

PawFriends App
Documentación Técnica



App para gestión de mascotas e interacción entre usuarios

DESCRIPCIÓN DEL PROYECTO

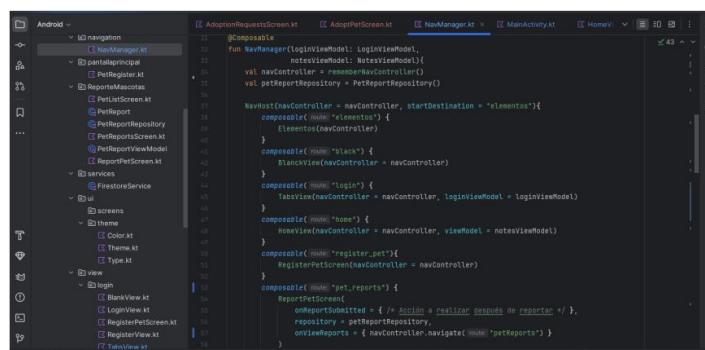
PawFriends es una aplicación diseñada para conectar a dueños de mascotas y amantes de los animales en una comunidad de apoyo mutuo. Los usuarios pueden registrarse para reportar mascotas perdidas, ofrecer mascotas en adopción, solicitar y ofrecer servicios de cuidado temporal, y acceder a recursos para el bienestar animal. La aplicación está desarrollada en Android Studio utilizando Jetpack Compose para la interfaz y Firebase para la autenticación y el almacenamiento de datos.

Ruta de navegación: NavManager

El código NavManager es el corazón de la navegación en la aplicación. Define la estructura y el flujo de las diferentes pantallas o vistas que el usuario puede visitar. Imagina que es un mapa de la aplicación, indicando los caminos que el usuario puede tomar para moverse de una pantalla a otra.

Componentes Clave y Funcionalidades

- ✓ **NavHost:** Es el contenedor principal de la navegación. Piensa en él como una estación central de trenes donde parten todos los trenes (en este caso, las rutas hacia diferentes pantallas).
- ✓ **composable:** Cada composable define una pantalla específica. Es como una estación de tren con un nombre único.
- ✓ **navController:** Es el conductor del tren. Te permite navegar entre las diferentes estaciones (pantallas).
- ✓ **Rutas:** Cada composable tiene una ruta asociada. Es como el destino del tren. Por ejemplo, "login" es la ruta para la pantalla de inicio de sesión.
- ✓ **Parámetros:** Algunos composables aceptan parámetros. Estos parámetros se utilizan para pasar datos entre pantallas. Por ejemplo,



The screenshot shows the Android Studio interface with the 'NavManager.kt' file open in the navigation graph editor. The code defines a navigation graph with several nodes and their associated routes:

```
graph LR; Start(( )) --> Login[Login]; Login --> Register[RegisterPet]; Register --> Home[Home]; Home --> PetList[PetList]; PetList --> Report[Report]; Report --> Details[Details]; Details --> Logout[Logout];
```

The 'NavManager.kt' file also includes logic for handling navigation events and managing the NavController.

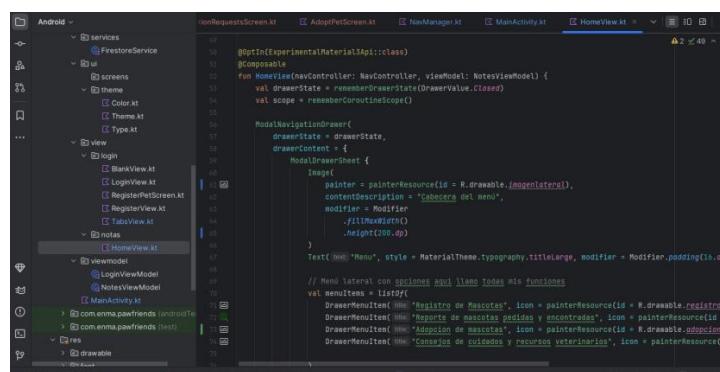
el ID de una solicitud de adopción se pasa a la pantalla de detalles de la adopción.

Union de los Items(HomeView):

La función HomeView es la encargada de definir la pantalla principal de la aplicación, es decir, lo primero que ve el usuario al iniciar sesión o registrarse. Esta pantalla suele ser el punto de partida para acceder a otras secciones de la app. En este caso particular, HomeView proporciona un menú lateral para navegar a diferentes funcionalidades y una interfaz principal con un botón para explorar opciones de adopción.

Componentes Clave y Funcionalidades:

- ✓ **Cajón de navegación (Navigation Drawer):** Permite acceder a diferentes secciones de la app deslizando un menú desde el borde de la pantalla. Incluye opciones como registro de mascotas, reportes, adopción, consejos y servicios. Al seleccionar una opción, la pantalla se actualiza para mostrar el contenido correspondiente.
- ✓ **Scaffold:** Proporciona una estructura básica para la pantalla, incluyendo una barra superior, una barra inferior y un área de contenido principal.
- ✓ **Barra superior:** Muestra el título de la app ("Paw Friends") y botones para abrir el menú lateral y cerrar sesión.
- ✓ **Barra inferior:** Contiene iconos para navegar rápidamente entre las secciones principales de la app (inicio, consejos, mensajería).
- ✓ **Área de contenido principal:** Muestra un mensaje de bienvenida y un botón para explorar opciones de adopción. Puede contener otros elementos según las funcionalidades de la app.
- ✓ **Flujo de Datos:**
 - El NavController se utiliza para navegar entre las diferentes pantallas de la aplicación.
 - El viewModel (en este caso, NotesViewModel) podría utilizarse para gestionar datos relacionados con notas o cualquier otra funcionalidad específica de la pantalla.



1. REGISTRO Y PERFIL DE USUARIO:

Los usuarios podrán registrarse y crear perfiles donde detallen su experiencia con mascotas (si tienen, qué tipo de mascota, etc.), sus intereses (adopción, cuidado temporal, búsqueda de mascotas perdidas) y los servicios que pueden ofrecer o solicitar.

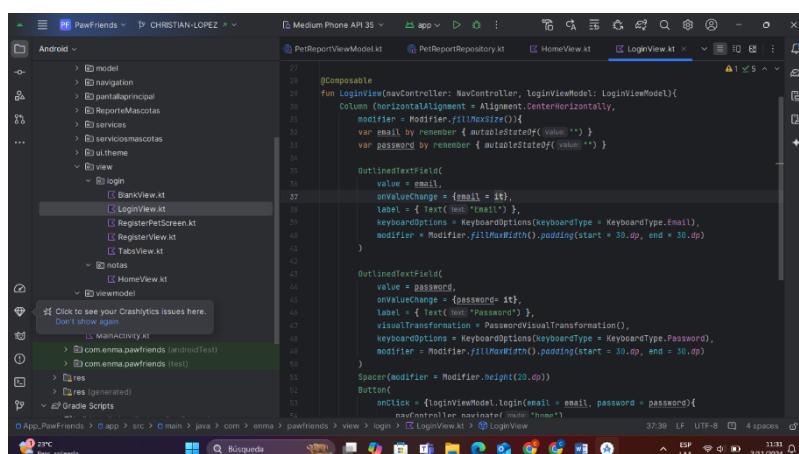
LoginView: Este archivo define una pantalla de inicio de sesión (LoginView) en la aplicación usando Jetpack Compose. La función principal de esta pantalla es permitir que el usuario ingrese su correo electrónico y contraseña para iniciar sesión.

Componentes:

- ✓ **OutlinedTextField para Email y Contraseña:** Campos de entrada donde el usuario ingresa su correo electrónico y contraseña.
- ✓ **onValueChange:** Actualiza el valor en email y password a medida que el usuario escribe.
- ✓ **keyboardOptions:** Configure el teclado, como el tipo de teclado para el correo y la contraseña (texto y contraseña).
- ✓ **visualTransformation:** En el campo de la contraseña, se aplica para ocultar el texto (por ejemplo, se muestra *en lugar de los caracteres).
- ✓ **Botón:** El botón llama al método login del LoginViewModel, pasando los valores de email y password para autenticar al usuario. Navega hacia una llamada de pantalla home en caso de autenticación exitosa.

Uso del ViewModel:

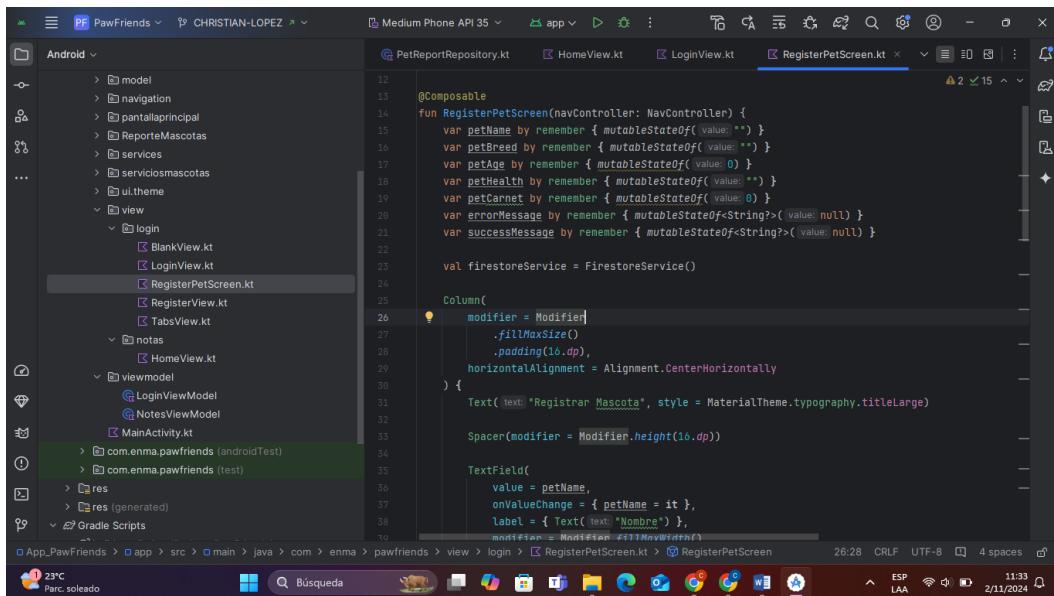
LoginViewModel maneja la lógica de autenticación y se pasa como parámetro a la pantalla.



RegisterPetScreen: Este archivo define una pantalla para registrar una mascota en el sistema (RegisterPetScreen), usando Jetpack Compose.

Componentes:

- ✓ **Variables de Estado (remember):** Se definen varias variables (como petName, petBreed, petAge, etc.) para almacenar los datos de la mascota que el usuario ingresa. Se usan mutableStateOf remember para permitir la reactividad en los campos de entrada.
- ✓ **Servicio Firestore:** Se crea una instancia de FirestoreService para interactuar con Firestore, que es la base de datos en la que se almacenan los datos de la mascota.
- ✓ **Columna y campos de texto:** Estructura de disposición que organiza los campos de entrada para el nombre, raza, edad, salud y otros detalles de la mascota. Cada TextField actualiza su valor correspondiente usando onValueChange.



```
12 @Composable
13 fun RegisterPetScreen(navController: NavController) {
14     var petName by remember { mutableStateOf(" ") }
15     var petBreed by remember { mutableStateOf(" ") }
16     var petAge by remember { mutableStateOf(0) }
17     var petHealth by remember { mutableStateOf("") }
18     var petCarpet by remember { mutableStateOf(false) }
19     var errorMessage by remember { mutableStateOf<String?>(null) }
20     var successMessage by remember { mutableStateOf<String?>(null) }
21
22     val firestoreService = FirestoreService()
23
24     Column(
25         modifier = Modifier
26             .fillMaxSize()
27             .padding(16.dp),
28             horizontalAlignment = Alignment.CenterHorizontally
29     ) {
30         Text(text = "Registrar Mascota", style = MaterialTheme.typography.titleLarge)
31
32         Spacer(modifier = Modifier.height(16.dp))
33
34         TextField(
35             value = petName,
36             onValueChange = { petName = it },
37             label = { Text(text = "Nombre") },
38             modifier = Modifier.fillMaxWidth()
39     )
40 }
```

- ✓ **RegisterView:** El código define una función componible llamada RegisterView en Kotlin, que se utiliza típicamente en aplicaciones Android desarrolladas con Jetpack Compose. Esta función es responsable de crear la interfaz de usuario para la pantalla de registro de una aplicación.

Componentes y Funcionalidades Principales

- ✓ **Campos de entrada:** Se utilizan tres OutlinedTextField para capturar el nombre de usuario, correo electrónico y contraseña del usuario. Los modificadores fillMaxWidth y padding se utilizan para ajustar el tamaño y el espaciado de los campos. Las opciones del teclado se configuran para optimizar la entrada (por ejemplo, teclado numérico para la contraseña).

- ✓ **Botón de registro:** Un botón Button inicia el proceso de registro cuando se hace clic en él. Al hacer clic, se invoca la función register del loginViewModel, pasando los datos del formulario. Si el registro es exitoso, se navega a la pantalla de inicio ("home").
- ✓ **Manejo de errores:** Se utiliza un Alert para mostrar un mensaje de error si el registro falla. El estado del Alert se controla mediante la propiedad showAlert del loginViewModel.

```

@Composable
fun RegisterView(navController: NavController, loginViewModel: LoginViewModel){
    Column (horizontalAlignment = Alignment.CenterHorizontally,
        modifier = Modifier.fillMaxSize()){
        var email by remember { mutableStateOf("") }
        var password by remember { mutableStateOf("") }
        var userName by remember { mutableStateOf("") }

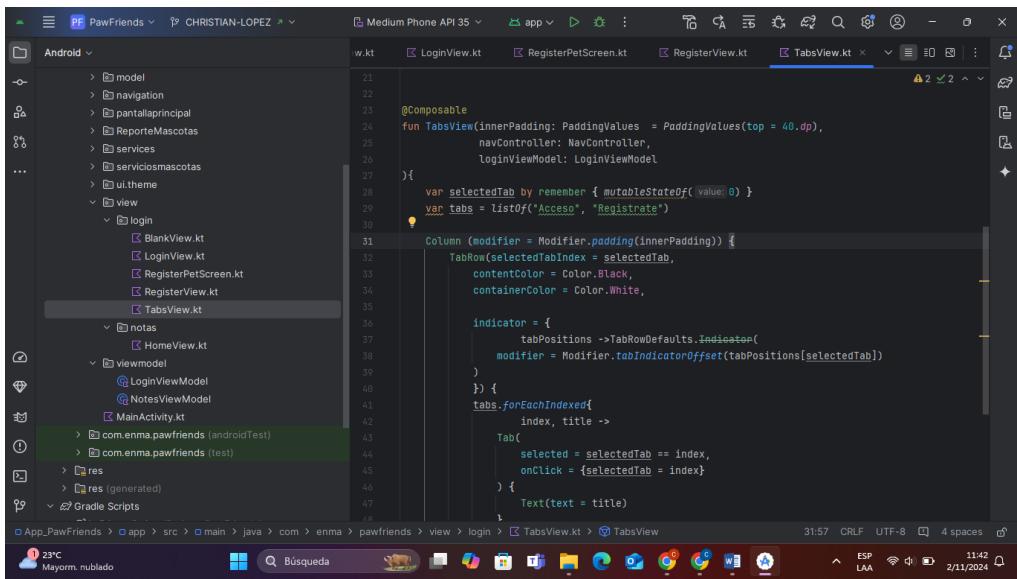
        OutlinedTextField(
            value = userName,
            onValueChange = {userName = it},
            label = { Text( text= "Nombre") },
            modifier = Modifier
                .fillMaxWidth()
                .padding(start = 30.dp, end = 30.dp)
        )
        OutlinedTextField(
            value = email,
            onValueChange = {email= it},
            label = { Text( text= "Email") },
            keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Email),
            modifier = Modifier
                .fillMaxWidth()
                .padding(start = 30.dp, end = 30.dp)
        )
    }
}

```

TabsView: Este archivo define una vista con pestañas (TabsView) que permite alternar entre diferentes secciones, como "Acceso" y "Regístrate".

Componentes:

- ✓ **Estado de la Pestaña Seleccionada:** La variable selectedTab rastrea qué pestaña está seleccionada actualmente. Se actualiza con el índice de la pestaña seleccionada por el usuario.
- ✓ **Fila de pestañas:** Representa el contenedor de pestañas, y permite mostrar las pestañas "Acceso" y "Regístrate".
- ✓ **selectedTabIndex:** Indica cuál pestaña está activa en el TabRow.
- ✓ **indicador:** Muestra una indicación visual en la pestaña seleccionada.
- ✓ **Pestaña:** Cada pestaña muestra un texto (title) y, al ser seleccionado, se actualiza selectedTab para reflejar el cambio de pestaña.



