**Documentación de la Aplicación**

**Repositorio de Autenticación (AuthRepository)**

**Descripción:**

La autenticación es una parte crucial de cualquier aplicación centrada en el usuario. En **healthccheckapp**, esta lógica se encapsula en el repositorio de autenticación.

**Código:**

class AuthRepositoryImpl @Inject constructor(private val firebaseAuth: FirebaseAuth): AuthRepository {

// El usuario actualmente autenticado.

override val currentUser: FirebaseUser? get() = firebaseAuth.currentUser

// Iniciar sesión con correo electrónico y contraseña.

override suspend fun login(email: String, password: String): Response<FirebaseUser> {

// Tratamiento de errores para iniciar sesión.

}

// Registrar un nuevo usuario.

override suspend fun signup(user: User): Response<FirebaseUser> {

// Tratamiento de errores para el registro.

}

// Cerrar la sesión del usuario actual.

override fun logout() {

firebaseAuth.signOut()

}

}

}

**Detalles:**

* **firebaseAuth**: Es una instancia de **FirebaseAuth**, que maneja las operaciones de autenticación en Firebase.
* **login**: Este método intenta autenticar a un usuario con un email y contraseña específicos.
* **signup**: Aquí, un nuevo usuario intenta registrarse con sus detalles, que incluyen correo electrónico y contraseña.
* **logout**: Este método simplemente cierra la sesión del usuario actual.

**Modelo de Respuesta (Response)**

**Descripción:**

Para manejar las respuestas de diversas operaciones en la aplicación (por ejemplo, operaciones de red), se utiliza un modelo de respuesta genérico.

**Código:**

sealed class Response<out T>{

object Loading: Response<Nothing>()

data class Success<out T>(val data: T): Response<T>()

data class Failure<out T>(val exception: Exception?): Response<T>()

}: Exception?): Response<T>()

}

**Detalles:**

Este modelo es una clase sellada que puede tomar uno de tres estados:

* **Loading**: Indica que una operación está en curso.
* **Success**: Indica que la operación fue exitosa y contiene los datos resultantes.
* **Failure**: Indica que la operación falló y contiene información sobre el error.

**ViewModel de Nuevo Post (NewPostViewModel) VER EN PAQUETE PRESENTATIONS/SCREENS/…**

**Descripción:**

El ViewModel **NewPostViewModel** se encarga de la lógica relacionada con la creación de nuevos posts o registros de salud.

**Código:**

@HiltViewModel

class NewPostViewModel @Inject constructor(...) {

// Variables y métodos...

fun createPost(post: Post) = viewModelScope.launch {

// Lógica para crear un nuevo post.

}

// ... otros métodos auxiliares...

} // Lógica para crear un nuevo post.

}

étodos auxiliares...

}

**Detalles:**

* **state**: Representa el estado actual del formulario de post. Es observable y se actualiza según las interacciones del usuario.
* **createPost**: Este método inicia la creación de un nuevo post en el backend.
* **onNewPost**: Prepara y envía un nuevo post basado en el estado actual.
* **pickImage y takePhoto**: Estos métodos facilitan la selección de imágenes desde la galería o la captura de nuevas fotos.

**Utilidad de Proveedor de Archivos (ComposeFileProvider)**

**Descripción:**

Gestionar archivos e imágenes en una aplicación puede ser complicado. **ComposeFileProvider** es una clase de utilidad que facilita este proceso.

**Código:**

class ComposeFileProvider: FileProvider(R.xml.file\_paths) {

// ... métodos y lógica...

} // ... métodos y lógica...

}

**Detalles:**

Esta clase proporciona métodos para:

* **createFileFromUri**: Convierte un **Uri** en un archivo en el sistema de archivos del dispositivo.
* **getImageUri**: Genera un `Uri’ para una imagen en particular en el directorio de caché.
* **getPathFromBitmap**: Convierte un objeto **Bitmap** en un archivo y devuelve su ruta.

Estos métodos son esenciales para manejar imágenes en la aplicación, ya sea al cargarlas, guardarlas o convertirlas para diferentes usos.

**Modelo de Usuario (User)**

**Descripción:**

Un modelo fundamental en la mayoría de las aplicaciones centradas en el usuario es el modelo de usuario. En **healthccheckapp**, el modelo de usuario se encarga de representar a un usuario en el sistema.

**Código:**

data class User(

var id: String = "",

var username: String = "",

var email: String = "",

var password: String = "",

var image: String = ""

) {

fun toJson(): String { ... }

companion object {

fun fromJson(data: String): User { ... }

}

}... }

}

**Detalles:**

* El modelo **User** tiene cinco propiedades: **id**, **username**, **email**, **password** y **image**.
* **toJson**: Convierte el objeto de usuario en una representación en formato JSON, útil para enviarlo a través de la red o guardarlo en una base de datos.
* **fromJson**: Realiza la operación inversa al anterior, convirtiendo una cadena JSON en un objeto **User**.

