



MYGNUHEALTH

ANDROID

SERGIO GARCÍA GÓMEZ

Trabajo fin de ciclo: Desarrollo de Aplicaciones
Multiplataforma

Tutor: Juan Antonio Carrasco

Índice

1.	Introducción.....	2
2.	Análisis de la aplicación	3
2.1.	Requisitos funcionales.	3
2.2.	Requisitos no funcionales.....	4
3.	Recursos necesarios para el desarrollo.	5
3.1.	Recursos hardware.	5
3.2.	Recursos software.....	5
4.	Metodología.....	6
4.1.	Metodología de desarrollo.	6
4.2.	Planificación.....	16
5.	Diseño del proyecto.....	22
5.1.	Diseño de la BBDD.	22
5.2.	Interacción con el usuario.....	24
5.2.1.	Casos de uso.	25
5.2.2.	Diseño de la interfaz.....	26
5.2.3.	Diseño de la arquitectura	27
5.2.4.	Arquitectura del servidor.	28
5.2.5.	Arquitectura del cliente.....	28
5.2.6.	Relación con otros sistemas.	31
6.	Detalles de implementación.....	32
7.	Fase de pruebas.....	57
7.1.	Pruebas de unidad.	57
7.2.	Pruebas de validación y aceptación.....	63
7.3.	Pruebas de usabilidad.....	63
7.4.	Pruebas de integración del sistema.....	64
8.	Trabajo futuro.....	66
9.	Conclusiones.....	67
10.	Manuales.....	68
10.1.	Manual de Usuario.....	68
10.2.	Manual de Administrador.....	76
11.	Descargar aplicación	78
12.	Bibliografía	79

1. Introducción.

El objetivo de este proyecto es desarrollar una aplicación móvil para la gestión de datos médicos en entornos hospitalarios, concebida para uso en países con bajo desarrollo, pero adaptable a otros contextos por su software libre. La aplicación busca facilitar la comunicación y el intercambio de información entre pacientes y profesionales de la salud, tanto en el ámbito clínico como en el domiciliario.

Para abordar las necesidades específicas de este contexto, se ha diseñado una solución versátil que permite la captura de datos médicos tanto en consultas médicas como en el hogar del paciente. La aplicación ofrece la capacidad de recopilar datos de dispositivos médicos a través de Bluetooth, tensiómetros y termómetros, y enviarlos de forma segura a un servidor central para almacenar y acceder después.

Inicialmente, se consideró la plataforma Kivy para el desarrollo, que permite aplicaciones multiplataforma utilizando Python. Sin embargo, tras evaluar las limitaciones relacionadas con la integración de dispositivos médicos mediante Bluetooth y la distribución en dispositivos Android, se optó por desarrollar la aplicación de forma nativa en Android Studio.

Aunque esta decisión implica renunciar al software libre en su totalidad, ofrece la oportunidad de explorar a fondo las posibilidades y limitaciones de la aplicación en un entorno real, adaptándose de manera más efectiva a las necesidades y requerimientos tecnológicos específicos del contexto gambiano.

La interfaz gráfica de la aplicación se diseñará con modificaciones mínimas sobre la base de la interfaz original para garantizar su funcionalidad y usabilidad para los usuarios. Asimismo, se ofrecerá la aplicación en varios idiomas para mejorar su accesibilidad y utilidad.

A medida que avance el desarrollo de la aplicación y se implemente en entornos de prueba, se realizarán ajustes y mejoras adicionales para garantizar su eficacia y fiabilidad. En las secciones posteriores de esta memoria, se proporcionarán detalles técnicos adicionales sobre la aplicación y su implementación.

El origen del proyecto radica en la dificultad de los médicos para mantener un registro eficiente de los pacientes y gestionar todos los datos clínicos. Por lo tanto, se decidió digitalizar una solución que permitiera a los profesionales acceder de manera sencilla a toda la información médica relevante de sus pacientes.

Elegí este proyecto porque me pareció una maravillosa oportunidad para colaborar con una ONG que busca ayudar a zonas desfavorecidas y mejorar la calidad de vida de personas con menos recursos.

2. Análisis de la aplicación

2.1. Requisitos funcionales.

- **R1.** El sistema debe permitir que los usuarios inicien sesión utilizando sus credenciales (nombre de usuario y contraseña).
- **R2.** Debe proporcionar registrar nuevos usuarios, permitiendo su inclusión en el sistema para futuros accesos y almacenando la información en el servidor para su posterior gestión y análisis.
- **R3.** Accesibilidad a los formularios.
 - **SR1.** Proporcionar a los usuarios un acceso independiente a los formularios de datos médicos.
- **R4.** Completar los formularios de datos médicos.
 - **SR1.** Debe proporcionar todos los elementos que añadan los datos necesarios.
- **R5.** Configuración del sistema.
 - **SR1.** Configurar conexión servidor con las credenciales necesarias.
 - **SR2.** Configurar el usuario una vez ya estamos logeados.
- **R6.** La aplicación debe subir los datos de manera unidireccional al servidor.
 - **SR1.** Debe poder tener los datos en el servidor y en el dispositivo en local.
- **R7.** Se debe implementar la capacidad de establecer conexiones con dispositivos médicos a través de Bluetooth.
 - **SR1.** La aplicación debe recibir datos actualizados de forma automática.
- **R8.** Debe tener una persistencia de los datos recibidos e introducidos.
- **R9.** Debe poder mostrar en graficas los datos médicos.