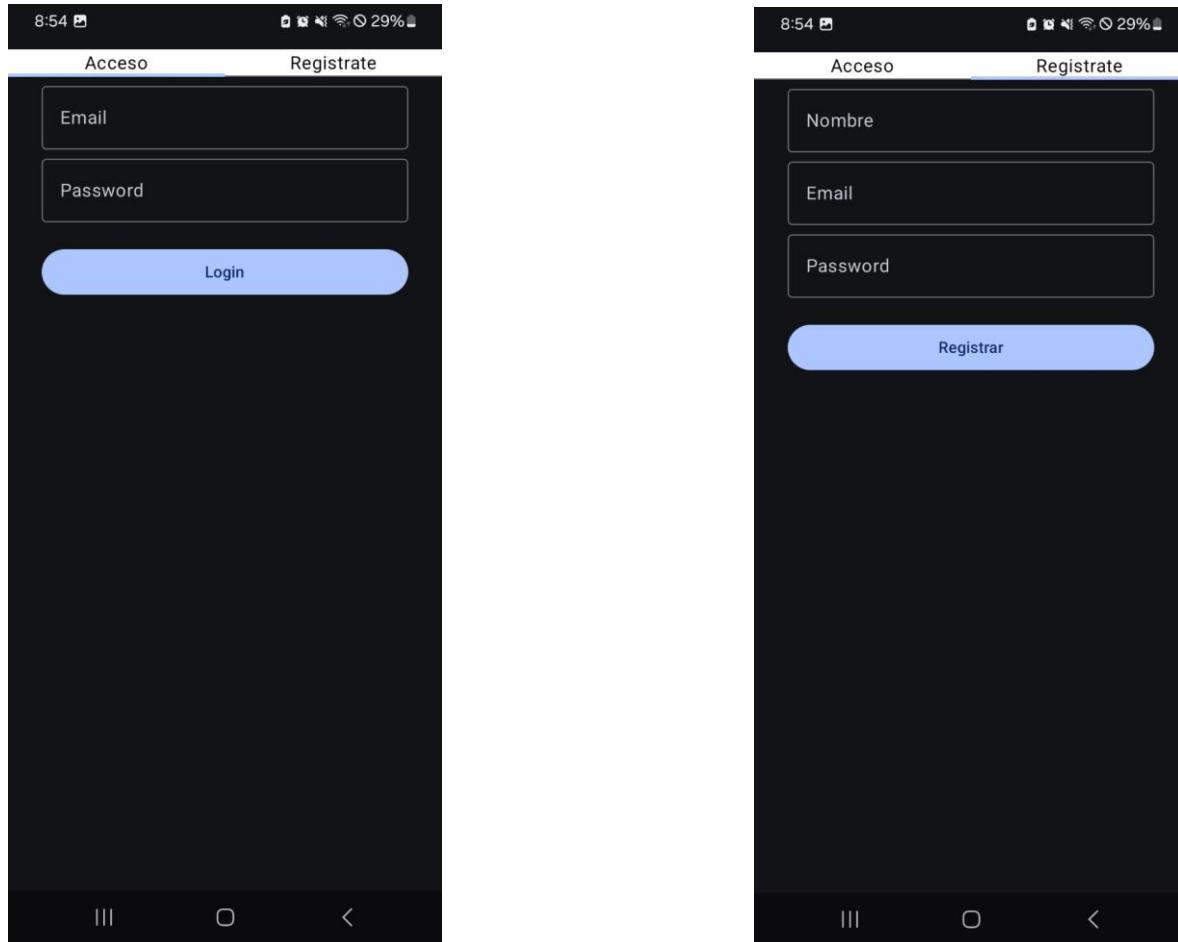
```

@Composable
fun TabsView(innerPadding: PaddingValues = PaddingValues(top = 40.dp),
    navController: NavController,
    loginViewModel: LoginViewModel)
{
    var selectedTab by remember { mutableStateOf(value: 0) }
    val tabs = listOf("Acceso", "Registrate")

    Column(modifier = Modifier.padding(innerPadding)) {
        TabRow(selectedTabIndex = selectedTab,
            contentColor = Color.Black,
            containerColor = Color.White,
            indicator = {
                TabRowDefaults.Indicator(
                    modifier = Modifier.tabIndicatorOffset(tabPositions[selectedTab])
                )
            })
        tabs.forEachIndexed { index, title ->
            Tab(
                selected = selectedTab == index,
                onClick = {selectedTab = index}
            ) {
                Text(text = title)
            }
        }
    }
}

```

Aspecto Final del Apartado “REGISTRO Y PERFIL DE USUARIO”



2. REGISTRO Y GESTIÓN DE MASCOTAS:

- Los usuarios podrán agregar mascotas a su perfil, proporcionando detalles como nombre, raza, edad, salud, e historial de vacunación.
- Se permitirá cargar imágenes y videos de las mascotas para que otros usuarios las vean.

REGISTRO DE MASCOTAS

La aplicación '**Paw Friends**' permite a los usuarios gestionar y registrar mascotas proporcionando detalles como nombre, raza, edad, salud, historial de vacunación, y además permite subir imágenes y videos de las mascotas.

Modelo de datos Pet.kt

El modelo Pet representa los datos de cada mascota.

Atributos:

- name: Nombre de la mascota.
- breed: Raza.
- age: Edad.
- health: Estado de salud.
- carnet: Carnet de vacunación.
- ownerId: ID del propietario (generalmente su correo electrónico).
- imageUrl: URL de la imagen subida a Firebase Storage.
- videoUrl: URL del video subido a Firebase Storage (opcional).

```
NavManager.kt      FirestoreService.kt      MainActivity.kt      MascotasScreen.kt
1 package com.enma.pawfriends.model
2
3 data class Pet(
4     val name: String = "",
5     val breed: String = "",
6     val age: Int = 0,
7     val health: String = "",
8     val carnet: Int = 0,
9     val ownerId: String = "",
10    val imageUrl: String = "", // URL de la imagen subida
11    val videoUrl: String = "" // URL del video (opcional)
12 )
13
```

PANTALLA PARA MOSTRAR MASCOTAS REGISTRADAS: MASCOTASSCREEN.KT

La pantalla MascotasScreen.kt tiene como objetivo mostrar una lista de mascotas que los usuarios han registrado previamente en Firebase Firestore.

Utiliza Jetpack Compose para crear una interfaz interactiva que incluye detalles importantes de cada mascota, como su nombre, raza, edad, estado de salud, carnet de vacunación, una imagen y el correo electrónico del dueño.

Variables de Estado

Estas variables de estado en Kotlin con **Jetpack Compose** son utilizadas para manejar el comportamiento dinámico de la UI en tiempo real, permitiendo que la interfaz gráfica reaccione a los cambios de datos. Vamos a desglosar su propósito:

- pets: Estado que almacena la lista de mascotas, inicialmente vacío.
- isLoading: Controla si la pantalla está en proceso de cargar datos (inicia en true).
- errorMessage: Almacena un mensaje de error si ocurre algún fallo al cargar los datos.

```
No Devices ▾ app ▾ ▶ : To CA E F
vManager.kt FirestoreService.kt MainActivit... ▫ MascotasScreen.kt ✘ Pet.kt RegisterPetScree...
10 import androidx.compose.ui.Alignment
11 import androidx.compose.ui.Modifier
12 import androidx.compose.ui.graphics.Color
13 import androidx.compose.ui.unit.dp
14 import androidx.compose.ui.unit.sp
15 import coil.compose.rememberImagePainter
16 import com.enma.pawfriends.model.Pet
17 import com.enma.pawfriends.services.FirestoreService // Asegúrate de importar FirestoreService
18
19 @Composable
20 fun MascotasScreen(firestoreService: FirestoreService) { // Cambiado a FirestoreService
21     var pets by remember { mutableStateOf<List<Pet>>(emptyList()) }
22     var isLoading by remember { mutableStateOf( value: true ) }
23     var errorMessage by remember { mutableStateOf( value: "" ) }
```

Efecto de Lanzamiento (LaunchedEffect)

LaunchedEffect es una función que se usa para ejecutar una tarea solo una vez cuando se compone la pantalla. Esto es útil para lanzar operaciones asíncronas como la carga de datos, algo que no debería suceder en cada recomposición de la UI, sino solo una vez, al iniciarse la pantalla. **Unit**: El argumento Unit indica que el bloque de código se ejecutará solo una vez, cuando la pantalla se cargue por primera vez, ya que Unit es un tipo en Kotlin que esencialmente significa "sin valor significativo". Se usa aquí para asegurar que el efecto no dependa de otros parámetros.

- **Inicio de la pantalla:**
- **Cuando la pantalla se carga por primera vez**, el bloque LaunchedEffect se ejecuta. Aquí comienza una operación de carga de datos invocando el método getPets del servicio firestoreService.
- **Llamada a firestoreService.getPets:**
- Esta es una llamada asíncrona que intenta obtener los datos de las mascotas almacenados en Firebase Firestore.
- **Dos posibles resultados:**
 - **onSuccess:** Si la operación es exitosa y los datos de las mascotas se recuperan correctamente:
 - La variable de estado pets se actualiza con los datos obtenidos (retrievedPets).
 - isLoading se establece en false, indicando que la carga ha terminado y la UI debería dejar de mostrar cualquier indicador de carga.
 - Como pets es una variable de estado, cualquier cambio en su valor provocará una recomposición de la UI, mostrando ahora la lista de mascotas.
- **onError:** Si ocurre un error durante la carga de los datos (por ejemplo, un problema de conexión o un fallo en Firestore):
 - errorMessage se actualiza con un mensaje de error que incluye la descripción de la excepción (e.message).
 - isLoading se establece en false, deteniendo el indicador de carga.
 - La UI se vuelve a componer y mostrará un mensaje de error en lugar de la lista de mascotas.

```
// Cargar los registros de mascotas
LaunchedEffect(Unit) { this: CoroutineScope
    firestoreService.getPets(
        onSuccess = { retrievedPets ->
            pets = retrievedPets
            isLoading = false
        },
        onError = { e ->
            errorMessage = "Error al cargar mascotas: ${e.message}"
            isLoading = false
        }
    )
}
```

Diseño General

Definimos en Jetpack Compose una interfaz de usuario básica utilizando un contenedor vertical llamado Column para organizar los elementos de manera vertical en la pantalla. Vamos a desglosar cada parte para entender mejor su funcionamiento y propósito.

1. Column:

- **Propósito:** El componente Column es un contenedor en el que los elementos hijos se organizan verticalmente (de arriba hacia abajo). En este caso, se utiliza para disponer los elementos como un título y otros contenidos debajo de él, uno tras otro.
- **Usos:** Es útil cuando quieras alinear componentes de manera vertical y tener control sobre la alineación y el espacio entre ellos.

2. modifier = Modifier.fillMaxSize():

- **Propósito:** Modifier.fillMaxSize() hace que el Column ocupe todo el espacio disponible en la pantalla (tanto en ancho como en alto). Esto asegura que el contenedor sea tan grande como la pantalla del dispositivo, permitiendo que los elementos dentro se distribuyan adecuadamente.
- **Usos:** Este modificador es común en layouts principales que deben ajustarse al tamaño completo de la pantalla.

3. padding(16.dp):

- **Propósito:** Este modificador añade un margen de 16dp alrededor de los elementos dentro del Column. El valor de 16.dp es una unidad de densidad independiente de la pantalla (density-independent pixels), lo que asegura que el margen se vea consistente en pantallas de diferentes tamaños y resoluciones.
- **Usos:** Este padding proporciona un espaciado adecuado alrededor de los elementos, asegurando que no se peguen a los bordes de la pantalla.

4. horizontalAlignment = Alignment.CenterHorizontally:

- **Propósito:** Alinea los elementos hijos dentro del Column horizontalmente al centro. Esto significa que cada componente dentro de la columna estará centrado horizontalmente, independientemente de su tamaño o contenido.

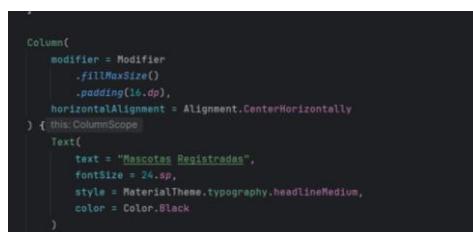
- **Usos:** Es útil para centrar visualmente los elementos dentro del contenedor.

5. Text (Título):

- **Propósito:** Este es un elemento de texto que muestra un título: "Mascotas Registradas". Se utiliza para dar contexto a la pantalla, en este caso indicando que se mostrarán las mascotas que han sido registradas.
- **Propiedades importantes:**
 - text = "Mascotas Registradas": Es el contenido que se mostrará dentro del componente de texto.
 - fontSize = 24.sp: Define el tamaño de la fuente del texto en sp (scalable pixels), que es una unidad escalable de píxeles que se ajusta en función de las configuraciones de accesibilidad del usuario (como el tamaño de fuente del sistema).
 - style = MaterialTheme.typography.headlineMedium: Aplica un estilo predefinido que viene de los temas de Material Design, lo que da consistencia visual a la tipografía en toda la aplicación.
 - color = Color.Black: Establece el color del texto en negro.

Explicación del diseño:

- La Column organiza los elementos de forma vertical, permitiendo que el título "Mascotas Registradas" se muestre en la parte superior, seguido por otros elementos (que podrían ser la lista de mascotas, por ejemplo).
- fillMaxSize() asegura que toda la pantalla se use, mientras que el padding(16.dp) añade un margen alrededor de los elementos para que no queden pegados a los bordes de la pantalla.
- Los elementos están centrados horizontalmente dentro de la columna gracias a horizontalAlignment = Alignment.CenterHorizontally, lo que da una apariencia visual equilibrada.
- El Text de "Mascotas Registradas" actúa como un encabezado destacado, que con su tamaño de fuente (24sp) y color negro, resalta y define el propósito de la pantalla.



Lista de Mascotas

Usando Jetpack Compose muestra una lista de mascotas utilizando un LazyColumn, que es un contenedor eficiente para mostrar grandes cantidades de datos sin consumir demasiados recursos, ya que solo renderiza los elementos visibles en pantalla. Veamos el propósito de cada parte del código en más detalle:

1. LazyColumn:

- **Propósito:** LazyColumn es una lista vertical que carga elementos de manera eficiente. Solo renderiza los elementos que son visibles en la pantalla y carga los demás conforme se hace scroll. Es ideal para mostrar listas de contenido dinámico o potencialmente grande, como en este caso, una lista de mascotas.
- **Uso:** Aquí, LazyColumn se utiliza para iterar sobre la lista de mascotas (pets) y mostrar cada una dentro de una tarjeta.

2. items(pets):

- **Propósito:** items es una función dentro de LazyColumn que itera sobre la lista de mascotas llamada pets. Por cada mascota en la lista, se ejecuta el bloque de código que sigue, donde se define cómo mostrar cada mascota.
- **Uso:** Permite recorrer la lista de mascotas y renderizar cada una en su propia tarjeta (Card).

3. Card:

- **Propósito:** El componente Card se utiliza para mostrar la información de cada mascota dentro de un recuadro con una ligera elevación, dándole un aspecto visual destacado y separado del fondo. Cada mascota tiene su propia tarjeta en la lista.
- **Propiedades importantes:**
 - modifier = Modifier.fillMaxWidth(): Hace que la tarjeta ocupe todo el ancho disponible de la pantalla.
 - padding(vertical = 4.dp): Añade un pequeño espacio vertical entre las tarjetas, lo que mejora la separación y legibilidad.
 - elevation = CardDefaults.cardElevation(defaultElevation = 4.dp): Añade una sombra alrededor de la tarjeta, dándole un efecto visual de elevación, lo que la hace resaltar del fondo.
 - colors = CardDefaults.cardColors(containerColor = Color.Transparent): Define que el fondo de la tarjeta sea

transparente, lo que puede ser útil si el fondo de la pantalla ya tiene un color o imagen.

4. Column(modifier = Modifier.padding(16.dp)):

- Propósito: El componente Column dentro de cada tarjeta organiza los detalles de la mascota de manera vertical (uno debajo del otro).
- padding(16.dp): Añade un margen alrededor de todos los elementos dentro de la tarjeta, asegurando que el contenido no esté pegado a los bordes de la tarjeta.

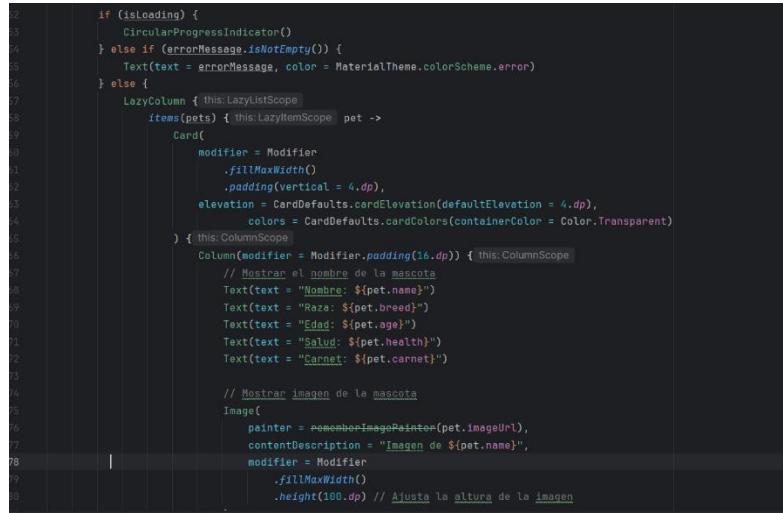
5. Text (Detalles de la Mascota):

- Propósito: Se utilizan varios componentes Text para mostrar la información de cada mascota:
 - Nombre: Muestra el nombre de la mascota (pet.name).
 - Raza: Muestra la raza de la mascota (pet.breed).
 - Edad: Muestra la edad de la mascota (pet.age).
 - Salud: Muestra el estado de salud de la mascota (pet.health).
 - Carnet: Muestra el estado del carnet de vacunación de la mascota (pet.carnet).
 - Registrado por: Muestra el correo electrónico del dueño que registró la mascota (pet.ownerId). Aquí, ownerId se refiere a algún identificador del dueño (en este caso, el correo electrónico) almacenado en la base de datos.
- Uso: Los componentes de texto simplemente despliegan cada detalle importante de la mascota en la tarjeta, organizados en un formato legible

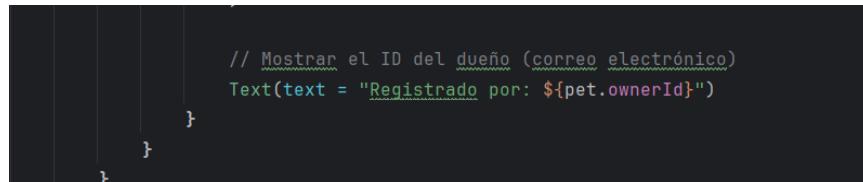
6. Image (Imagen de la Mascota):

- **Propósito:** El componente Image carga y muestra la imagen de la mascota utilizando la biblioteca **Coil**, que es una librería popular en Android para cargar imágenes desde URLs.
- **Propiedades importantes:**
 - painter = rememberImagePainter(pet.imageUrl): Utiliza rememberImagePainter para cargar la imagen desde la URL proporcionada por pet.imageUrl, que es un campo en la base de datos donde se almacena la URL de la imagen de la mascota en Firebase Storage.
 - contentDescription = "Imagen de \${pet.name)": Proporciona una descripción de la imagen para accesibilidad, describiendo que es la imagen de la mascota específica.