**Arquitectura y Diseño**

Basado en el patrón **Clean Architecture**, se han separado las responsabilidades en diferentes capas:

* **Presentation**: Donde se manejan las interacciones de la interfaz de usuario, como **ViewModels** y **Activities**.
* **Domain**: Contiene la lógica de negocio y las definiciones de modelos.
* **Data**: Se encarga de la comunicación con fuentes de datos, como bases de datos o servicios web.

El uso de **MVVM** facilita la conexión entre las capas de Presentation y Domain, permitiendo una separación clara entre la lógica de la interfaz de usuario y la lógica de negocio.

**FEATURES y sus funcionalidades**

**1. Autenticación y Gestión de Usuarios**

1.1 Descripción:

La autenticación y gestión de usuarios es una parte importante de **healthccheckapp**. Estas características garantizan que cada usuario tenga una experiencia personalizada y segura dentro de la aplicación.

1.2 Funcionalidad:

* **Inicio de Sesión**:
* **Descripción:** Los usuarios pueden iniciar sesión en la aplicación usando su dirección de correo electrónico y contraseña.
* **Fragmento de código:**

override suspend fun login(email: String, password: String):Response<FirebaseUser> {

return try{

val result = firebaseAuth.signInWithEmailAndPassword(email, password).await()

Response.Success(result.user!!)

}

catch (e:Exception){

e.printStackTrace()

Response.Failure(e)

}

}}

Aquí se usa la función **signInWithEmailAndPassword** de FirebaseAuth para autenticar al usuario. Si la autenticación es exitosa, devuelve un **FirebaseUser**.

**Registro**:

**Descripción:** Los nuevos usuarios pueden crear una cuenta proporcionando su dirección de correo electrónico y contraseña.

* **Fragmento de código:**

override suspend fun signup(user: User): Response<FirebaseUser> {

return try {

val result = firebaseAuth.createUserWithEmailAndPassword(user.email, user.password).await()

Response.Success(result.user!!)

}catch(e: Exception){

e.printStackTrace()

Response.Failure(e)

}

* }
* e.printStackTrace()

Similar al inicio de sesión, la función **createUserWithEmailAndPassword** de FirebaseAuth se utiliza para registrar un nuevo usuario. Si el registro es exitoso, devuelve un **FirebaseUser**.

**Perfil de Usuario**:

**Descripción:** Cada usuario tiene una identidad única en la aplicación, que consta de un ID, nombre de usuario, correo electrónico, contraseña y una imagen opcional.

* **Fragmento de código:**

data class User(

var id: String = "",

var username: String = "",

var email: String = "",

var password: String = "",

var image: String = ""

)

Esta clase **User** actúa como un modelo para representar a un usuario en la aplicación. Los detalles como **id**, **username**, **email**, etc., se utilizan para personalizar la experiencia del usuario y también para operaciones relacionadas con la base de datos.

**Cerrar Sesión**:

**Descripción:** Los usuarios pueden cerrar sesión de la aplicación en cualquier momento para proteger su privacidad y datos.

* **Fragmento de código:**

override fun logout(){

firebaseAuth.signOut()

}

* firebaseAuth.signOut()

La función **signOut** de FirebaseAuth se utiliza para cerrar la sesión de un usuario.

**2. Gestión de Imágenes en Publicaciones**

2.1 Descripción:

La capacidad de trabajar con imágenes es esencial para proporcionar una experiencia atractiva. Las imágenes pueden ser relevantes, por ejemplo, para mostrar registros visuales de progreso en la salud o para compartir momentos significativos relacionados con el bienestar.

2.2 Funcionalidad:

* **Obtener URI de Imagen**:
* **Descripción:** Antes de guardar o mostrar una imagen, es esencial obtener su URI (Identificador de Recurso Uniforme). Este URI actúa como una referencia a la imagen.
* **Fragmento de código:**

fun getImageUri(context: Context): Uri {

val directory = File(context.cacheDir, "images")

directory.mkdirs()

val file = File.createTempFile(

"selected\_image\_",

".jpg",

directory

)

val authority = context.packageName + ".fileprovider"

return getUriForFile(

context,

authority,

file

)

}

* context,
* )

Aquí, la función **getImageUri** crea un directorio temporal en el caché del dispositivo para almacenar imágenes. Luego, utiliza **getUriForFile** para obtener el URI de la imagen.

**Convertir Bitmap a Path**:

**Descripción:** Para guardar y recuperar imágenes, a menudo es necesario convertir entre diferentes formatos. Una operación común es convertir una imagen Bitmap a una ruta de archivo (path).

**Fragmento de código:**

fun getPathFromBitmap(context: Context, bitmap: Bitmap): String {

val wrapper = ContextWrapper(context)

var file = wrapper.getDir("Images", Context.MODE\_PRIVATE)

file = File(file,"${UUID.randomUUID()}.jpg")

val stream: OutputStream = FileOutputStream(file)

bitmap.compress(Bitmap.CompressFormat.JPEG,100,stream)

stream.flush()

stream.close()

return file.path

}

Esta función **getPathFromBitmap** toma un **Bitmap** y lo guarda en el almacenamiento privado de la aplicación. Luego, devuelve la ruta de acceso al archivo de la imagen guardada.