2.2. Requisitos no funcionales.

- **R1.** La aplicación debe estar optimizada para dispositivos Android de gama media, asegurando un rendimiento eficiente y una experiencia fluida para el usuario.
- **R2.** Los usuarios deben estar correctamente vinculados con sus respectivos datos para asegurar la precisión y confiabilidad de la información accedida.
- **R3.** Se debe implementar una estructura de menú principal y barra de herramientas clara y organizada, asegurando una navegación intuitiva entre las diferentes pantallas de la aplicación para una experiencia de usuario sin confusiones.
 - **SR1.** La aplicación debe proporcionar a los usuarios un acceso directo y separado a los formularios de datos médicos.
 - **SR2.** Esto garantiza que los usuarios no se desvén accidentalmente hacia otras áreas de la aplicación mientras completan los formularios.
 - **SR3.** La separación también contribuye a la precisión de los datos introducidos, ya que los usuarios no se verán distraídos por otras funciones o pantallas.
- **R4.** La interfaz gráfica debe ser responsive y adaptarse a diferentes tamaños de pantalla de dispositivos Android en formato vertical, manteniendo un aspecto de resolución de 16:9 o 16:10. Es importante destacar que, en formatos cuadrados, como 4:3, es posible que los datos no se muestren correctamente.
- **R5.** Se debe garantizar la seguridad de los datos.
 - **SR1.** Debe asegurar la confidencialidad y protección de la información sensible.
 - **SR2.** Debe cifrar las contraseñas.
- **R6.** La aplicación debe ser compatible con una amplia variedad de versiones de Android, asegurando su accesibilidad para un mayor número de usuarios y dispositivos.
- **R7.** La aplicación debe ser estable y libre de fallos frecuentes durante su uso.
- **R8.** Acceso servidor para poder enviar los datos recogidos en la aplicación.
- **R9.** La aplicación debe ser compatible con una amplia gama de dispositivos Android y todas las API desde la 27 en adelante.

3. Recursos necesarios para el desarrollo.

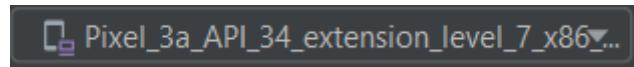
3.1. Recursos hardware.

- **Ordenador Personal:** Con características mínimas para poder utilizar el software necesarias, sacadas de las páginas oficiales:
 - Microsoft® Windows® 8/10/11 de 64 bits
 - Arquitectura de CPU x86_64; procesador Intel Core de segunda generación o posterior, o CPU AMD compatible con un hipervisor de Windows
 - 8 GB de RAM o más
 - 8 GB de espacio disponible en el disco como mínimo (IDE + SDK de Android + Android Emulator)
 - Resolución de pantalla mínima de 1280 × 800
- **Dispositivo Android (Opcional):** Un dispositivo Android real para pruebas adicionales y verificación del rendimiento en hardware real.
- **Conexión a Internet:** Necesaria para descargar bibliotecas, herramientas y actualizaciones.
- **Tensiómetro con Bluetooth:** Aparato médico para poder conectarlo con la aplicación y su prueba de transferencia de datos y posterior subida al servidor.

3.2. Recursos software.

Recursos software para el desarrollo:

- **Android Studio:** Plataforma de desarrollo oficial para Android. Se utiliza para escribir, compilar y depurar el código de la aplicación, con versión mínima 2022.3.1 Patch 1.
- **Kotlin Programming Language:** Lenguaje de programación utilizado para el desarrollo de la aplicación Android. Puedes obtener más información sobre Kotlin en este enlace. La versión usada ha sido: 1.9.0-release-358.
- **Git:** Sistema de control de versiones para el seguimiento y gestión del código fuente.
- **GitHub:** Plataforma de alojamiento de código fuente que permite el trabajo colaborativo y el control de versiones, además de tener conocimientos sobre GitFlow.
- **Android Emulator:** Herramienta para probar la aplicación en un entorno virtual antes de implementarla en un dispositivo real. Esta herramienta está incluida con Android Studio. El emulador utilizado ha sido:



Todas las versiones y emuladores utilizados son orientativos, ya que, seguro que con esas versiones se puede trabajar bajo este proyecto sin problema, pero probablemente con superiores no haya ningún problema.

Recursos software para la documentación:

- Office 365 Versión 2403 (Word, Excel)
- Draw IO Versión online
- Figma Versión online

4. Metodología.

4.1. Metodología de desarrollo.

Metodología en espiral

Para la planificación y ejecución del desarrollo de la aplicación móvil, se optó por seguir la metodología en espiral. Esta elección se basó en la naturaleza iterativa y flexible de esta metodología, lo que la hace especialmente adecuada para proyectos donde los requisitos pueden evolucionar y cambiar con el tiempo, como es el caso de una aplicación médica en constante desarrollo y mejora.

1^a Ciclo

• Fase 1: Análisis y Requisitos

En el primer ciclo, nos centramos en analizar y recoger los requisitos iniciales del proyecto. La principal tarea fue replicar funcionalidades de la aplicación existente en Android nativo.

Para ello, estudiamos el funcionamiento de la aplicación proporcionada y definimos los siguientes requisitos: la aplicación debía contar con una pantalla de inicio de sesión que permitiera crear una nueva cuenta o iniciar sesión con una existente; una vez autenticado, el usuario accedería al menú principal, desde donde no podría forzar la subida de datos al servidor.

Además, se estableció la necesidad de tener un segundo menú que llevara a pantallas dedicadas para la subida manual de datos en cada opción disponible.

La aplicación también debía incluir un menú de herramientas para modificar el perfil del usuario y las opciones del servidor.

Como añadido, se implementaron dos pantallas adicionales para la transmisión de datos a través de dispositivos Bluetooth.

Además, se estableció como requisito que todos los datos recopilados se almacenaran en las Polis.