- modifier = Modifier.fillMaxWidth().height(100.dp): Ajusta la imagen para que ocupe todo el ancho disponible en la tarjeta, pero limita su altura a 100dp, para que la imagen no sea demasiado grande y mantenga proporciones adecuadas dentro de la tarjeta.



Text(text = "Registrado por: \${pet.ownerId}"): Muestra el correo electrónico del dueño que registró la mascota.



FIRESTORESERVICE.KT

Propiedades

- **firestore**: Crea una instancia de FirebaseFirestore, que permite realizar operaciones de lectura y escritura en la base de datos Firestore.
- **storage**: Crea una referencia a Firebase Storage, que se utiliza para cargar y acceder a archivos multimedia almacenados.

```
// FirestoreService.kt
package com.enma.pawfriends.services

import android.net.Uri
import com.enma.pawfriends.model.Pet
import com.google.firebaseio.firebaseio.FirebaseFirestore
import com.google.firebase.storage.FirebaseStorage
import java.util.UUID
|
class FirestoreService {

    private val firestore = FirebaseFirestore.getInstance()
    private val storage = FirebaseStorage.getInstance().reference
```

Métodos

Este método permite cargar un archivo multimedia (como una imagen) a Firebase Storage.

- **Parámetros:**

- uri: Uri: URI del archivo que se va a cargar.
- onSuccess: (String) -> Unit: Función de callback que se ejecuta si la carga es exitosa, recibiendo la URL de descarga del archivo.
- onError: (Exception) -> Unit: Función de callback que se ejecuta si ocurre un error durante la carga, recibiendo la excepción correspondiente.
- filePath: Define la ruta en la que se almacenará el archivo en Firebase Storage, utilizando un UUID para asegurar que el nombre del archivo sea único.
- fileRef: Crea una referencia al archivo en la ruta especificada.

```
private val firestore = FirebaseFirestore.getInstance()
private val storage = FirebaseStorage.getInstance().reference

fun uploadMediaFile(uri: Uri, onSuccess: (String) -> Unit, onError: (Exception) -> Unit) {
    val filePath = "pets/${UUID.randomUUID()}"
    val fileRef = storage.child(filePath)

    fileRef.putFile(uri)
        .addOnSuccessListener { it.UploadTask.TaskSnapshot |
            fileRef.downloadUrl.addOnSuccessListener { downloadUri -> onSuccess(downloadUri.toString()) }
        }
        .addOnFailureListener { e -> onError(e) }
}
```

- putFile(uri): Inicia la carga del archivo desde la URI proporcionada.
- addOnSuccessListener: Se ejecuta si la carga es exitosa. Obtiene la URL de descarga del archivo cargado y llama a la función onSuccess con esta URL.
- addOnFailureListener: Se ejecuta si la carga falla, llamando a onError con la excepción recibida.

```
        fileRef.putFile(uri)
            .addOnSuccessListener { it: UploadTask.TaskSnapshot!
                fileRef.downloadUrl.addOnSuccessListener { downloadUri -> onSuccess(downloadUri.toString()) }
            }
            .addOnFailureListener { e -> onError(e) }
    }
```

Este método guarda los datos de una mascota en la colección "pets" de Firestore.

□ Parámetros:

- **pet: Pet**: Objeto de tipo Pet que contiene los detalles de la mascota a guardar.
- **onSuccess: () -> Unit**: Función de callback que se ejecuta si la operación es exitosa.
- **onError: (Exception) -> Unit**: Función de callback que se ejecuta si ocurre un error al guardar la mascota, recibiendo la excepción.

```
}
```

```
fun savePet(pet: Pet, onSuccess: () -> Unit, onError: (Exception) -> Unit) {
    firestore.collection("pets")
        .add(pet)
        .addOnSuccessListener { onSuccess() }
        .addOnFailureListener { e -> onError(e) }
}
```

- **firestore.collection("pets")**: Selecciona la colección "pets" en Firestore.
- **add(pet)**: Agrega el objeto pet a la colección.
- **addOnSuccessListener**: Se ejecuta si la operación es exitosa, llamando a onSuccess.
- **addOnFailureListener**: Se ejecuta si ocurre un error, llamando a onError con la excepción.

Este método recupera la lista de mascotas almacenadas en Firestore.

Parámetros:

- **onSuccess: (List<Pet>) -> Unit**: Función de callback que se ejecuta si la operación es exitosa, recibiendo una lista de objetos Pet.
- **onError: (Exception) -> Unit**: Función de callback que se ejecuta si ocurre un error, recibiendo la excepción correspondiente.
- **firestore.collection("pets").get()**: Recupera todos los documentos de la colección "pets".
- **addOnSuccessListener**: Se ejecuta si la operación es exitosa, procesando el querySnapshot:

- **querySnapshot.documents.mapNotNull { it.toObject(Pet::class.java) }:** Convierte cada documento de Firestore en un objeto Pet, ignorando los documentos que no se pueden convertir.
- **onSuccess(pets):** Llama a onSuccess pasando la lista de mascotas obtenida.
- **addOnFailureListener:** Se ejecuta si ocurre un error, llamando a onError con la excepción.

```
fun getPets(onSuccess: (List<Pet>) -> Unit, onError: (Exception) -> Unit) {
    firestore.collection(collectionPath: "pets")
        .get()
        .addOnSuccessListener { querySnapshot ->
            val pets = querySnapshot.documents.mapNotNull { it.toObject(Pet::class.java) }
            onSuccess(pets)
        }
        .addOnFailureListener { e -> onError(e) }
}
```

REGISTERPETSCREEN

Variables de estado

- **mContext:** Obtiene el contexto actual de la aplicación, que se utiliza para mostrar mensajes emergentes.
- **videoUri:** Variable que almacena la URI del video seleccionado, inicializada como null.
- **errorMessage:** Variable que almacena un mensaje de error, inicializada como una cadena vacía

```
@Composable
fun RegisterPetScreen(navController: NavHostController) {
    val mContext = LocalContext.current

    // Estados para los campos de texto
    var name by remember { mutableStateOf("") }
    var breed by remember { mutableStateOf("") }
    var age by remember { mutableStateOf("") }
    var health by remember { mutableStateOf("") }
    var carnet by remember { mutableStateOf("") }

    // Estado para las URI de video seleccionados
    var videoUri by remember { mutableStateOf<Uri?> (value: null) }

    // Mensaje de error
    var errorMessage by remember { mutableStateOf("") }

    // Variables de estado para mostrar etiquetas
    var showNameLabel by remember { mutableStateOf( value: true) }
    var showBreedLabel by remember { mutableStateOf( value: true) }
    var showAgeLabel by remember { mutableStateOf( value: true) }
    var showHealthLabel by remember { mutableStateOf( value: true) }
    var showCarnetLabel by remember { mutableStateOf( value: true) }
```

Launcher para seleccionar video

Este launcher permite seleccionar un video desde la galería. Al seleccionar un video, la URI del video se almacena en la variable videoUri.

```
// Launcher para seleccionar video desde la galería
val videoLauncher = rememberLauncherForActivityResult(ActivityResultContracts.GetContent()) { uri: Uri? ->
    videoUri = uri
}
```

Composición de la UI

Crea una columna que ocupa todo el tamaño disponible y tiene un espaciado vertical de 16 dp. Los elementos dentro de la columna están alineados horizontalmente al centro.

Campos de entrada de texto

Para cada campo de entrada de texto (nombre, raza, edad, salud y carnet de vacunación):

Cada TextField permite al usuario ingresar información sobre la mascota. Se controla la visibilidad de la etiqueta de cada campo, así como el tamaño y el estilo del texto. El valor del campo se actualiza a medida que el usuario escribe.

```
// Titulo de la pantalla
Text(
    text = "Registro de Mascotas",
    fontSize = 24.sp,
    style = MaterialTheme.typography.headlineMedium
)

// Campos de entrada de texto
TextField(
    value = name,
    onValueChange = { it: String ->
        name = it
        showNameLabel = name.isNotBlank() // Mantener etiqueta visible si el campo está vacío
    },
    label = { if (showNameLabel) Text(text = "Nombre") },
    textStyle = MaterialTheme.typography.bodyMedium.copy(fontSize = 16.sp),
    modifier = Modifier
        .fillMaxWidth()
        .padding(vertical = 4.dp)
        .onFocusChanged { focusState ->
            showNameLabel = if (focusState.isFocused) false else name.isNotBlank()
        }
)
```

Mostrar URI de video y mensaje de error: Si hay un video seleccionado, se muestra su URI en un texto con un tamaño de fuente más pequeño.

```

9
10    // Mostrar video seleccionado con texto más pequeño
11    videoUri?.let { uri ->
12        Text(
13            text = "Video seleccionado: $uri",
14            fontSize = 12.sp // Tamaño de fuente más pequeño
15        )
16    }
17
18    // Mensaje de error con texto más pequeño
19    if (errorMessage.isNotEmpty()) {
20        Text(
21            text = errorMessage,
22            color = MaterialTheme.colorScheme.error,
23            fontSize = 12.sp // Tamaño de fuente más pequeño
24        )
25    }

```

Botones

Botón para seleccionar video

Al hacer clic, se lanza el videoLauncher.

Botones para limpiar campos

Este botón reinicia todas las variables de estado a sus valores iniciales, limpiando los campos de entrada.

Registrar Mascota: Este botón valida los campos de entrada. Si todos los campos están completos y un video ha sido seleccionado, se crea un objeto Pet con la información ingresada. En caso de éxito, se puede llamar a un método para guardar la mascota en Firestore y se muestra un mensaje de éxito al usuario.

Este diseño provee una experiencia de usuario fluida, brindando información relevante sobre las mascotas en un formato claro y visualmente atractivo.

```

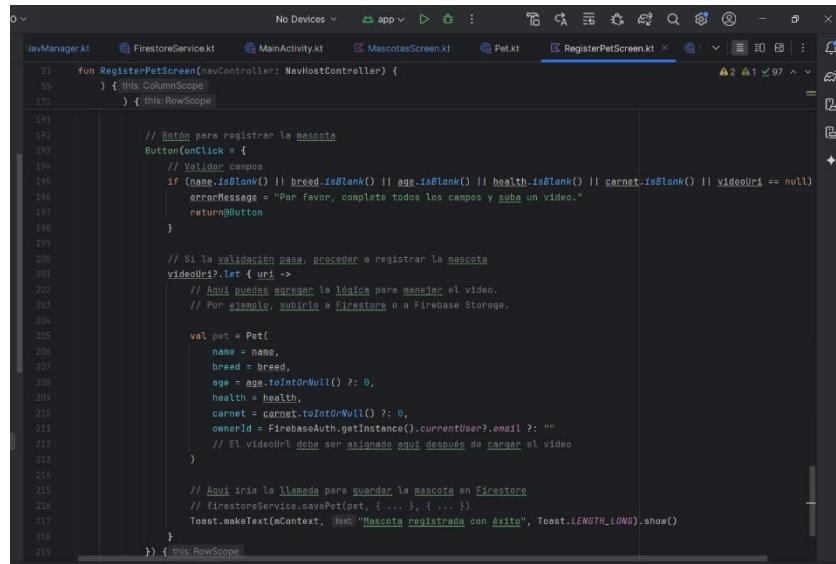
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195

```

Además, facilita la navegación y el manejo de errores de manera transparente.

Características Principales

- **Validación de Campos:** Antes de registrar la mascota, se verifica que todos los campos estén completos. Si falta información, se muestra un mensaje de error al usuario.
- **Interfaz Amigable:** Utiliza Jetpack Compose para proporcionar una interfaz de usuario intuitiva y responsive, con etiquetas que se mantienen visibles hasta que el usuario comienza a escribir en el campo correspondiente.
- **Gestión de Estado:** Se utilizan variables de estado para controlar los valores de entrada, el mensaje de error, y la selección de video, asegurando que la interfaz se actualice dinámicamente conforme el usuario interactúa.
- **Registro de Mascota:** Al completar todos los campos, se crea un objeto Pet con la información ingresada y se puede llamar a un método para guardar esta información en Firestore, registrando la mascota en la base de datos de la aplicación.
- **Mensajes de Éxito:** Se proporcionan mensajes emergentes (Toast) para notificar al usuario sobre el éxito del registro de la mascota.



```
0 ~ NavManager.kt FireStoreService.kt MainActivity.kt MascotasScreen.kt Pet.kt RegisterPetScreen.kt 1 21 fun RegisterPetScreen(navController: NavController) { 22     // Botón para registrar la mascota 23     Button(onClick = { 24         // Validar campos 25         if (name.isBlank() || breed.isBlank() || age.isBlank() || health.isBlank() || carnet.isBlank() || videoUri == null) 26             errorMessage = "Por favor, complete todos los campos y sube un video." 27         return@Button 28     }) 29     // Si la validación pasa, proceder a registrar la mascota 30     videoUri?.let { uri -> 31         // Aquí puedes agregar la lógica para manejar el video. 32         // Por ejemplo, subirlo a Firestore o a Firebase Storage. 33         val pet = Pet( 34             name = name, 35             breed = breed, 36             age = age.toIntOrNull() ?: 0, 37             health = health, 38             carnet = carnet.toIntOrNull() ?: 0, 39             ownerId = FirebaseAuth.getInstance().currentUser?.email ?: "", 40             // El videoUri debe ser asignado aquí después de cargar el video 41         ) 42         // Aquí iría la llamada para guardar la mascota en Firestore 43         // fireStoreService.savePet(pet, { ... }, { ... }) 44         Toast.makeText(mContext, "Mascota registrada con éxito", Toast.LENGTH_LONG).show() 45     } 46 } 47 } { this RowScope
```

Flujo de Trabajo

1. **Ingreso de Datos:** El usuario ingresa la información requerida en los campos correspondientes.

2. **Selección de Video:** Se puede subir un video opcional que esté relacionado con la mascota.
3. **Validación:** Al hacer clic en el botón de "Registrar Mascota", se valida que todos los campos estén completos y que se haya seleccionado un video.
4. **Registro:** Si la validación es exitosa, se registra la mascota y se notifica al usuario del éxito.

Este proceso de registro está diseñado para ser sencillo y eficiente, mejorando la experiencia del usuario en la aplicación "Paw Friends".

3. Reporte de mascotas perdidas y encontradas:

- Los usuarios podrán reportar una mascota perdida, añadiendo detalles sobre el lugar, fecha y una descripción detallada de la mascota.
- Opción para reportar mascotas encontradas, permitiendo que las personas busquen a los dueños a través de la aplicación.

4. Adopción de mascotas:

- Los usuarios podrán ofrecer mascotas para adopción, proporcionando toda la información relevante, incluyendo fotos, historial médico, y una breve descripción.
- Los interesados podrán enviar solicitudes de adopción, y los dueños podrán aceptar o rechazar dichas solicitudes.

Aspecto Final del Apartado “. REGISTRO Y GESTIÓN DE MASCOTAS”

No hay SIM 4G K/S 3:36

Mascotas Registradas

Nombre: Max
Raza: Pastor alemán
Edad: 3
Salud: Buena
Carnet: 83929299



Registrado por:
cchamom2@miumg.edu.gt

Nombre: Pepe
Raza: Gato doméstico
Edad: 0
Salud: Buena
Carnet: 73890139



Registrado por:
cchamom2@miumg.edu.gt

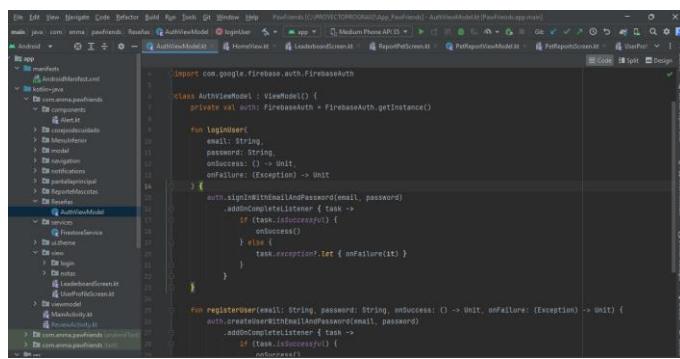
Nombre: Green

Raza:

5. SISTEMA DE CALIFICACIONES Y RESEÑAS:

- Los dueños de mascotas y adoptantes podrán calificarse mutuamente, dejando una puntuación de 1 a 5 estrellas y una reseña detallada sobre la experiencia.
- Se permitirá dejar **agradecimientos** por la adopción, cuidado temporal u otros servicios, fomentando un ambiente de confianza y colaboración dentro de la comunidad.

