario regresar a la pantalla principal.

Al iniciar la actividad que contiene esta pantalla, se llama a la función **obtenerRutinas()** para obtener las rutinas del usuario desde Firebase Firestore. Estas rutinas se almacenan en una lista mutable (**listaRutinas**) que se utiliza para mostrar la información en la pantalla.



Cada rutina en la lista se muestra mediante el componente **RutinaItem**, que muestra el nombre de la rutina y proporciona dos opciones: "Ver rutina" y "Eliminar rutina". Al hacer clic en "Ver rutina", se invoca la función **mostrarRutina(rutina)** de la interfaz **RutinasAction**, lo que redirige al usuario a la pantalla de visualización detallada de la rutina seleccionada. Al hacer clic en "Eliminar rutina", se invoca la función **eliminarDocumento()** de la rutina correspondiente, lo que elimina la rutina de Firebase Firestore y la remueve de la lista **listaRutinas**.



### verRutina

La clase **verRutinaScreen** representa la pantalla de visualización detallada de cada rutina en la aplicación de gimnasio. Para ello, recibe como parámetros una rutina, verRutinaActions y un remitente. A continuación, describiré cómo funciona esta pantalla:

La pantalla se compone de los siguientes elementos:

* Barra de navegación superior: Utiliza el componente **TopAppBar** para mostrar el título de la aplicación. Además, incluye un ícono de flecha hacia atrás que permite al usuario regresar a la pantalla anterior (puede ser la pantalla de calendario o la lista de rutinas, dependiendo del remitente).
* Botón flotante para añadir ejercicios: Utiliza el componente FloatingActionButton para mostrar un botón flotante con un ícono de "+" que representa la acción de añadir ejercicios a la rutina. Al hacer clic en este botón, se invoca la función **añadirEjercicio(rutina)** de la interfaz **verRutinaAction**, lo que redirige al usuario a la pantalla de selección de ejercicios (**añadirEjerciciosScreen**, es diferente a la pantalla para crear rutinas **EjerciciosScreen**) para agregar nuevos ejercicios a la rutina actual.
* Nombre de la rutina: Muestra el nombre de la rutina actual en un componente Text con un tamaño de fuente de 20.sp. Este componente está rodeado de un espacio inferior (**padding**) de 16.dp.
* Lista de ejercicios de la rutina: Utiliza el componente **LazyColumn** de Jetpack Compose para mostrar una lista vertical de ejercicios pertenecientes a la rutina actual. Cada ejercicio se representa mediante el componente **EjercicioItem**, que muestra el nombre del ejercicio, una imagen de este y dos botones: "Ver ejercicio" y "Eliminar ejercicio".

Al iniciar la actividad que contiene esta pantalla, se obtiene la rutina actual a través de un **intent** y se almacena en la variable **rutina**. Además, se obtiene un valor booleano llamado **remitente** que indica la pantalla desde la cual se accedió a **verRutinaScreen** (si es verdadero, indica que se accedió desde la pantalla de rutinas; si es falso, indica que se accedió desde la pantalla de calendario). La variable remitente se utiliza para saber a que pantalla habría que volver en caso de que el usuario pulse el botón para volver a la pantalla anterior.

La lista de ejercicios de la rutina se almacena en una SnapshotStateList mutable llamada "listaEjercicios", que permite actualizarla. Esta lista se pasa como parámetro al componente LazyColumn, que se encarga de mostrar cada ejercicio mediante el componente EjercicioItem.

El componente **EjercicioItem** muestra el nombre del ejercicio y una imagen utilizando el componente **Image** de **Coil** para cargar y mostrar la imagen desde la URL correspondiente al ejercicio. Además, incluye dos botones: "Ver ejercicio" y "Eliminar ejercicio". Al hacer clic en "Ver ejercicio", se invoca la función mostrarEjercicio(ejercicio) de la interfaz verRutinaAction, lo que redirige al usuario a la pantalla de visualización detallada de ese ejercicio. Al hacer clic en "Eliminar ejercicio", se invoca la función eliminarEjercicio(ejercicio) de la rutina actual, lo que elimina el ejercicio de la rutina y lo remueve de la lista "listaEjercicios".

En resumen, la pantalla "verRutina" muestra el nombre de una rutina y una lista de ejercicios asociados a esa rutina. Permite al usuario ver los detalles de cada ejercicio y eliminarlos de la rutina. Además, proporciona un botón flotante para añadir nuevos ejercicios a la rutina actual.

### añadirEjercicios

### Ejercicios

### verEjercicio

### Nutricion

### Alimento

### Calendario

### añadirRutinaCalendario.

## Otros detalles

## Tutorial de la aplicación

Para comenzar a utilizar la aplicación lo primero que hay que hacer es iniciar sesión o registrarse en caso de que no haya una cuenta existente. La primera pantalla de la aplicación es la pantalla de [Login](#_Login), la cual cuenta con un botón para acceder a la pantalla de registro en caso de que no te hayas registrado anteriormente.

Para registrarte, debes acceder a la pantalla de [Registro](#_Registro) y rellenar los campos correspondientes. Una vez rellenados los campos, el usuario deberá darla al botón de “Registrar Usuario” para acceder a la aplicación.

Si ya se ha registrado el usuario con anterioridad, deberá iniciar sesión desde la pantalla de Login, ya sea usando una cuenta de Google como con el email y contraseña que utilizó para registrarse (se ha de tener en cuenta que aunque se use el mismo correo electrónico tanto para registrarse como para iniciar sesión en Google, las cuentas serán diferentes).

Una vez el usuario se ha registrado o ha iniciado sesión, pasará a la pantalla [Principal](#_Principal), desde la cual podrá acceder a otras tres pantallas: [Rutinas](#_Rutinas), [Nutrición](#_Nutricion) y [Calendario](#_Calendario).

En la pantalla de [Rutinas](#_Rutinas) el usuario será capaz de visualizar las rutinas existentes y crear nuevas rutinas a través de un botón flotante

# **Conclusiones y mejoras**

# **Referencias**

[Firebase Authentication: Email & Password and Register](https://firebase.google.com/docs/auth/web/password-auth?hl=es-419#web-namespaced-api)

[Firebase Google Authentication](https://firebase.google.com/docs/auth/android/google-signin?hl=es)

https://developer.android.com/courses/fundamentals-training/toc-v2?hl=es-419