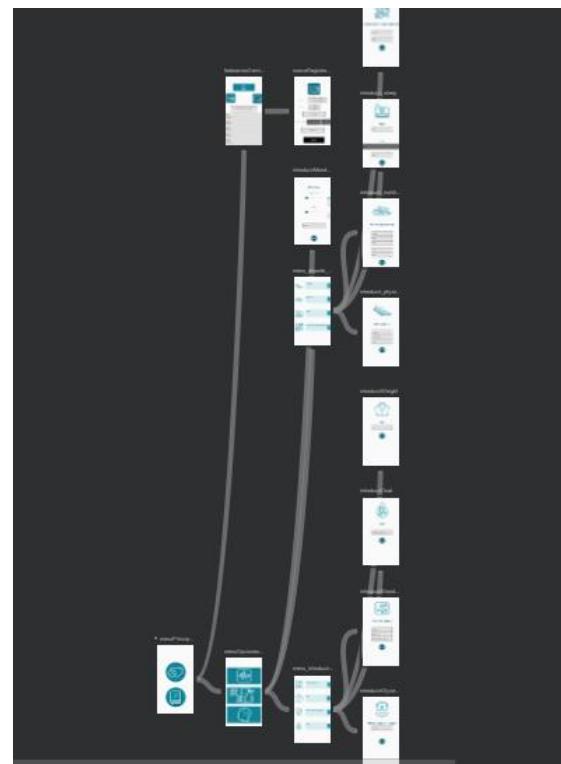
- **Fase 2: Diseño**

Se pedía un diseño algo especificado, que contara con una pantalla de inicio con un login, otra pantalla para registrar más usuarios, los menús correspondientes y las pantallas de configuraciones del servidor y perfil.



- **Fase 3: Desarrollo**

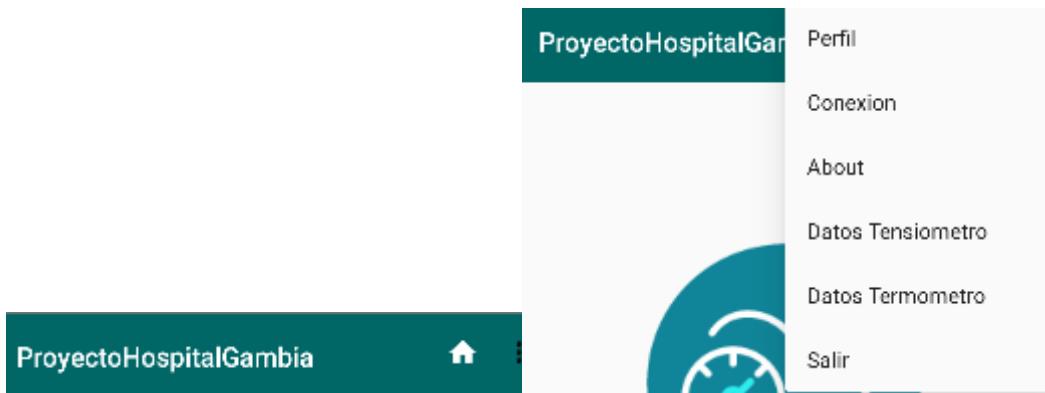
Esta parte se codificó por partes, primero se realizó la parte grafica realizada en el diseño, una vez con la parte grafica se desarrolló la parte de la unión de los fragments y los activitys, los activitys se unieron a través de intents quitando todo tipo de animación, mientras que las actividades se unieron mediante el navigation, se separó en varios activitys para evitar las sobrecargas por lo que se decidió que en fragments solo se quedara la parte de los menus y sus correspondientes formularios para subir la información médica a mano.



Por otra parte, también se programaron todas las seekbars y los spinners que tuvo menos carga de trabajo.



También se desarrolló la implementación de la toolbar o barra de herramientas.



La integración fue constante mientras se iba desarrollando, pero la unión se realizó una vez se tenía todas las pantallas realizadas.

- **Fase 4: Pruebas**

Las principales pruebas que se han realizado en este ciclo han sido las pruebas de integración entre las distintas funcionalidades para que las conexiones fueran las esperadas.

Otro tipo de pruebas que se realizaron fueron pruebas unitarias de ciertas funcionalidades como los spinners o las seekbars.

Otras pruebas que se realizaron fueron pruebas con usuarios reales para probar si la aplicación cumplía con los requisitos pedidos y en caso de sacar algún fallo poder corregirlo.

2^a Ciclo

- **Fase 1: Análisis y Requisitos**

En este segundo ciclo nos enfocaremos en definir los requisitos para la conexión con los dispositivos médicos mediante Bluetooth.

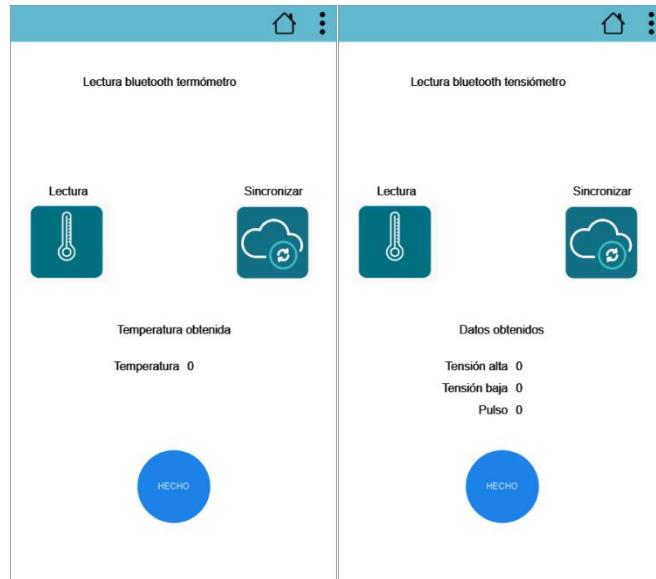
Desde la pantalla designada para cada dispositivo médico previamente configurado, se debe permitir al usuario conectar el dispositivo y recibir la información más reciente registrada, mostrándola en pantalla.