AuthViewModel.kt



```
import com.google.firebase.auth.FirebaseAuth

class AuthViewModel : ViewModel() {
    private val auth: FirebaseAuth = FirebaseAuth.getInstance()

    fun loginUser(
        email: String,
        password: String,
        onSuccess: () -> Unit,
        onFailure: (Exception) -> Unit
    ) {
        auth.signInWithEmailAndPassword(email, password)
            .addOnCompleteListener { task ->
                if (task.isSuccessful) {
                    onSuccess()
                } else {
                    task.exception?.let { onFailure(it) }
                }
            }
    }

    fun registerUser(email: String, password: String, onSuccess: () -> Unit, onFailure: (Exception) -> Unit) {
        auth.createUserWithEmailAndPassword(email, password)
            .addOnCompleteListener { task ->
                if (task.isSuccessful) {
                    onSuccess()
                } else {
                    task.exception?.let { onFailure(it) }
                }
            }
    }
}
```

Paquete: com.enma.pawfriends.Reseñas

Documentación Técnica:

Descripción General:

Este archivo define una clase `AuthViewModel` que extiende `ViewModel` y proporciona métodos para gestionar la autenticación de usuarios utilizando Firebase Authentication. Ofrece funciones para el inicio de sesión (`loginUser`) y el registro (`registerUser`) de usuarios. Cada operación tiene callbacks para manejar el éxito o el fallo del proceso.

Dependencias:

- Firebase Authentication: Utilizado para gestionar el inicio de sesión y registro de usuarios.
- Android Lifecycle ViewModel: Utilizado para proporcionar un ciclo de vida seguro y desacoplar la lógica de autenticación de la interfaz de usuario.

Componentes:

- `FirebaseAuth`: Proporciona acceso a las funciones de autenticación de Firebase.

- `ViewModel`: Proporciona una manera eficiente de manejar los datos de la UI y garantizar que sobrevivan a los cambios de configuración, como la rotación de la pantalla.

Funcionalidad:

- 1. AuthViewModel:** Esta clase extiende `ViewModel` y contiene lógica para autenticar a los usuarios a través de Firebase.

2. Variables:

- auth: instancia privada de `FirebaseAuth` que se utiliza para manejar las operaciones de autenticación, como iniciar sesión y registrar usuarios.

3. Métodos Públicos:

-loginUser(email: String, password: String, onSuccess: Unit, onFailure: (Exception) Unit):

- Este método se utiliza para iniciar sesión con las credenciales proporcionadas (email y contraseña).

- Parámetros:

- email: Correo electrónico del usuario.
- password: Contraseña del usuario.
- onSuccess: Callback a ejecutar si el inicio de sesión es exitoso.
- onFailure: Callback a ejecutar si el inicio de sesión falla, pasando la excepción como argumento.

- Proceso:

- Utiliza `signInWithEmailAndPassword()` para intentar iniciar sesión con Firebase.

- Si la tarea es exitosa (`task.isSuccessful`), ejecuta la función `onSuccess`.

- Si la tarea falla, se llama a `onFailure` pasando la excepción capturada.

- registerUser(email: String, password: String, onSuccess: () -> Unit, onFailure: (Exception) -> Unit):

- Este método se utiliza para registrar un nuevo usuario con el correo electrónico y contraseña proporcionados.

- Parámetros:

- email: Correo electrónico del nuevo usuario.

- password: Contraseña del nuevo usuario.
- onSuccess: Callback a ejecutar si el registro es exitoso.
- onFailure: Callback a ejecutar si el registro falla, pasando la excepción como argumento.

- Proceso:

- Utiliza `createUserWithEmailAndPassword()` para registrar un nuevo usuario con Firebase.
 - Si la tarea es exitosa (`task.isSuccessful`), se ejecuta la función `onSuccess`.
 - Si la tarea falla, se llama a `onFailure` pasando la excepción capturada.

Manejo de Errores:

- En ambos métodos (`loginUser` y `registerUser`), si la tarea de autenticación falla, la excepción se maneja llamando al callback `onFailure`.
- Las excepciones proporcionan detalles sobre por qué la autenticación falló, lo cual es útil para mejorar la experiencia del usuario (por ejemplo, mostrando mensajes específicos de error).

Uso:

- Esta clase se utiliza como un `ViewModel` para manejar la lógica de autenticación de manera segura y desacoplada de la UI.
- `AuthViewModel` se puede asociar a una actividad o fragmento para realizar el inicio de sesión o el registro de usuarios.
- Los métodos `loginUser` y `registerUser` permiten el uso de callbacks para manejar el éxito o el error de forma personalizada.

Notas:

- Los callbacks (`onSuccess` y `onFailure`) proporcionan flexibilidad al desarrollador para definir las acciones a tomar después de una operación de autenticación, como navegación o mostrar mensajes de error.
- Utilizar `ViewModel` garantiza que la lógica de autenticación sobreviva a cambios de configuración, como la rotación de la pantalla, mejorando la experiencia del usuario.
- Asegúrese de que `FirebaseAuth` esté correctamente configurado en el proyecto antes de utilizar `AuthViewModel` para evitar errores de configuración.

ReviewActivity.kt

The screenshot shows the Android Studio interface with the code editor open to the file `ReviewAndRatingScreen.kt`. The code defines a function `ReviewAndRatingScreen` that takes a `FirebaseFirestore` reference as a parameter. It uses a `Row` widget to display a rating scale from 1 to 5 stars. For each star, it creates an `IconButton` with a yellow star icon if the rating is less than or equal to the current star, otherwise a gray star icon. The `Icon` has a `contentDescription` of "Estrella \$star". Below the rating scale is a `Textfield` for writing a review. The code also includes imports for `Icons`, `Modifier`, and `Alignment`.

```
fun ReviewAndRatingScreen(db: FirebaseFirestore) {  
    Row(  
        modifier = Modifier.padding(8.dp),  
        verticalAlignment = Alignment.CenterVertically  
    ) {  
        (1 .. 5).forEach { star ->  
            IconButton(  
                onClick = { rating = star }  
            ) {  
                Icon(  
                    imageVector = Icons.Filled.Star,  
                    contentDescription = "Estrella $star",  
                    tint = if (star <= rating) Color.Yellow else Color.Gray,  
                    modifier = Modifier.size(40.dp)  
                )  
            }  
        }  
        Spacer(modifier = Modifier.height(16.dp))  
        // Campo de texto para la reseña  
        Textfield(  
            value = reviewText  
        )  
    }  
}
```

Paquete: com.enma.pawfriends

Documentación técnica:

Descripción General: Este archivo define una función componible, `ReviewAndRatingScreen` que se utiliza para permitir que los usuarios califiquen su experiencia con otro usuario. Los usuarios pueden dejar una calificación de 1 a 5 estrellas, escribir una reseña y un mensaje de agradecimiento. La información de la reseña se guarda en Firestore y se muestra un mensaje de confirmación una vez que se ha enviado.

Dependencias:

- Firebase Firestore: Utilizado para guardar la reseña en la base de datos.
- Jetpack Compose: Utilizado para construir la interfaz de usuario.
- Corrutinas de Kotlin: Utilizadas para manejar el estado de la interfaz de usuario.

Componentes:

- `FirebaseFirestore`: Interactúa con la base de datos Firestore para guardar la reseña del usuario.
- `rememberSaveable`: Utilizado para recordar el estado de las variables incluso si la composición se recompone.

Funcionalidad:

1. **Pantalla de revisión y calificación (base de datos: `FirebaseFirestore`) :**

- Esta es la función componible principal definida en este archivo. Toma un parámetro db de tipo FirebaseFirestore para guardar la reseña del usuario.
- Permite al usuario calificar, escribir una reseña detallada y un mensaje de agradecimiento.
- **Variables:**
 - rating: Estado mutable para almacenar la calificación, inicialmente establecida en 0.
 - reviewText: Estado mutable para almacenar el texto de la reseña, inicialmente vacío.
 - appreciationText: Estado mutable para almacenar el texto de agradecimiento, inicialmente vacío.
 - reviewSubmitted: Estado mutable para verificar si la reseña ya ha sido enviada.

2. Interfaz de usuario:

- **Título:**
 - Text: Muestra el título “Califica tu experiencia” estilizado con un tamaño de fuente de 20.sp.
- **Estrellas de calificación:**
 - Se muestran 5 íconos de estrellas, los cuales los usuarios pueden seleccionar para calificar su experiencia.
 - Las estrellas se llenan de amarillo según la calificación seleccionada por el usuario.
- **Campos de texto:**
 - **reviewText** : Campo de texto donde el usuario puede escribir una reseña detallada.
 - **apreciaciónTexto** : Campo de texto donde el usuario puede dejar un agradecimiento.
- **Botón de Enviar Reseña:**
 - Permite al usuario enviar la reseña y guardar la información en Firestore.

- La reseña se guarda en la colección “reseñas” en Firestore.
 - Si la reseña se guarda con éxito, se cambia el estado reviewSubmitted para mostrar el mensaje de agradecimiento.
- **Mensaje de confirmación:**
 - Si la reseña ya ha sido enviada, se muestra un mensaje de agradecimiento junto con la calificación, la reseña y el agradecimiento proporcionados por el usuario.

3. Manejo de Datos en Firestore :

- **Agregar Reseña:**

- La reseña se guarda en la colección “reseñas” con los campos rating, reviewText, y appreciationText.
- En caso de éxito, se cambia el estado reviewSubmitted para mostrar la confirmación.
- En caso de error, se imprime el mensaje de error en la consola.

4. Diseño:

- La pantalla está compuesta por un Column que organiza los elementos verticalmente.
- El Column tiene modificadores que aseguran que la pantalla se llene completamente (fillMaxSize()), con un padding de 16.dp para el margen.
- Alineación horizontal centrada para todos los elementos.

Manejo de errores:

- Si ocurre un error durante el proceso de guardar la reseña en Firestore, el error se imprime en la consola.
- Se puede agregar una lógica de manejo de errores más avanzada según sea necesario, por ejemplo, mostrar un mensaje de error en la interfaz de usuario.

Uso:

- Esta función componible debe ser utilizada en una pantalla que permita a los usuarios calificar su experiencia con otros usuarios.
- El parámetro dbes necesario para interactuar con la base de datos de Firestore y guardar la reseña.

Notas:

- Las consultas de Firestore se ejecutarán de forma asíncrona y la interfaz de usuario se actualizará automáticamente una vez que la reseña sea enviada con éxito.
- La calificación, reseña y agradecimiento proporcionados por el usuario se guardan en la base de datos para futuras referencias.

Aspecto Final del apartado “SISTEMA DE CALIFICACIONES Y RESEÑAS”



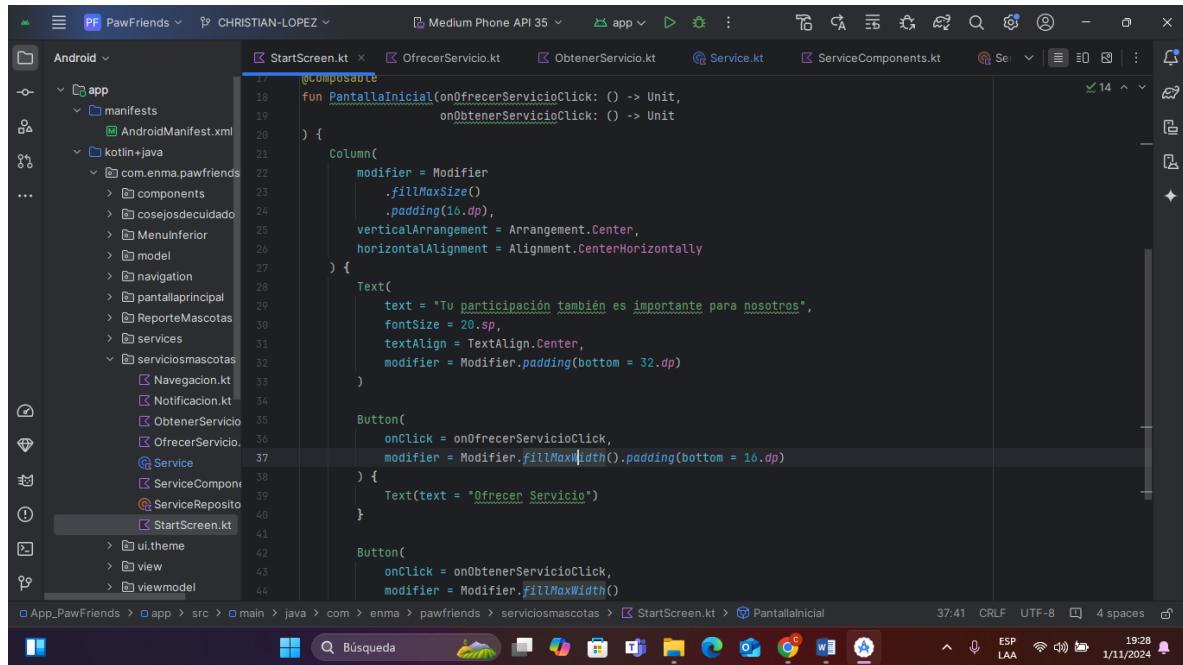
6. SERVICIOS DE MASCOTAS

1. PantallaInicial

Esta es la pantalla principal donde el usuario puede elegir entre:

Ofrecer un servicio: lleva a OfferServiceScreen.

Obtener un servicio: lleva a GetServiceScreen.



```
17 @Composable
18 fun PantallaInicial(onOfferServiceClick: () -> Unit,
19                     onGetServiceClick: () -> Unit
20 ) {
21     Column(
22         modifier = Modifier
23             .fillMaxSize()
24             .padding(10.dp),
25         verticalArrangement = Arrangement.Center,
26         horizontalAlignment = Alignment.CenterHorizontally
27     ) {
28         Text(
29             text = "Tu participación también es importante para nosotros",
30             fontSize = 20.sp,
31             textAlign = TextAlign.Center,
32             modifier = Modifier.padding(bottom = 32.dp)
33         )
34
35         Button(
36             onClick = onOfferServiceClick,
37             modifier = Modifier.fillMaxWidth().padding(bottom = 16.dp)
38         ) {
39             Text(text = "Ofrecer Servicio")
40         }
41
42         Button(
43             onClick = onGetServiceClick,
44             modifier = Modifier.fillMaxWidth()
45         )
46     }
47 }
```

2. OfferServiceScreen

Aquí, el usuario introduce los detalles de un servicio que desea ofrecer:

Campos de entrada para descripción, tipo de servicio, y, si el servicio es pago, precio.

Casilla de verificación para indicar si es gratuito.

Botón de "Servicio Público" que llama a ServiceRepository.addService para almacenar los datos en Firebase. Esto implica:

Crear una instancia de Service con los datos ingresados.

Enviar la instancia a la colección en Firebase usando la función addService.

```
fun OfferServiceScreen(serviceAdded: () -> Unit) {
    var description by remember { mutableStateOf("") }
    var type by remember { mutableStateOf("") }
    var price by remember { mutableStateOf("") }
    var isFree by remember { mutableStateOf(false) }
    val scope = rememberCoroutineScope()

    Column(
        modifier = Modifier.fillMaxSize().padding(16.dp),
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Text(text = "Describe el servicio")
        Spacer(modifier = Modifier.height(8.dp))
        BasicTextField(
            value = description,
            onValueChange = { description = it },
            modifier = Modifier.fillMaxWidth(),
            decorationBox = { innerTextField ->
                if (description.isEmpty()) Text(text = "Descripción")
                innerTextField
            }
        )
        Spacer(modifier = Modifier.height(8.dp))
        BasicTextField(
            value = type,
            onValueChange = { type = it },
            modifier = Modifier.fillMaxWidth(),
            decorationBox = { innerTextField ->
                if (type.isEmpty()) Text(text = "Tipo")
                innerTextField
            }
        )
    }
}
```

3.GetServiceScreen

Muestra una lista de los servicios disponibles. Funciona de la siguiente forma:

Recupera servicios llamando a ServiceRepository.getServices.