Posteriormente, el usuario podrá decidir si desea subir o no los datos adquiridos. Además, se debe proporcionar al usuario instrucciones claras sobre cómo realizar el escaneo, el cual se hizo por Mac al principio, pero los requisitos cambiaron a hacer por nombre del dispositivo lo cual lo dificultó, así como informar sobre el estado del proceso de transferencia de datos.

Otro requisito complicado fue que funcionase con cualquier API el escaneo Bluetooth que al principio tampoco se valoró.

- **Fase 2: Diseño**

Se pide un diseño donde tenga dos botones, uno para conseguir los datos del aparto y otro para subirlos al servidor, también debe tener los textos correspondientes a los datos que debe mostrar para que el usuario pueda ver los datos adquiridos.



- **Fase 3: Desarrollo**

Se integró esta parte al código existente, incorporando la funcionalidad nueva la cual se desarrolló de manera independiente donde la aplicación hace un escaneo en su área y se queda con el aparato que coincide con su nombre y una vez hecho eso se hace un traspaso de todos los datos donde depende de la Api del dispositivo se usa un método u otro, pero para Apis más viejas hay que usar métodos deprecated. Una vez se tienen los datos se guarda una Pol con sus datos.

- **Fase 4: Pruebas**

Las principales pruebas que se han realizado en este ciclo han sido las pruebas de integración entre las distintas funcionalidades para que los resultados fueran los esperados y los resultados coincidieran con los del aparato.

Otra prueba es que conecte al aparato y que sea claro si hay un error, aunque el error lo puede dar el aparto físico, lo cual no podemos terminar de gestionar.

3^a Ciclo

- **Fase 1: Análisis y Requisitos**

En este tercer ciclo recogeremos los requisitos que debe tener la parte de muestra de los datos en las gráficas.

Los requisitos son sencillos, ya que los datos de cada usuario dependiendo de los datos que quiera observar el usuario se deben mostrar en unas graficas muy visuales, tanto graficas de barras como puntos dependiendo de los datos, para que vea fácilmente sus propios datos.

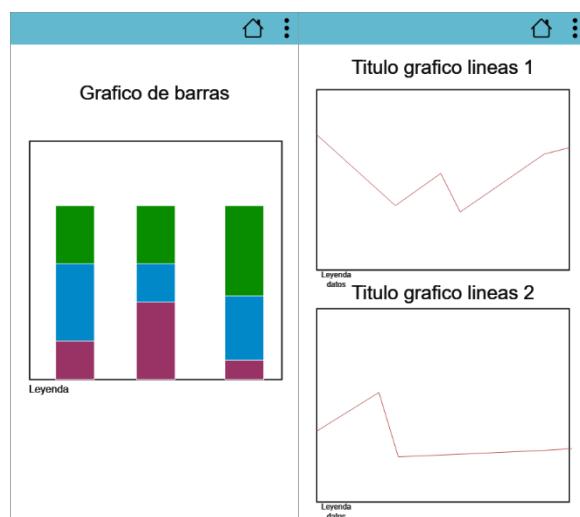
Los datos deben salir de la base de datos en local a través de consultas hacia ella.

También se deben mostrar de manera ordenada por fecha sacada de las Polis con sus datos.

- **Fase 2: Diseño**

El diseño requerido es simple: mostrar la gráfica solicitada junto con un título que describa su contenido.

Consistirá en dos elementos básicos: texto, que informará al usuario sobre el tipo de datos visualizados, y la gráfica o gráficas respectivas. Cada gráfica contendrá una leyenda que permitirá al usuario asociar cada color con su respectivo dato.



- **Fase 3: Desarrollo**

Se integró esta funcionalidad al código existente para permitir la accesibilidad a las gráficas desde diversos menús. La implementación requirió la incorporación de una biblioteca externa que facilitara la creación e integración de gráficas, así como la introducción de datos desde diferentes activitys o fragments.

Para cumplir con esta tarea, optamos por utilizar la biblioteca "MPAndroidChart", ya que las herramientas nativas de Android no ofrecían el nivel de funcionalidad necesario. Al no estar disponible en el repositorio de Maven, añadimos el repositorio "jitpack.io" al archivo Gradle para ampliar la búsqueda y localizar la librería requerida.

La versatilidad de esta librería nos permitió generar distintos tipos de gráficos, desde gráficos de línea, que son los más comunes, hasta gráficos de barras como los utilizados en el contexto de nutrición.

- **Fase 4: Pruebas**

Las principales pruebas que se han realizado en este ciclo han sido las pruebas de integración entre las distintas funcionalidades para que los resultados fueran los esperados y los resultados coincidieran con los introducidos.

4^a Ciclo

- **Fase 1: Análisis y Requisitos**

En este cuarto ciclo nos enfocamos en trabajar con el servidor e intentar subir datos a él. Primero, revisamos los requisitos solicitados y evaluamos su viabilidad. Se planteaba una sincronización entre el servidor y la aplicación, pero nos encontramos con dificultades, ya que el servidor no estaba preparado para estas operaciones. Solo pudimos implementar la funcionalidad de realizar un POST, que fue el único requisito que logramos cumplir.

Posteriormente, intentamos subir solo los datos necesarios del JSON, pero mantener una estructura adecuada para esta operación resultó desafiante y complicó el cumplimiento de los requisitos.

También consideramos copiar la base de datos en local, aunque actualmente solo se utilizan las tablas de People y Pols, dejando el sistema preparado para futuros cambios.

Además, se contempló el uso de Retrofit, pero debido a la complejidad del POST y a los cambios constantes en la URL, optamos por dejarlo preparado para futuras mejoras en el servidor, mientras implementamos el POST de forma directa por el momento.