Despliega servicios en una lista usando LazyColumn. Cada servicio se muestra como un ServiceItem con los detalles, y un botón para solicitar el servicio.

```
@Composable
fun GetServiceScreen(serviceRequested: (Service) -> Unit) {
    val scope = rememberCoroutineScope()
    var services by remember { mutableStateOf(listOf<Service>()) }

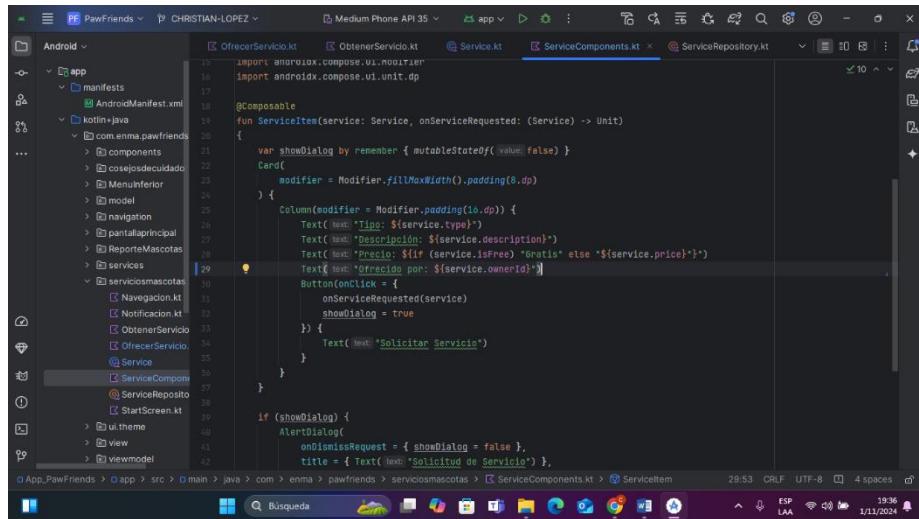
    LaunchedEffect(Unit) {
        scope.launch {
            services = ServiceRepository.getServices()
        }
    }

    LazyColumn(
        modifier = Modifier.fillMaxSize().padding(16.dp)
    ) {
        items(services.size) { index ->
            val service = services[index]
            ServiceItem(service, onServiceRequested)
        }
    }
}
```

4.ServiceItem

Cada elemento de servicio incluye:

Los detalles del servicio como tipo, descripción, precio y quién lo ofrece.
Un botón para solicitar el servicio que activa una AlertDialog de confirmación para mostrar que el servicio ha sido solicitado con éxito.



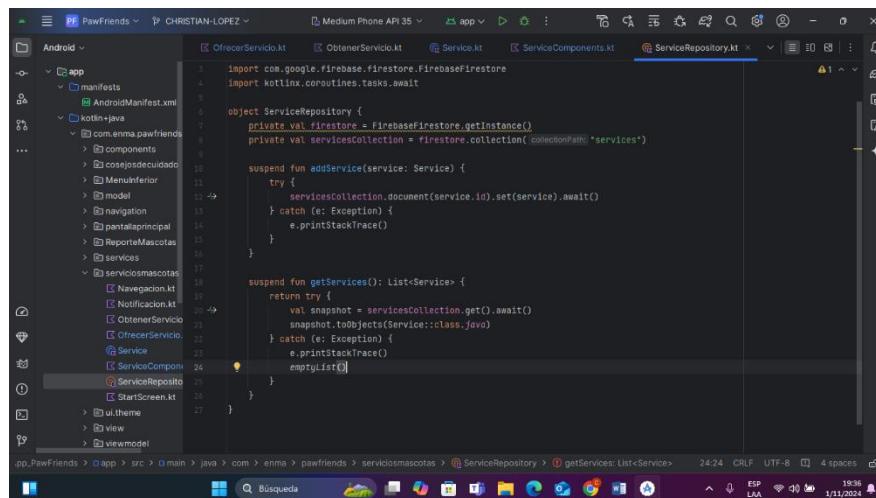
```
15 import androidx.compose.ui.Modifier
16 import androidx.compose.ui.layout.layoutId
17
18 @Composable
19 fun ServiceItems(service: Service, onServiceRequested: (Service) -> Unit)
20 {
21     var showDialog by remember { mutableStateOf(false) }
22     Card(
23         modifier = Modifier.fillMaxWidth().padding(8.dp)
24     ) {
25         Column(modifier = Modifier.padding(16.dp)) {
26             Text(text = "Tipo: ${service.type}")
27             Text(text = "Descripción: ${service.description}")
28             Text(text = "Precio: ${if (service.isFree) "Gratis" else "${service.price}"}")
29             Text(text = "Gratis por: ${service.ownerId}")
30             Button(onClick = {
31                 onServiceRequested(service)
32                 showDialog = true
33             }) {
34                 Text(text = "Solicitar Servicio")
35             }
36         }
37     }
38     if (showDialog) {
39         AlertDialog(
40             onDismissRequest = { showDialog = false },
41             title = { Text(text = "Solicitud de Servicio") },
42             text = { Text(text = "¿Deseas solicitar este servicio?") },
43             confirmButton = { TextButton(onClick = { showDialog = false }) { Text(text = "Sí") } },
44             dismissButton = { TextButton(onClick = { showDialog = false }) { Text(text = "No") } }
45         )
46     }
47 }
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
```

5.ServiceRepository

Este objeto actúa como intermediario entre la aplicación y Firebase:

addService: Agrega un servicio a la colección en Firebase.

getServices: Recupera todos los servicios de Firebase y los convierte en una lista de objetos Service.

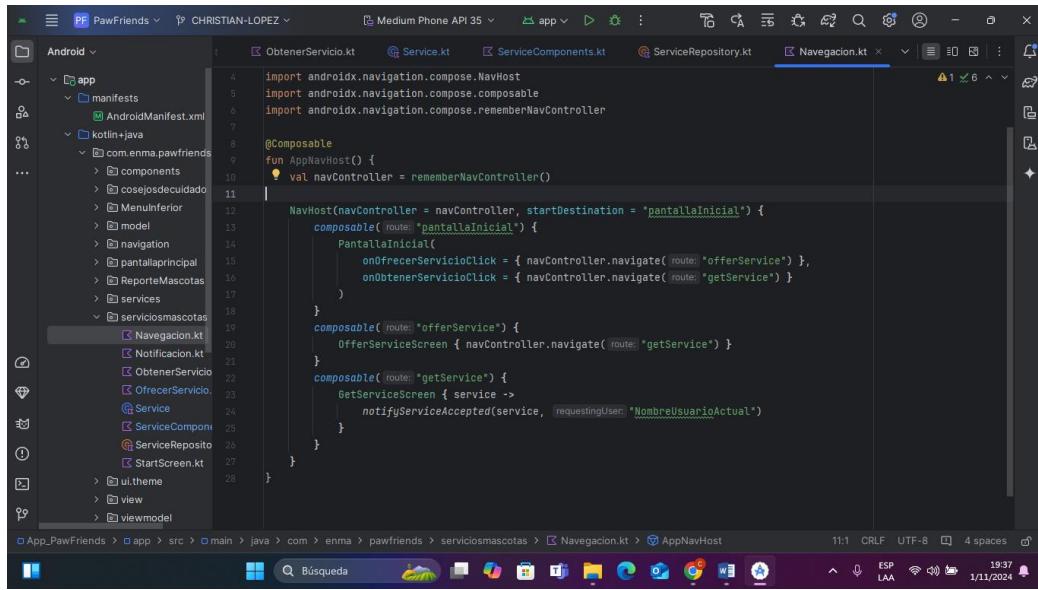


```
1 import com.google.firebase.firestore.FirebaseFirestore
2 import kotlinx.coroutines.tasks.await
3
4 object ServiceRepository {
5     private val firestore = FirebaseFirestore.getInstance()
6     private val servicesCollection = firestore.collection(collectionPath: "services")
7
8     suspend fun addService(service: Service) {
9         try {
10             servicesCollection.document(service.id).set(service).await()
11         } catch (e: Exception) {
12             e.printStackTrace()
13         }
14     }
15
16     suspend fun getServices(): List<Service> {
17         return try {
18             val snapshot = servicesCollection.get().await()
19             snapshot.toObjects(Service::class.java)
20         } catch (e: Exception) {
21             e.printStackTrace()
22             emptyList()
23         }
24     }
25 }
```

6.AppNavHost

Defina la navegación entre pantallas:

Usa NavHosty composable para especificar las rutas entre las pantallas pantallaInicial, offerService, y getService.



```
import androidx.navigation.compose.NavHost
import androidx.navigation.compose.composable
import androidx.navigation.compose.rememberNavController

@Composable
fun AppNavHost() {
    val navController = rememberNavController()

    NavHost(navController = navController, startDestination = "pantallaInicial") {
        composable(route: "pantallaInicial") {
            PantallaInicial(
                onOffererServicioClick = { navController.navigate(route: "offerService") },
                onObtenerServicioClick = { navController.navigate(route: "getService") }
            )
        }
        composable(route: "offerService") {
            OfferServiceScreen { navController.navigate(route: "getService") }
        }
        composable(route: "getService") {
            GetServiceScreen { service ->
                notifyServiceAccepted(service, requestingUser: "NombreUsuarioActual")
            }
        }
    }
}
```

Aspecto Final del apartado “SERVICIOS DE MASCOTAS”

8:55 29% **Describe el servicio**

Descripción
Tipo de Servicio (Paseo, Aseo, etc.)
 ¿Servicio Gratuito?
Precio
Publicar Servicio

8:55 29% **Tipo: Aseo**
Descripción: Baño y aseo en general de mascotas
Precio: null
Ofrecido por:
Solicitar Servicio

8:55 28% **Tipo: Aseo**
Descripción: Limpieza y aseo de mascotas en general
Precio: 100.0
Ofrecido por:
Solicitar Servicio

8:55 28% **Tipo: paseo**
Descripción: manitas club
Precio: null
Ofrecido por:
Solicitar Servicio

Tu participación también es importante para nosotros

Ofrecer Servicio

Obtener Servicio

||| ○ <

||| ○ <

||| ○ <

9. CONSEJOS DE CUIDADO DE MASCOTAS

Sección dedicada a **recomendaciones de cuidado de mascotas**, con artículos, videos y guías que los usuarios pueden consultar.

Listado de clínicas veterinarias cercanas con datos de contacto y la posibilidad de dejar reseñas sobre los servicios veterinarios.

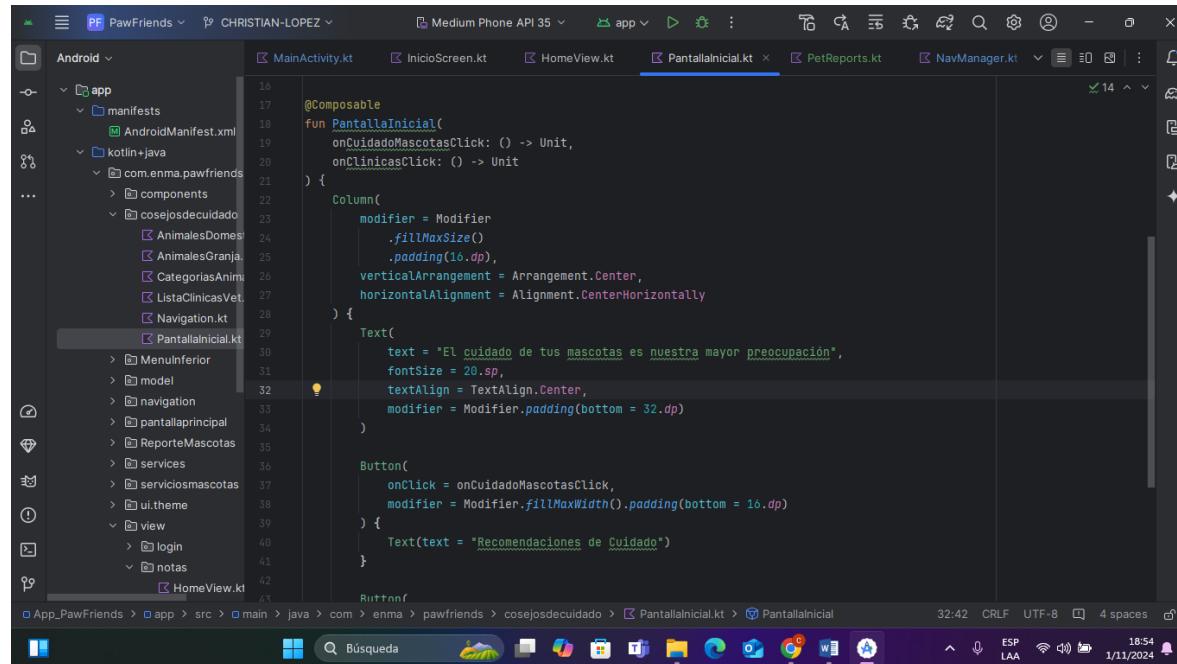
1. Pantalla Inicial (PantallaInicial)

Paquete: com.enma.pawfriends

Descripción: Pantalla principal de la aplicación, que ofrece al usuario dos opciones: ver de cuidado de mascotas o ver la lista de clínicas veterinarias cercanas.

Componentes:

- ✓ **Button:** Para navegar a las secciones de recomendaciones de cuidado y clínicas veterinarias.
- ✓ **Parámetros:**
- ✓ **onCuidadoMascotasClick:** Devolución de llamada para navegar a las recomendaciones de cuidado.
- ✓ **onClinicasClick:** Devolución de llamada para navegar a la lista de clínicas veterinarias



```
16 @Composable
17 fun PantallaInicial(
18     onCuidadoMascotasClick: () -> Unit,
19     onClinicasClick: () -> Unit
20 ) {
21     Column(
22         modifier = Modifier
23             .fillMaxSize()
24             .padding(10.dp),
25         verticalArrangement = Arrangement.Center,
26         horizontalAlignment = Alignment.CenterHorizontally
27     ) {
28         Text(
29             text = "El cuidado de tus mascotas es nuestra mayor preocupación.",
30             fontSize = 20.sp,
31             textAlign = TextAlign.Center,
32             modifier = Modifier.padding(bottom = 32.dp)
33         )
34
35         Button(
36             onClick = onCuidadoMascotasClick,
37             modifier = Modifier.fillMaxWidth().padding(bottom = 16.dp)
38         ) {
39             Text(text = "Recomendaciones de Cuidado")
40         }
41     }
42 }
```

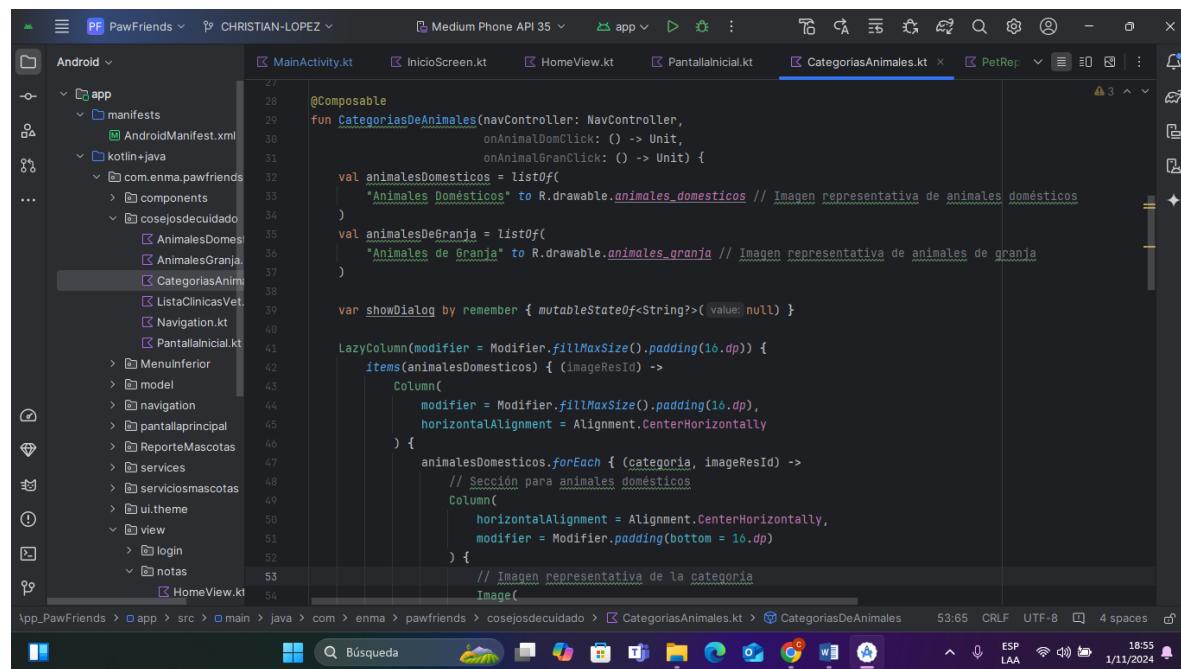
2. Pantalla de Categorías de Animales (CategoriasDeAnimales)

Paquete: com.enma.pawfriends.cosejosdecuidado

Descripción: Pantalla de selección de categoría, donde el usuario puede elegir entre "Animales Domésticos" y "Animales de Granja". Cada categoría se muestra con una imagen y dos botones: uno para ver recomendaciones generales (en un AlertDialog) y otro para navegar hacia la lista de animales de esa categoría.

Componentes:

- ✓ **LazyColumn:** Despliega las categorías de animales.
- ✓ **AlertDialog:** Muestra recomendaciones generales para animales domésticos y de granja.
- ✓ **Parámetros:**
- ✓ **NavController navController:** Para gestionar la navegación.
- ✓ **onAnimalDomClick:** Devolución de llamada para navegar hacia la pantalla de animales domésticos.
- ✓ **onAnimalGranClick:** Devolución de llamada para navegar hacia la pantalla de animales de granja.