Otro requisito era subir los usuarios y las Pols a los usuarios creados, pero no fue posible, ya que el servidor encripta las contraseñas de una manera que no pudimos analizar. Por lo tanto, subimos ambas partes por separado, y las Pols se cargan en otro usuario ya creado para poder utilizar sus credenciales al verificar el acceso para realizar el Post.

- **Fase 2: Diseño**

No se implementó ningún diseño específico, ya que se requería para otra funcionalidad relacionada con el acceso para llenar un formulario. Nos centramos únicamente en implementar la funcionalidad necesaria y asegurar su correcto funcionamiento.



Fase 3: Desarrollo

Esta parte se integró fácilmente en el código existente, ya que se implementó de manera independiente. Esto incluyó la funcionalidad de realizar un POST al servidor y utilizar métodos asíncronos para llamar al servidor con las Pols guardadas que aún no se han subido. Esto resulta en una carga visible para el usuario, que luego se envía al servidor. Además, se desarrolló un adaptador para mostrar en la lista las Pols que ya han sido subidas.

- **Fase 4: Pruebas**

Las pruebas principales realizadas en este ciclo han consistido en subir las Pols guardadas al servidor y verificar si se completaba correctamente o si quedaban pendientes de subir.

5º Ciclo

- **Fase 1: Análisis y requisitos**

En el último ciclo, la meta era actualizar los elementos de la interfaz gráfica para darle un aspecto más contemporáneo. Optamos por adoptar el "Material Design" como tema, e implementamos sus elementos siempre que fuera factible.

- **Fase 2: Diseño**

La idea de diseño consistía en renovar la apariencia visual de los elementos y la aplicación en su conjunto, sin alterar la estructura de diseño, es decir, manteniendo los elementos en su ubicación original.

- **Fase 3: Desarrollo**

Para aplicar el tema, inicialmente se agregó al archivo correspondiente (themes.xml), seleccionando los colores primarios y secundarios adecuados para modificar elementos como la barra de notificaciones del teléfono cuando la aplicación está en uso. Dado que solo disponemos del diseño en modo claro, los mismos colores se eligieron tanto para el archivo del modo claro como para el oscuro, garantizando que, independientemente de la preferencia del usuario, la apariencia sea coherente con nuestras elecciones.

```
<resources>
    <!-- Base application theme. -->
    <style name="Base.Theme.ProyectoHospitalGambia" parent="Theme.MaterialComponents.Light.NoActionBar">
        <!-- Customize your light theme here. -->
        <item name="colorPrimary">@color/azul_toolbar</item>
        <item name="colorPrimaryDark">@color/azul_toolbar</item>
        <item name="colorAccent">@color/mi_color_accent</item>
    </style>

    <style name="Theme.ProyectoHospitalGambia" parent="Base.Theme.ProyectoHospitalGambia" />
</resources>
```

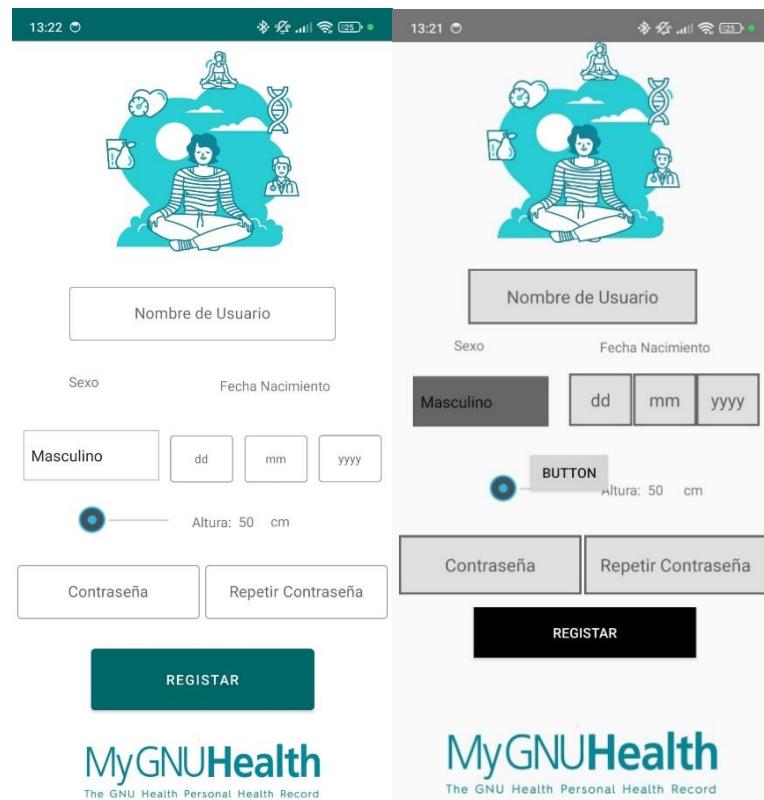
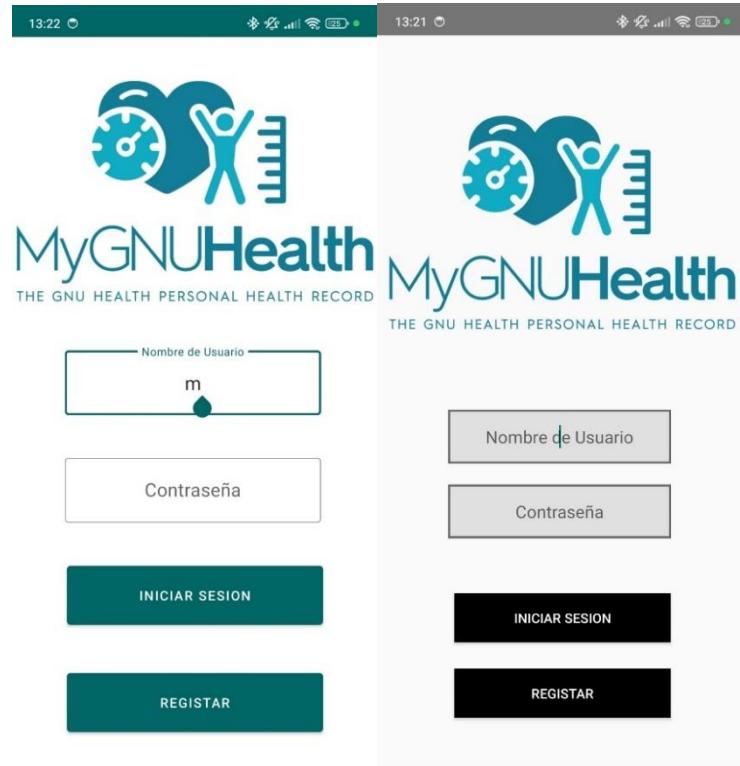
El siguiente paso consistió en actualizar los elementos iniciales de los diseños, reemplazándolos con sus versiones mejoradas del "Material Design". En particular, se sustituyeron todos los "Edit text" por "material.TextInputLayout", que incorpora características como animaciones en los "hint". Además, se ajustaron los tamaños de los elementos y del texto para garantizar una mayor coherencia con el nuevo tema.

```
<com.google.android.material.textfield.TextInputLayout
    android:id="@+id/edt_contraseniaRepetirUsuarioRegistrar_layout"
    style="@style/Widget.MaterialComponents.TextInputLayout.OutlinedBox"
    android:layout_width="188dp"
    android:layout_height="61dp"
    app:layout_constraintBottom_toTopOf="@+id/btn_registrarUsuario"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/edt_contraseniaUsuarioRegistrar_layout"
    app:layout_constraintTop_toBottomOf="@+id/sk_altura"
    android:hint="Repetir Contraseña">

    <com.google.android.material.textfield.TextInputEditText
        android:id="@+id/edt_contraseniaRepetirUsuarioRegistrar"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center"
        android:inputType="textPassword" />

</com.google.android.material.textfield.TextInputLayout>
```

Comparación de la pantalla de Inicio y Registrar, antigua frente a la nueva, donde se puede apreciar el cambio visual:

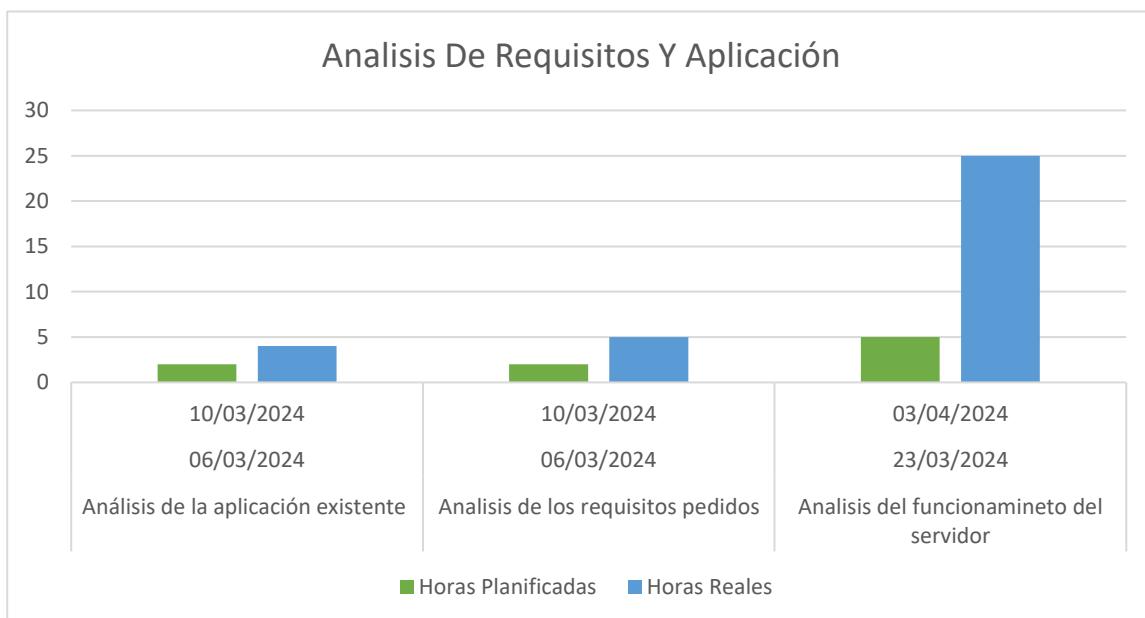


- **Fase 4: Pruebas**

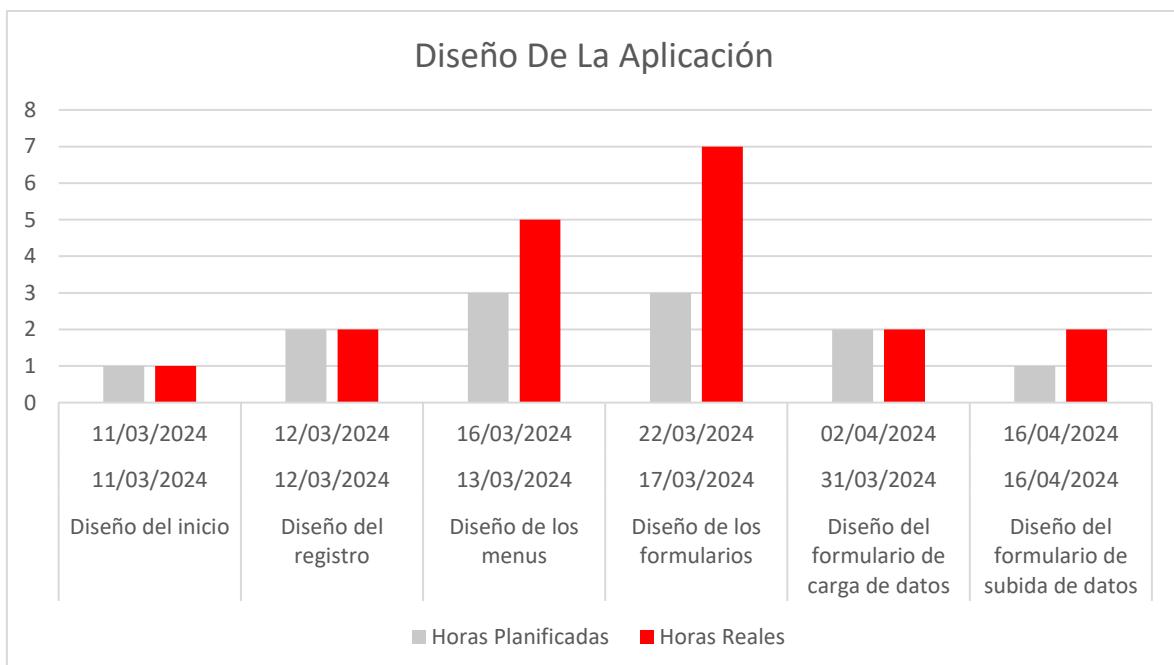
Dado que las modificaciones eran puramente visuales, las pruebas consistieron en utilizar la aplicación de forma habitual y verificar su funcionamiento correcto, asegurándose de que todos los elementos se mostraran según lo esperado.

4.2. Planificación.

ANALISIS DE REQUISITOS Y APLICACIÓN					
Analisis Realizado	Fecha de inicio	Fecha de fin	Horas Planificadas	Horas Reales	Justificación
Análisis de la aplicación existente	06/03/2024	10/03/2024	2	4	Hubo que analizar la aplicación existente para comprender su funcionamiento y flujo, el cual presentaba deficiencias y no facilitaba la captura de los requisitos que debíamos abordar posteriormente.
Analisis de los requisitos pedidos	06/03/2024	10/03/2024	2	5	Una vez analizada la aplicación, se procedió a extraer los requisitos que debía cumplir la nueva aplicación. Sin embargo, este proceso llevó más tiempo del esperado debido a la falta de claridad en el análisis inicial de la aplicación, lo que nos obligó a retroceder en varias ocasiones durante esta etapa.