The screenshot shows the Android Studio interface with the code editor open to the `CategoriasAnimales.kt` file. The code defines a Composable function `CategoriasDeAnimales` that takes a `NavController` and two click handlers. It initializes lists of animal categories and their corresponding drawables. A `showDialog` variable is used to manage a dialog. The main part of the code is a `LazyColumn` that iterates over the domestic animals list, creating a `Column` for each item. This inner `Column` contains an `Image` and another `Column` with horizontal alignment centered horizontally and bottom padding. The code is annotated with comments explaining the logic for displaying categories and their images.

```
27
28 @Composable
29 fun CategoriasDeAnimales(navController: NavController,
30     onAnimalDomClick: () -> Unit,
31     onAnimalGranClick: () -> Unit) {
32     val animalesDomesticos = listOf(
33         "Animales Domésticos" to R.drawable.animales_domesticos // Imagen representativa de animales domésticos
34     )
35     val animalesDeGranja = listOf(
36         "Animales de Granja" to R.drawable.animales_granja // Imagen representativa de animales de granja
37     )
38
39     var showDialog by remember { mutableStateOf<String?>() }
40
41     LazyColumn(modifier = Modifier.fillMaxSize().padding(16.dp)) {
42         items(animalesDomesticos) { (imageResourceId) ->
43             Column(
44                 modifier = Modifier.fillMaxSize().padding(16.dp),
45                 horizontalAlignment = Alignment.CenterHorizontally
46             ) {
47                 animalesDomesticos.forEach { (categoria, imageResourceId) ->
48                     // Sección para animales domésticos
49                     Column(
50                         horizontalAlignment = Alignment.CenterHorizontally,
51                         modifier = Modifier.padding(bottom = 16.dp)
52                     ) {
53                         // Imagen representativa de la categoría
54                         Image(
```

```
fun CategoriasDeAnimales(navController: NavController, modifier = Modifier.fillMaxSize().padding(16.dp)) {
    LazyColumn(modifier = modifier) {
        items(animalesDomesticos) { (imageResId) ->
            // Imagen representativa de la categoría
            Image(
                painter = painterResource(id = imageResId),
                contentDescription = "Imagen de $categoria",
                contentScale = ContentScale.Crop,
                modifier = Modifier
                    .size(250.dp)
                    .padding(bottom = 8.dp)
            )
            // Nombre de la categoría
            Text(text = categoria, fontSize = 22.sp, modifier = Modifier.padding(8.dp))
            // Botón para mostrar las recomendaciones generales
            Button(
                onClick = {
                    showDialog = "domesticos"
                }, // Mostrar el AlertDialog para animales domésticos
                modifier = Modifier.fillMaxWidth().padding(bottom = 16.dp)
            ) {
                Text(text = "Recomendaciones Generales")
            }
            // Botón para navegar a la pantalla de animales domésticos
            Button(
                onClick = { navController.navigate(route: "animales_domesticos") }, // Navega a la pantalla con
            )
        }
    }
}
```

```
        }
        animalesDeGranja.forEach { (categoria, imageResId) ->
            // Sección para animales de granja
            Column(
                horizontalAlignment = Alignment.CenterHorizontally,
                modifier = Modifier.padding(bottom = 16.dp)
            ) {
                // Imagen representativa de la categoría
                Image(
                    painter = painterResource(id = imageResId),
                    contentDescription = "Imagen de $categoria",
                    contentScale = ContentScale.Crop,
                    modifier = Modifier
                        .size(250.dp)
                        .padding(bottom = 8.dp)
                )
                // Nombre de la categoría
                Text(text = categoria, fontSize = 22.sp, modifier = Modifier.padding(8.dp))
                // Botón para mostrar las recomendaciones generales
                Button(
                    onClick = {
                        showDialog = "granja"
                    }
                )
            }
        }
    }
}
```

3. Pantalla de Animales Domésticos (AnimalesDomesticos)

Paquete: com.enma.pawfriends.cosejosdecuidado

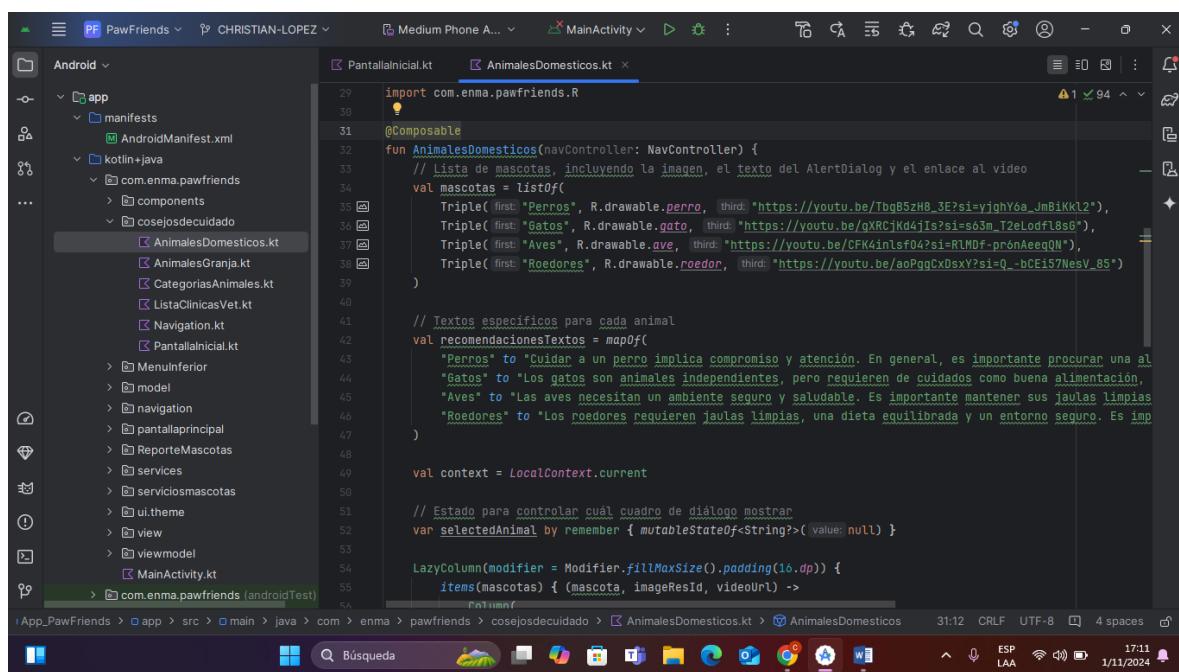
Descripción: Pantalla que despliega una lista de animales domésticos (perros, gatos, aves y roedores). Cada elemento de la lista incluye una imagen representativa del animal, un botón para mostrar de recomendaciones de cuidado en un AlertDialog y otro botón para ver un video de cuidados específicos del animal en YouTube.

Componentes:

- ✓ **LazyColumn:** Para desplegar los elementos de manera vertical.
- ✓ **Image:** Muestra la imagen del animal.
- ✓ **AlertDialog:** Despliega recomendaciones generales de cuidado para cada animal.
- ✓ **Intent:** Lanza un enlace a YouTube con videos de cuidados específicos de cada animal.

Parámetros:

- ✓ **NavController navController:** Para gestionar la navegación entre pantallas.



The screenshot shows the Android Studio interface with the code editor open to the file `AnimalesDomesticos.kt`. The code defines a Composable function `AnimalesDomesticos` that takes a `NavController` parameter. It lists four types of pets: Perros, Gatos, Aves, and Roedores, each associated with a drawable resource and a YouTube video URL. It also contains specific text recommendations for each animal. The code uses `LazyColumn` to display the items.

```
import com.enma.pawfriends.R
@Composable
fun AnimalesDomesticos(navController: NavController) {
    // Lista de mascotas, incluyendo la imagen, el texto del AlertDialog y el enlace al video
    val mascotas = listOf(
        Triple(first: "Perros", R.drawable.perro, third: "https://youtu.be/TboB5zH8_3E?si=yjghY6a_JmBiKKL2"),
        Triple(first: "Gatos", R.drawable.gato, third: "https://youtu.be/gXRCJk04Jls?si=s63m_I2eLoftL86"),
        Triple(first: "Aves", R.drawable.ave, third: "https://youtu.be/CFK4inlsf04?si=RMDF-pr0AeqQN"),
        Triple(first: "Roedores", R.drawable.roedor, third: "https://youtu.be/aoPggCxDsX?si=Q_-bCEi57NesV_85")
    )

    // Textos específicos para cada animal
    val recomendacionesTextos = mapOf(
        "Perros" to "Cuidar a un perro implica compromiso y atención. En general, es importante procurar una alimentación adecuada y regular los paseos. Los perros necesitan ejercicio y socialización para su bienestar emocional.",
        "Gatos" to "Los gatos son animales independientes, pero requieren de cuidados como buena alimentación y un ambiente seguro. Es importante mantener sus jaulas limpias y seguras para evitar enfermedades.",
        "Aves" to "Las aves necesitan un ambiente seguro y saludable. Es importante mantener sus jaulas limpias y seguras para evitar enfermedades. Las aves necesitan volar y explorar libremente.",
        "Roedores" to "Los roedores requieren jaulas limpias, una dieta equilibrada y un entorno seguro. Es importante proporcionarles juguetes y estímulos mentales para su desarrollo."
    )

    val context = LocalContext.current

    // Estado para controlar cuál cuadro de diálogo mostrar
    var selectedAnimal by remember { mutableStateOf<String?>(value: null) }

    LazyColumn(modifier = Modifier.fillMaxSize().padding(16.dp)) {
        items(mascotas) { (mascota, imageResId, videoUrl) ->
            Column {
                Image(
                    modifier = Modifier.size(100.dp),
                    contentDescription = "Imagen de $mascota",
                    painter = painterResource(id = imageResId)
                )
                Row {
                    Text("Mascota: $mascota")
                    Text("Video: $videoUrl")
                }
                Text("Recomendaciones: ${recomendacionesTextos[mascota]}")
                if (selectedAnimal == null) {
                    Text("Haz clic para ver las recomendaciones")
                } else if (selectedAnimal == mascota) {
                    Text("Ver recomendaciones")
                } else {
                    Text("Ver video")
                }
            }
        }
    }
}
```

4. Pantalla de Animales de Granja (AnimalesGranja)

Paquete: com.enma.pawfriends.cosejosdecuidado

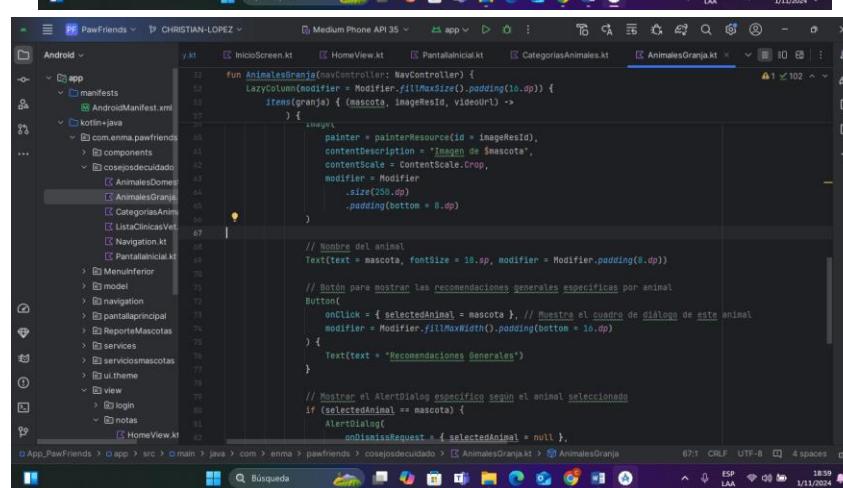
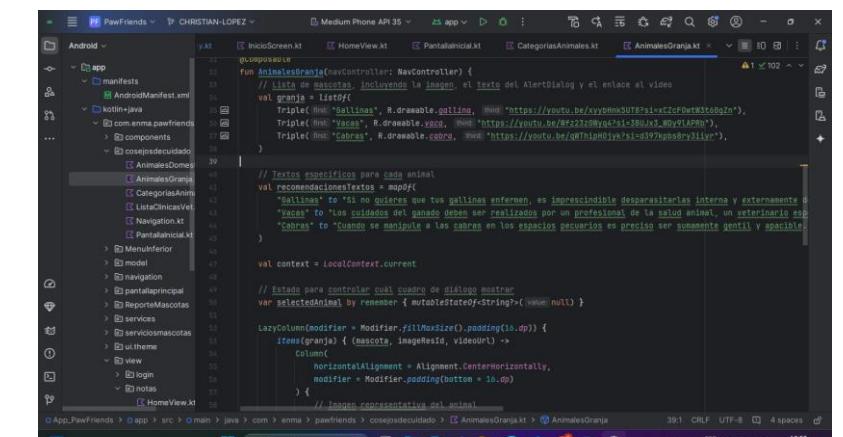
Descripción: Pantalla que presenta una lista de animales de granja (gallinas, vacas, cabras). De manera similar AnimalesDomesticos, esta pantalla incluye una imagen representativa, recomendaciones generales de cuidado en un animal AlertDialog y un botón para ver un video de YouTube sobre cuidados específicos para cada animal.

Componentes:

- ✓ **LazyColumn, Image, AlertDialog, Intent:** Funcionan igual que en AnimalesDomesticos.

Parámetros:

- ✓ **NavController navController:** Para gestionar la navegación entre pantallas.



```
fun AnimalesGranja(navController: NavController) {
    LazyColumn(modifier = Modifier.fillMaxSize().padding(10.dp)) {
        items(granos) { mascota, resourceId, videoUrl ->
            Column {
                Image(
                    painter = painterResource(id = resourceId),
                    contentDescription = "Imagen de $mascota",
                    contentScale = ContentScale.Crop,
                    modifier = Modifier
                        .size(200.dp)
                        .padding(bottom = 8.dp)
                )
                // Nombre del animal
                Text(text = mascota, fontSize = 18.sp, modifier = Modifier.padding(8.dp))
                // Botón para mostrar las recomendaciones generales específicas por animal
                Button(onClick = { selectedAnimal = mascota }, // Muestra el cuadro de diálogo de este animal
                    modifier = Modifier.fillMaxWidth().padding(bottom = 10.dp)
                ) {
                    Text(text = "Recomendaciones Generales")
                }
                // Muestra el AlertDialog específico según el animal seleccionado
                if (selectedAnimal == mascota) {
                    AlertDialog(
                        onDismissRequest = { selectedAnimal = null },
                        title = "Recomendaciones para $mascota"
                    ) {
                        Text(text = "Aquí se detallan las recomendaciones generales para $mascota")
                    }
                }
            }
        }
    }
}
```

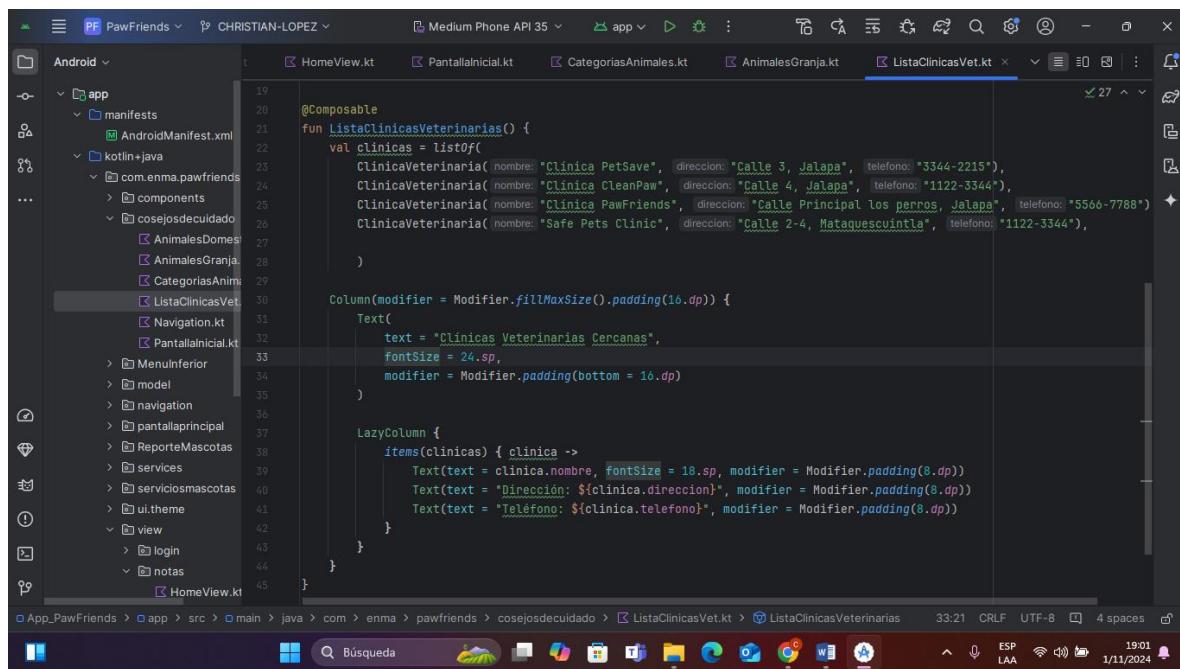
5. Pantalla de Clínicas Veterinarias (ListaClinicasVeterinarias)

Paquete: com.enma.pawfriends

Descripción: Pantalla que despliega una lista de clínicas veterinarias cercanas. Cada clínica se muestra con su nombre, dirección y teléfono de contacto.

Componentes:

- ✓ **LazyColumn:** Para implementar la lista de clínicas de manera vertical.
- ✓ **Datos:** Las clínicas veterinarias se modelan a través de la clase de datos ClinicaVeterinaria, que incluyen los atributos nombre, direccion, y telefono.



```
19 @Composable
20 fun ListaClinicasVeterinarias() {
21     val clinicas = listOf(
22         ClinicaVeterinaria(nombre: "Clinica PetSave", direccion: "Calle 3, Jalapa", telefono: "3344-2215"),
23         ClinicaVeterinaria(nombre: "Clinica CleanPaw", direccion: "Calle 4, Jalapa", telefono: "1122-3344"),
24         ClinicaVeterinaria(nombre: "Clinica PawFriends", direccion: "Calle Principal los perros, Jalapa", telefono: "5566-7788"),
25         ClinicaVeterinaria(nombre: "Safe Pets Clinic", direccion: "Calle 2-4, Mataquesquintla", telefono: "1122-3344"),
26     )
27
28     Column(modifier = Modifier.fillMaxSize().padding(16.dp)) {
29         Text(
30             text = "Clínicas Veterinarias Cercanas",
31             fontSize = 24.sp,
32             modifier = Modifier.padding(bottom = 16.dp)
33         )
34
35         LazyColumn {
36             items(clinicas) { clinica ->
37                 Text(text = clinica.nombre, fontSize = 18.sp, modifier = Modifier.padding(8.dp))
38                 Text(text = "Dirección: ${clinica.direccion}", modifier = Modifier.padding(8.dp))
39                 Text(text = "Teléfono: ${clinica.telefono}", modifier = Modifier.padding(8.dp))
40             }
41         }
42     }
43 }
44 }
```

6. Navegación de la Aplicación (AppNavigation)

Paquete: com.enma.pawfriends

Descripción: Gestiona la navegación entre las pantallas principales de la aplicación mediante NavHost.

Pantallas Registradas:

- ✓ **pantalla_inicial:** Pantalla inicial con opciones de navegación a recomendaciones de cuidado y clínicas veterinarias.
- ✓ **categorias_animales:** Pantalla de categorías de animales.
- ✓ **animales_domesticos:** Pantalla con la lista de animales domésticos.

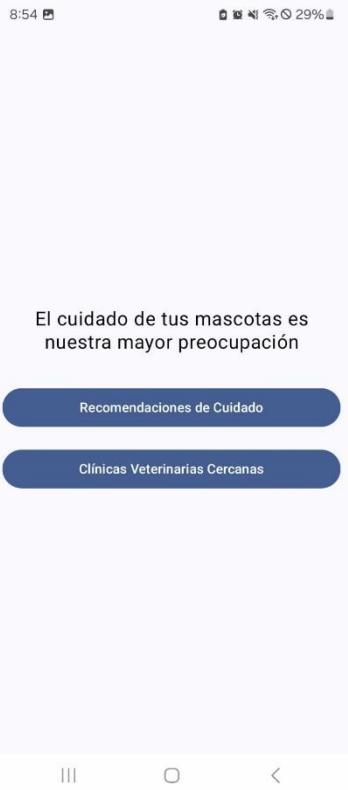
- ✓ **animales_granja:** Pantalla con la lista de animales de granja.
- ✓ **clinicas_veterinarias:** Pantalla de la lista de clínicas veterinarias.

The screenshot shows the Android Studio interface with the file `Navigation.kt` open in the editor. The code implements a navigation graph for a pet care application. It defines a `NavHost` for the initial screen and composable screens for categories of animals and animal farms. The code includes comments in Spanish explaining the purpose of each section.

```
7 @Composable
8 fun AppNavigation() {
9     val navController = rememberNavController()
10
11     NavHost(navController = navController, startDestination = "pantalla_inicial") {
12         // Pantalla inicial
13         composable(route: "pantalla_inicial") {
14             PantallaInicial(
15                 onCuidadoMascotasClick = { navController.navigate(route: "categorias_animales") }, // Navegar a la nueva pantalla
16                 onClinicasClick = { navController.navigate(route: "clinicas_veterinarias") }
17             )
18         }
19     }
20
21     // Nueva pantalla de Categorías de Animales
22     composable(route: "categorias_animales") {
23         CategoriasDeAnimales(
24             navController = navController,
25             onAnimalDomesticosClick = { navController.navigate(route: "animales_domesticos") },
26             onAnimalGranjaClick = { navController.navigate(route: "animales_granja") }
27         )
28     }
29     // Pantalla de Recomendaciones de Cuidado (Perros, Gatos, Aves, Roedores)
30     composable(route: "animales_domesticos") {
31         AnimalesDomesticos(navController)
32     }
33     composable(route: "animales_granja") {
34         AnimalesGranja(navController)
35     }
36 }
```

Aspecto Final del apartado “CONSEJOS DE CUIDADO DE MASCOTAS”

8:54



El cuidado de tus mascotas es nuestra mayor preocupación

[Recomendaciones de Cuidado](#)

[Clínicas Veterinarias Cercanas](#)

III ○ <

8:54



Animales Domésticos

[Recomendaciones Generales](#)

[Ver Animales](#)



Animales de Granja

[Recomendaciones Generales](#)

[Ver Animales](#)



III ○ <

8:55



Clínicas Veterinarias Cercanas

Clínica PetSave

Dirección: Calle 3, Jalapa

Teléfono: 3344-2215

Clínica CleanPaw

Dirección: Calle 4, Jalapa

Teléfono: 1122-3344

Clínica PawFriends

Dirección: Calle Principal los perros, Jalapa

Teléfono: 5566-7788

Safe Pets Clinic

Dirección: Calle 2-4, Mataquescuintla

Teléfono: 1122-3344

III ○ <

10. SISTEMA DE MENSAJERÍA DIRECTA:

- Los usuarios podrán chatear entre ellos para coordinar servicios, discutir detalles de adopciones o preguntar sobre mascotas reportadas.
- Opción de habilitar notificaciones para mensajes importantes.

MESSAGE

Modelo de Datos Message

Define un modelo de datos Message que representa un mensaje en la aplicación de chat.

Propiedades:

- userEmail: Cadena que almacena el correo electrónico del usuario que envió el mensaje.
- content: Cadena que contiene el contenido del mensaje.

Características:

- La clase es un data class, lo que proporciona automáticamente métodos equals(), hashCode(), y toString() entre otros, facilitando la manipulación y comparación de objetos de tipo Message.

```
data class Message(  
    val userEmail: String, // Cambia el nombre de la propiedad a userEmail  
    val content: String  
)
```

ViewModel MessagingViewModel

Esta clase MessagingViewModel extiende ViewModel y se utiliza para gestionar el estado de los mensajes en la aplicación de chat.

Propiedades:

- _messages: Un MutableStateFlow que almacena una lista mutable de objetos Message. Comienza como una lista vacía.
- messages: Un flujo de estado inmutable (StateFlow) que permite a las vistas observar los mensajes sin poder modificar la lista directamente.

Esto promueve un diseño más seguro y predecible al trabajar con estados.

```
class MessagingViewModel : ViewModel() {  
    private val _messages = MutableStateFlow<List<Message>>(emptyList())  
    val messages = _messages.asStateFlow()
```

MÉTODO SENDMESSAGE

Agrega un nuevo mensaje a la lista de mensajes existente.

Parámetros:

- content: El contenido del mensaje que se desea enviar.
- userEmail: El correo electrónico del usuario que envía el mensaje.

Operaciones:

- Crea un nuevo objeto Message utilizando el contenido y el correo electrónico del usuario.
- Actualiza el valor de _messages añadiendo el nuevo mensaje a la lista existente. Esto se realiza mediante la operación de concatenación (+), que crea una nueva lista que incluye todos los mensajes anteriores más el nuevo mensaje.

```
fun sendMessage(content: String, userEmail: String) {  
    val newMessage = Message(userEmail = userEmail, content = content)  
    _messages.value = _messages.value + newMessage  
}
```

El código define un modelo de datos Message para representar los mensajes en un chat y un ViewModel MessagingViewModel que gestiona una lista de mensajes utilizando StateFlow. Este diseño permite una comunicación reactiva entre el modelo de datos y las vistas de la aplicación, facilitando la actualización de la interfaz de usuario en respuesta a los cambios en el estado de los mensajes.

ChatNavigationBar

Esta función define una barra de navegación en la parte inferior de la pantalla para la aplicación de chat.

- **Parámetros:**

- navController: Un controlador de navegación que gestiona las rutas y la navegación entre pantallas.
- uid: El identificador único del usuario que se utiliza para navegar a la pantalla de perfil.

```
@Composable  
fun ChatNavigationBar(navController: NavHostController, uid: String) {
```

BottomAppBar

Crea una barra de aplicaciones en la parte inferior de la pantalla.

Parámetros:

- modifier: Permite ajustar la barra para que ocupe el ancho completo de la pantalla (fillMaxWidth()).
- containerColor: Establece el color de fondo de la barra utilizando el color primario del tema actual de la aplicación.

```
BottomAppBar(  
    modifier = Modifier.fillMaxWidth(),  
    containerColor = MaterialTheme.colorsScheme.primary  
) {
```

Elementos de Navegación

Define un elemento de navegación en la barra inferior.

Parámetros:

- icon: Composable que muestra el ícono de inicio. Icons.Default.Home representa el ícono de inicio y contentDescription proporciona una descripción accesible para usuarios con discapacidad visual.
- label: Composable que muestra el texto "Inicio" bajo el ícono.
- selected: Indica si este elemento de navegación está seleccionado. Actualmente se establece en false, pero debería actualizarse a true dependiendo de la pantalla actual para reflejar la navegación activa.
- onClick: Define la acción que se ejecuta al hacer clic en este elemento, en este caso, navega a la pantalla "home".

```
// Ajusta los iconos y rutas según pantallas.  
NavigationBarItem(  
    icon = { Icon(Icons.Default.Home, contentDescription = "Inicio") },  
    label = { Text("Inicio") },  
    selected = false, // Cambia a true según la pantalla actual  
    onClick = { navController.navigate("home") }  
)
```

Segundo Elemento de Navegación

Define un segundo elemento de navegación para la barra inferior.

Parámetros:

- icon: Composable que muestra el ícono del perfil. Icons.Default.Person representa el ícono de perfil.
- label: Composable que muestra el texto "Perfil".
- selected: Al igual que en el elemento anterior, este indica si este elemento está seleccionado.
- onClick: Navega a la pantalla de perfil del usuario, pasando el uid como parte de la ruta.

```
NavigationBarItem(  
    icon = { Icon(Icons.Default.Person, contentDescription = "Perfil") },  
    label = { Text("Perfil") },  
    selected = false, // Cambia a true según la pantalla actual  
    onClick = { navController.navigate("profile/$uid") }  
)
```

ChatNavigationBar que crea una barra de navegación en la parte inferior de la interfaz de usuario de la aplicación de chat. Utiliza BottomAppBar y NavigationBarItem de la biblioteca Material3 de Jetpack Compose para proporcionar navegación entre las pantallas "Inicio" y "Perfil". La barra de navegación es responsive, ocupando el ancho completo de la pantalla y utilizando los colores del tema de la aplicación para su diseño visual. La lógica de selección y navegación está diseñada para ser flexible, permitiendo la adaptación a las necesidades de la aplicación.

MainScreen

Esta función define la pantalla principal de la aplicación de chat, donde se inicializan componentes y se configuran efectos secundarios.

```
@Composable  
fun MainScreen() {
```

Obtener el Contexto

Obtiene el contexto actual de la aplicación. Este contexto se utiliza para acceder a recursos, servicios y funcionalidades específicas de Android, como la creación de canales de notificación.

```
val context = LocalContext.current
```

Efecto Secundario con LaunchedEffect

Ejecuta un efecto secundario cuando el contexto cambia. Aquí, se utiliza para crear un canal de notificación.

Detalles:

- key1 = context: Esta es la clave que desencadena el efecto. Si el contexto cambia (por ejemplo, si se vuelve a componer la pantalla), se ejecutará nuevamente.
- Dentro del bloque LaunchedEffect, se llama a createNotificationChannel(context) para asegurarse de que el canal de notificación esté creado. Esto es importante para enviar notificaciones en Android, especialmente para dispositivos que ejecutan Android O (API 26) o superior.

```
LaunchedEffect(key1 = context) {  
    createNotificationChannel(context)  
}
```

Controlador de Navegación

Crea un controlador de navegación que se recuerda entre recomposiciones. Esto es esencial para gestionar la navegación en la aplicación.

Funcionalidad:

- navController se utilizará para navegar entre diferentes pantallas de la aplicación, permitiendo pasar datos y controlar el flujo de la aplicación.

```
val navController = rememberNavController()
```

Llamada al Composable App

Llama al composable App, pasando el navController como argumento.

Funcionalidad:

- Este paso es crucial para integrar el sistema de navegación en la aplicación. El composable App probablemente contendrá la definición de las diferentes rutas y pantallas, utilizando el navController para manejar las transiciones entre ellas.

```
// Llamada al composable App pasándole el navController
App(navController = navController)
```

El código define una clase de datos llamada Mensaje, que representa un mensaje en un sistema de mensajería en la aplicación "Paw Friends". La clase tiene tres propiedades: autorId, contenido y timestamp, cada una de las cuales se anota con @PropertyName para especificar el nombre del campo correspondiente en Firestore. Esto permite que la clase se utilice para almacenar y recuperar mensajes en la base de datos Firestore de manera efectiva, asegurando que las propiedades de la clase se mapeen correctamente a los campos de la base de datos. La estructura de esta clase facilita la manipulación y el manejo de mensajes dentro de la aplicación de chat.

```
package com.enma.pawfriends.mensages.chat.ui1

import com.google.firebase.firestore.PropertyName

data class Mensaje(
    @PropertyName("autorld") val autorld: String = "",
    @PropertyName("contenido") val contenido: String = "",
    @PropertyName("timestamp") val timestamp: String = "",
```

Define una clase de datos Message que se utiliza para representar un mensaje en un sistema de mensajería. La clase tiene tres propiedades:

- userEmail: el correo electrónico del usuario que envió el mensaje.
- content: el texto del mensaje.
- timestamp: la marca de tiempo que indica cuándo se envió el mensaje.

Esta estructura es esencial para gestionar y mostrar mensajes en una aplicación de chat, ya que permite almacenar tanto el contenido como la información del remitente y el tiempo en que fue enviado.

```
package com.enma.pawfriends.mensages.chat.ui1

data class Message(
    val userEmail: String = "", // Propiedad del remitente
    val content: String = "", // Contenido del mensaje
    val timestamp: Long = System.currentTimeMillis() // Marca de tiempo para
ordenar los mensajes
)
```

MessagingScreen

Constante CHANNEL_ID

```
const val CHANNEL_ID = "direct_message_channel"
```

Propósito: Esta constante define el identificador único del canal de notificaciones, CHANNEL_ID utilizado para referenciar el canal en toda la

aplicación. Este ID se usa al crear o actualizar el canal, y también al enviar notificaciones a través de este canal.

Tipo: Stringin mutable y constante en tiempo de compilación.

Valor: "direct_message_channel".

Función createNotificationChannel

```
9
0     fun createNotificationChannel(context: Context) {
```

- Propósito: La función createNotificationChannel configura y crea un canal de notificaciones si el dispositivo cumple con el requisito de la versión mínima de Android.
- Parámetros: context: Context, contexto necesario para acceder a los servicios del sistema Android, en este caso, el servicio de notificaciones.