Analisis del funcionamiento del servidor	23/03/2024	03/04/2024	5	25	El análisis del servidor resultó ser un desafío considerable, dado que no había acceso directo a él y la documentación disponible era escasa. Esto obligó a realizar numerosas pruebas y a realizar conjeturas para comprender su funcionamiento. A parte que también hubo que sacar la estructura la BBDD, lo cual también fue complicado.



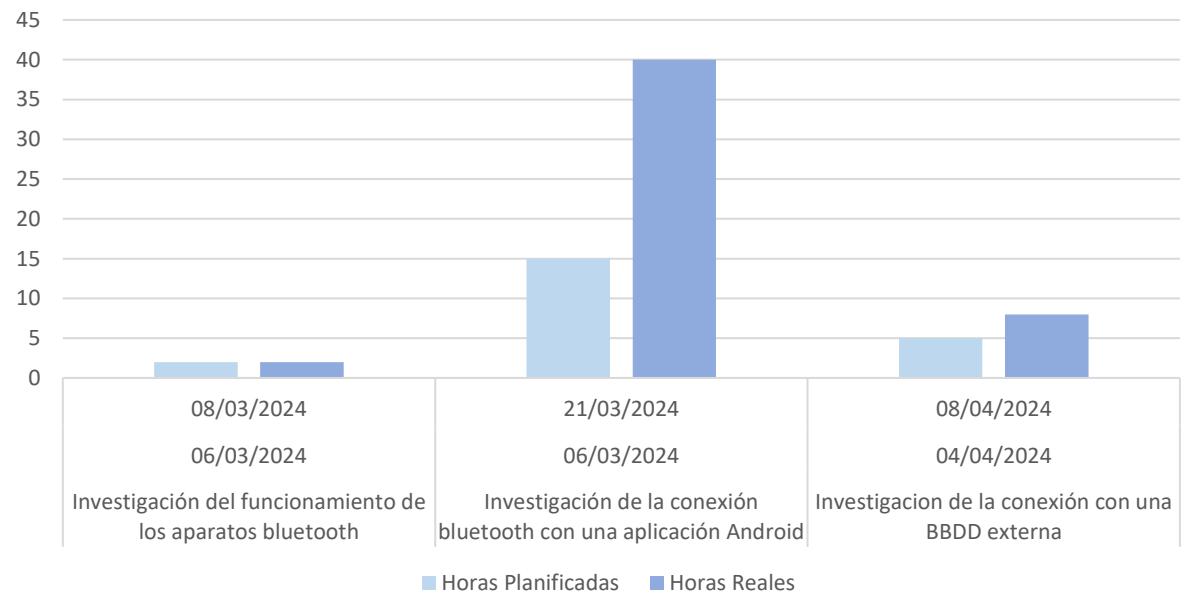
DISEÑO DE LA APLICACIÓN					
Diseño Realizado	Fecha de inicio	Fecha de fin	Horas Planificadas	Horas Reales	Justificación
Diseño del inicio	11/03/2024	11/03/2024	1	1	El diseño fue sencillo, ya que nos basamos en la aplicación existente, la cual no es muy compleja.
Diseño del registro	12/03/2024	12/03/2024	2	2	También fue un diseño sencillo, aunque la complejidad radicó en la integración de numerosos elementos en una sola pantalla.
Diseño de los menús	13/03/2024	16/03/2024	3	5	El diseño de los menús fue sencillo, pero adaptarlos al funcionamiento de la aplicación resultó ser un desafío.
Diseño de los formularios	17/03/2024	22/03/2024	3	7	Estos diseños fueron simples, pero el proceso resultó tedioso y prolongado debido a la gran cantidad de formularios requeridos.
Diseño del formulario de carga de datos	31/03/2024	02/04/2024	2	2	Fueron diseños propios muy simples, ya que requerían muy pocas modificaciones.
Diseño del formulario de subida de datos	16/04/2024	16/04/2024	1	2	Este también fue un diseño basado en la aplicación existente, con algunas modificaciones, y resultó muy sencillo debido a la cantidad limitada de elementos necesarios.