Verificación de Versión del Sistema Operativo

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
```

Propósito: Dado que los canales de notificación fueron introducidos en Android, este bloque condicional verifica si el dispositivo ejecuta al menos esta versión antes de intentar crear el canal.

Creación del Canal de Notificación

Componentes:

- **name:** Defina el nombre visible del canal, en este caso, "Mensajes Directos" que aparecerá en la configuración de notificaciones del dispositivo.
- **descriptionText:** Definir una descripción breve del canal: "Notificaciones de mensajes directos".
- **importance:** Nivel de importancia del canal, en este caso, NotificationManager.IMPORTANCE_HIGH. Esto asegura que las notificaciones aparecerán como alertas en la pantalla.
- **channel:** Instancia de NotificationChannelconfigurada con el ID, nombre, nivel de importancia y descripción.

Acceso al NotificationManager

```
const val CHANNEL_ID = "direct_message_channel"

fun createNotificationChannel(context: Context) {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        val name = "Mensajes Directos"
        val descriptionText = "Notificaciones de mensajes directos"
        val importance = NotificationManager.IMPORTANCE_HIGH
        val channel = NotificationChannel(CHANNEL_ID, name, importance).apply {
            description = descriptionText
        }
        val notificationManager: NotificationManager =
            context.getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
        notificationManager.createNotificationChannel(channel)
    }
}
```

Propósito: Obtiene una instancia de NotificationManager usando el contexto y el servicio de notificaciones del sistema.

Función `createNotificationChannel`: Registrar el canal de notificación con el sistema operativo, permitiendo que las notificaciones enviadas a través de este canal sigan la configuración definida por el usuario.

MessagingViewModel

Importaciones de Paquetes

```
package com.enma.pawfriends.chat.viewmodel

import androidx.lifecycle.ViewModel
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.asStateFlow
```

androidx.lifecycle.ViewModel: Proporciona la clase base ViewModel que almacena y gestiona datos relacionados con la interfaz de usuario de manera que sea resistente a cambios de configuración.

kotlinx.coroutines.flow.MutableStateFlow y **asStateFlow**: MutableStateFlow es un flujo de datos en tiempo real que se usa para mantener y emitir el estado actual de un valor mutable. La extensión `asStateFlow()` convierte el MutableStateFlow en un flujo inmutable, exponiéndolo de forma segura fuera del ViewModel.

Clase de datos Message

```
data class Message(  
    val userEmail: String, // Cambia el nombre de la propiedad a userEmail  
    val content: String  
)
```

Propósito: La clase de datos Message representa un mensaje enviado en el chat.

Propiedades:

- **userEmail:** String que contiene el correo electrónico del usuario que envía el mensaje.
- **content:** String que contiene el contenido del mensaje.
- **Uso:** Message se utiliza para encapsular la información de cada mensaje que se envía o recibir, facilitando su manipulación dentro de listas de mensajes.

Clase MessagingViewModel

```
class MessagingViewModel : ViewModel() {
```

Propósito: MessagingViewModel gestiona la lista de mensajes y exponen un flujo reactivo (StateFlow) para que la interfaz de usuario pueda observar y actualizarse automáticamente cuando se envían nuevos mensajes.

Herencia: Extiende ViewModel, lo que permite que los datos en esta clase sobrevivan a cambios de configuración.

Propiedad privada _messages

```
private val _messages = MutableStateFlow<List<Message>>(emptyList())  
val messages = _messages.asStateFlow()
```

Tipo: MutableStateFlow<List<Message>>— un flujo mutable que contiene una lista de Message.

Inicialización: La lista de mensajes se inicializa como vacía (emptyList()).

Visibilidad: Privada (_messages) para que no se pueda modificar directamente desde fuera de la clase.

Uso: _messages el contenedor de datos internos que se actualizará cada vez que se envíe un nuevo mensaje.

Función sendMessage

```
fun sendMessage(content: String, userEmail: String) {  
    val newMessage = Message(userEmail = userEmail, content = content)  
    _messages.value = _messages.value + newMessage  
}
```

Propósito: La función sendMessage permite agregar un nuevo mensaje a la lista de mensajes almacenados en _messages.

Parámetros

- **content:** El texto del mensaje que se desea enviar.
- **userEmail:** El correo electrónico del usuario que envía el mensaje.

Aspecto Final del Chat

Chat Global

Fri Nov 01 21:20:33 CST 2024

Autor: X5jr1357vlXODM77Ju91pLCJaGx1
Frgg
Sat Nov 02 01:27:37 CST 2024

Autor: X5jr1357vlXODM77Ju91pLCJaGx1
Jhyhy
Sat Nov 02 10:43:36 CST 2024

Autor: X5jr1357vlXODM77Ju91pLCJaGx1
Hola
Sat Nov 02 20:31:27 CST 2024

Autor: X5jr1357vlXODM77Ju91pLCJaGx1
Como te va
Sat Nov 02 20:31:32 CST 2024

Autor: X5jr1357vlXODM77Ju91pLCJaGx1
Que haces
Sat Nov 02 20:31:36 CST 2024

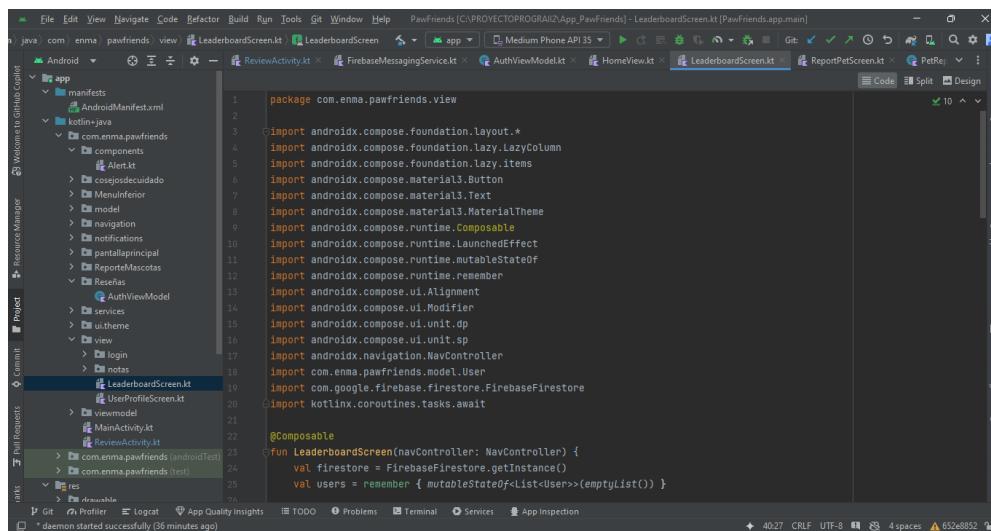
Escribe un mensaje... Enviar

11. RECOMPENSAS Y RECONOCIMIENTO DENTRO DE LA COMUNIDAD:

- Sistema de **puntos y medallas** para los usuarios más activos, por ejemplo, aquellos que adoptan mascotas, ayudan a encontrar mascotas perdidas, o proporcionan servicios de cuidado.
- Los usuarios podrán ganar **reconocimientos** como "**Rescatador del mes**" o "**Anfitrión destacado**", incentivando la participación y el apoyo comunitario.

LeaderboardScreen.kt

Pantalla Inicial:



```
1 package com.emma.pawfriends.view
2
3 import androidx.compose.foundation.layout.*
4 import androidx.compose.foundation.lazy.LazyColumn
5 import androidx.compose.foundation.lazy.items
6 import androidx.compose.material3.Button
7 import androidx.compose.material3.Text
8 import androidx.compose.material3.MaterialTheme
9 import androidx.compose.runtime.Composable
10 import androidx.compose.runtime.LaunchedEffect
11 import androidx.compose.runtime.mutableStateOf
12 import androidx.compose.runtime.remember
13 import androidx.compose.ui.Alignment
14 import androidx.compose.ui.Modifier
15 import androidx.compose.ui.unit.dp
16 import androidx.compose.ui.unit.sp
17 import androidx.navigation.NavController
18 import com.emma.pawfriends.model.User
19 import com.google.firebase.firestore.FirebaseFirestore
20 import kotlinx.coroutines.tasks.await
21
22 @Composable
23 fun LeaderboardScreen(navController: NavController) {
24     val firestore = FirebaseFirestore.getInstance()
25     val users = remember { mutableStateOf<List<User>>(emptyList()) }
26 }
```

Paquete: com.emma.pawfriends.view

Documentación Técnica:

Descripción

General:

Este archivo define una función composable, `LeaderboardScreen`, que se utiliza para mostrar una clasificación de usuarios obtenida de Firestore. La clasificación se ordena según los puntos de los usuarios en orden descendente, y se muestran los nombres, puntos y reconocimientos de cada usuario. Esta pantalla también incluye un botón para navegar de vuelta al menú principal.

Dependencias:

- Firebase Firestore:** Utilizado para obtener los datos de los usuarios.
- Jetpack Compose:** Utilizado para construir la interfaz de usuario.
- Corutinas de Kotlin:** Utilizadas para manejar la obtención de datos de forma asíncrona.

Componentes:

- NavController:** Maneja la navegación dentro de la aplicación.

-Firestore: Utilizado para obtener los datos de los usuarios de la colección "Users".

-LazyColumn: Utilizado para mostrar la lista de usuarios.

- Button: Utilizado para navegar de vuelta al menú principal.

Funcionalidad:

1. LeaderboardScreen(navController: NavController):

- Esta es la función composable principal definida en este archivo. Toma un parámetro NavController para manejar la navegación.

- Se utiliza Firestore para obtener los datos de los usuarios de la colección "Users".

- Los usuarios se muestran en orden descendente según sus puntos, y sus detalles se muestran en una lista.

Variables:

- firestore: Instancia de `FirebaseFirestore` utilizada para interactuar con la base de datos de Firestore. users: Estado mutable para almacenar la lista de usuarios, inicialmente establecida como una lista vacía.

Elementos de la UI:

- Text: Muestra el título "Clasificación de Usuarios".

- LazyColumn: Muestra la lista de usuarios obtenidos de Firestore. Para cada usuario, se muestran su nombre, puntos y reconocimientos.

- Button: Permite al usuario navegar de vuelta al menú principal.

- Obtención de Datos:

- Se utiliza un bloque `LaunchedEffect` para obtener datos de Firestore de forma asíncrona.

- La colección "Users" se ordena por el campo "points" en orden descendente, y los datos se obtienen utilizando `await()`.

- En caso de alguna excepción durante el proceso de obtención de datos, se captura y se registra.

2. Contenido de LazyColumn:

- Muestra la información de cada usuario en un `Column` separado.

- Detalles del Usuario:

- **Nombre:** Mostrado con un tamaño de fuente de 20.sp.

- **Puntos:** Mostrado con un tamaño de fuente de 18.sp.

- **Reconocimientos:** Si el usuario tiene reconocimientos, se muestran con un tamaño de fuente de 18.sp, separados por comas.

- **Spacer:** Añade espacio entre los detalles de los usuarios.

3. Navegación:

- El botón proporcionado al final de la pantalla navega al usuario de vuelta al menú principal (identificado por la ruta "elementos").

Interfaz de Usuario:

-Diseño: El diseño general de la pantalla se organiza usando un composable `Column` con modificadores para el padding y la alineación.

-Estilo de Texto: El título y los detalles de los usuarios se estilizan usando `MaterialTheme` y tamaños de fuente específicos.

-Botón: Se utiliza un botón de ancho completo para la navegación al menú principal.

Manejo de Errores:

-El proceso de obtención de datos está envuelto en un bloque try-catch para manejar posibles excepciones.

-Actualmente, las excepciones se registran usando `e.printStackTrace()`, pero se puede agregar una lógica de manejo de errores personalizada según sea necesario.

Uso:

-Esta función composable debe ser utilizada en una pantalla que muestre una clasificación de usuarios.

-El parámetro `navController` es necesario para facilitar la navegación desde esta pantalla.

Notas:

- Las consultas de Firestore se ejecutan de forma asíncrona y la interfaz de usuario se actualizará automáticamente una vez que se obtengan los datos.

- La ordenación de los usuarios por puntos garantiza que los usuarios con más puntos se muestren primero.

Perfil de usuario Screen.kt

Paquete: com.enma.pawfriends.view

Documentación técnica:

Descripción General: Este archivo define una función componible, `UserProfileScreen` que se utiliza para mostrar el perfil de un usuario. Los datos del perfil se obtienen de Firestore, y se muestra la información como el nombre, correo electrónico, puntos y reconocimientos del usuario. Si ocurre algún error, se mostrará un mensaje de error correspondiente.

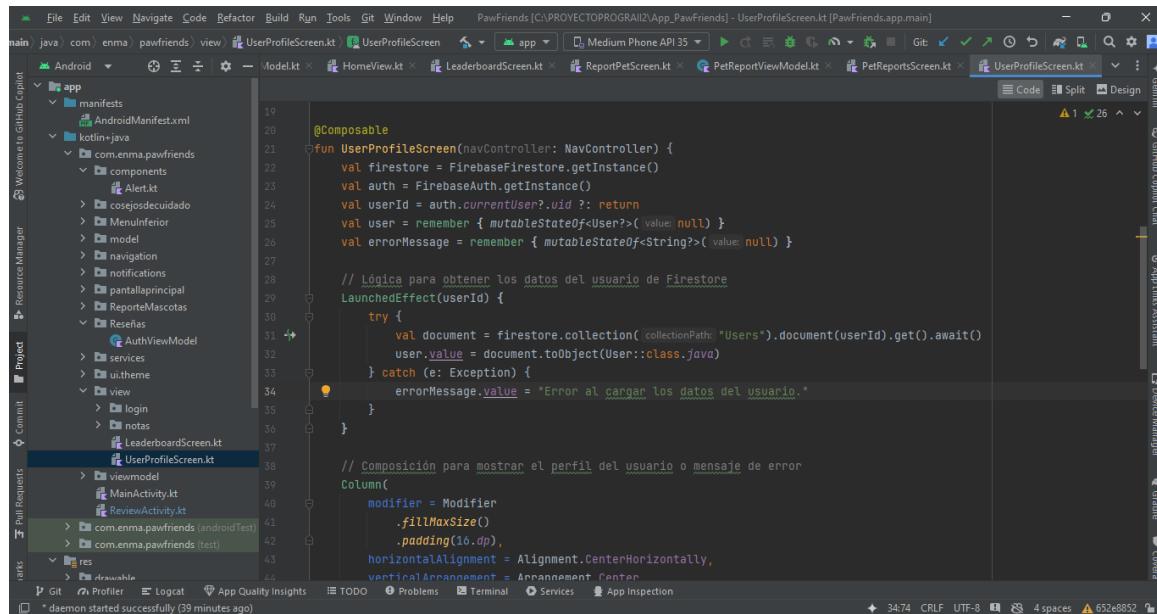
Dependencias:

- Firebase Firestore: Utilizado para obtener los datos del usuario.

- Autenticación de Firebase: Utilizado para autenticar al usuario actual.
- Jetpack Compose: Utilizado para construir la interfaz de usuario.
- Corutinas de Kotlin: Utilizadas para manejar la obtención de datos de forma asíncrona.

Componentes:

- NavController: Maneja la navegación dentro de la aplicación.
- FirebaseAuth: Utilizado para obtener la instancia de autenticación del usuario actual.
- Firestore: Utilizado para obtener los datos del usuario de la colección "Users".
- **UserProfileScreen**



```

    @Composable
    fun UserProfileScreen(navController: NavController) {
        val firestore = FirebaseFirestore.getInstance()
        val auth = FirebaseAuth.getInstance()
        val userId = auth.currentUser?.uid ?: return
        val user = remember { mutableStateOf<User?> (value: null) }
        val errorMessage = remember { mutableStateOf<String?> (value: null) }

        // Lógica para obtener los datos del usuario de Firestore
        LaunchedEffect(userId) {
            try {
                val document = firestore.collection(collectionPath: "Users").document(userId).get().await()
                user.value = document.toObject(User::class.java)
            } catch (e: Exception) {
                errorMessage.value = "Error al cargar los datos del usuario."
            }
        }

        // Composición para mostrar el perfil del usuario o mensaje de error
        Column(
            modifier = Modifier
                .fillMaxSize()
                .padding(16.dp),
            horizontalAlignment = Alignment.CenterHorizontally,
            verticalArrangement = Arrangement.Center
        )
    }
}

```

(navController: NavController):

- Esta es la función componible principal definida en este archivo. Toma un parámetro NavController para manejar la navegación.
- Obtiene el ID del usuario actualmente autenticado usando FirebaseAuth.
- Se utiliza Firestore para obtener los datos del usuario con el ID obtenido.

○ Variables:

- firestore: Instancia de FirebaseFirestore utilizada para interactuar con la base de datos.

- auth: Instancia de FirebaseAuth utilizada para autenticar al usuario.
- userId: ID del usuario autenticado. Si no hay usuario autenticado, la función retorna.
- user: Estado mutable para almacenar los datos del usuario, inicialmente establecido en null.
- errorMessage: Estado mutable para almacenar mensajes de error, inicialmente establecido en null.

- **Obtención de datos:**

- Se utiliza un bloque LaunchedEffect para obtener los datos del usuario desde Firestore de forma asíncrona.
- Se intenta obtener el documento correspondiente al usuario desde la colección "Users" y mapearlo al objeto User.
- En caso de una excepción durante el proceso, se asigna un mensaje de error a errorMessage.

2. Interfaz de usuario:

- **Título:**

- Text: Muestra el título "Perfil de Usuario" estilizado con MaterialTheme.

- **Detalles del usuario:**

- Si los datos del usuario (user.value) están disponibles, se muestra su información:
 - **Nombre:** Mostrado con un tamaño de fuente de 20.sp.
 - **Correo:** Mostrado con un tamaño de fuente de 18.sp.
 - **Puntos:** Mostrado con un tamaño de fuente de 18.sp.
 - **Reconocimientos:** Si existen, se muestran separados por comas con un tamaño de fuente de 18.sp.

- **Mensaje de error:**

- Si se produjo un error (errorMessage.value), se muestra un mensaje con el color de error definido en MaterialTheme.

3. Diseño:

- La pantalla está compuesta por un Column que organiza los elementos verticalmente.
- El Column tiene modificadores que aseguran que la pantalla se llene completamente (fillMaxSize()), con un padding de 16.dp para el margen.
- Alineación horizontal y vertical establecidas para centrar el contenido en la pantalla.

Manejo de errores:

- Si ocurre un error durante la obtención de los datos del usuario, se captura la excepción y se asigna un mensaje de error a errorMessage.
- El mensaje de error se muestra en la interfaz de usuario para informar al usuario sobre el problema.

Uso:

- Esta función componible debe ser utilizada en una pantalla que muestre la información del perfil del usuario autenticado.
- El parámetro navController es necesario para facilitar la navegación desde esta pantalla.

Notas:

- Las consultas de Firestore se ejecutarán de forma asíncrona y la interfaz de usuario se actualizará automáticamente una vez que se obtengan los datos.
- Si el usuario no está autenticado, la pantalla no mostrará ningún contenido, ya que se retorna antes de realizar cualquier acción.