INVESTIGACIÓN DE NUEVAS TECNOLOGÍAS

Investigaciones Realizadas	Fecha de inicio	Fecha de fin	Horas Planificadas	Horas Reales	Justificación
Investigación del funcionamiento de los aparatos bluetooth	06/03/2024	08/03/2024	2	2	Los aparatos médicos fueron fáciles de utilizar debido a su naturaleza intuitiva.
Investigación de la conexión bluetooth con una aplicación Android	06/03/2024	21/03/2024	15	40	Esta investigación fue la más complicada, ya que implicó aprender una tecnología completamente nueva y enfrentó numerosos problemas al conectar y transferir datos.
Investigación de la conexión con una BBDD externa	04/04/2024	08/04/2024	5	8	No fue demasiado complejo aprender a conectar la aplicación con una base de datos externa, dado que existe una cantidad considerable de documentación disponible.

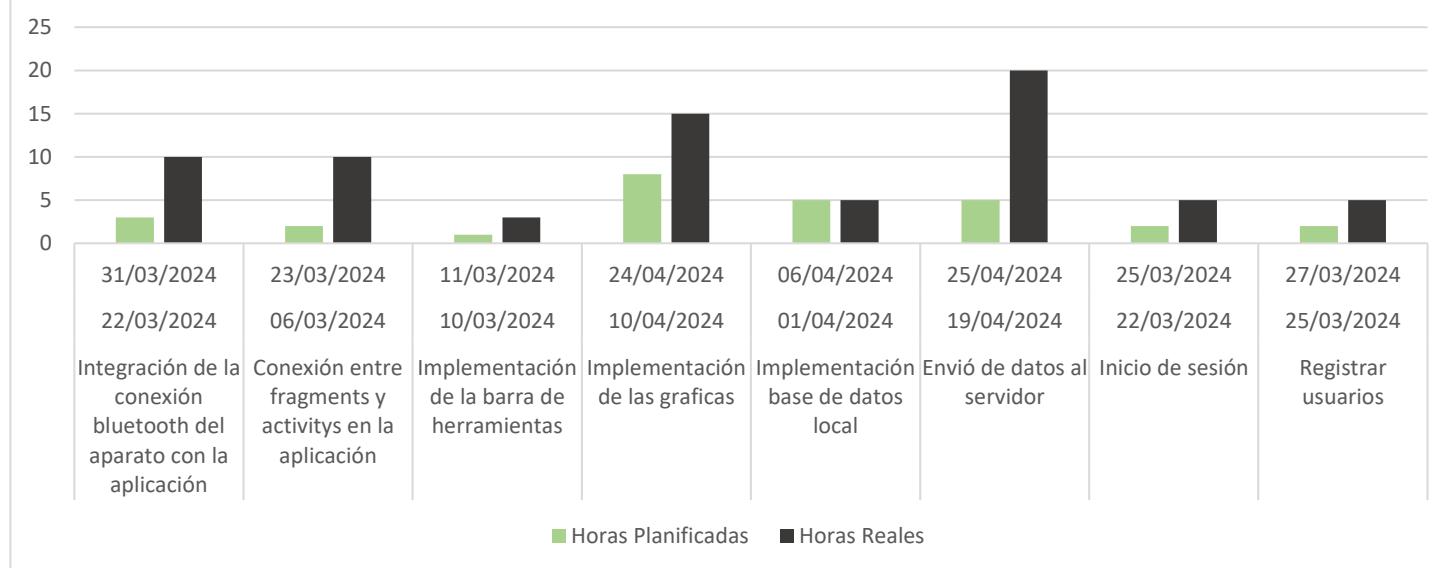
INVESTIGACIÓN DE NUEVAS TECNOLOGÍAS



Sergio García Gómez

INTEGRACIÓN DE NUEVAS FUNCIONALIDADES					
Integraciones Realizadas	Fecha de inicio	Fecha de fin	Horas Planificadas	Horas Reales	Justificación
Integración de la conexión bluetooth del aparato con la aplicación	22/03/2024	31/03/2024	3	10	Hubo problemas inesperados en la integración del código investigado de esta tecnología, pero al final resultó relativamente sencillo, ya que solo era necesario acoplarlo al resultado que necesitábamos.
Conexión entre fragments y activitys en la aplicación	06/03/2024	23/03/2024	2	10	Resultó tedioso debido a la gran cantidad de conexiones entre las pantallas, aunque en realidad la conexión era bastante sencilla. El único aspecto problemático fue la navegación a través de la barra de herramientas.
Implementación de la barra de herramientas	10/03/2024	11/03/2024	1	3	El proceso en sí fue sencillo, pero la incorporación en las pantallas necesarias resultó ser un poco más compleja.
Implementación de las gráficas	10/04/2024	24/04/2024	8	15	La implementación de las gráficas presentó numerosos problemas debido a las incompatibilidades con la biblioteca utilizada, y el manejo de los datos de la base de datos local resultó ser algo complejo.
Implementación base de datos local	01/04/2024	06/04/2024	5	5	La incorporación fue sencilla, pero la creación de la misma base de datos que requerían los requisitos resultó ser algo más compleja, ya que no tenía mucho sentido en ese contexto.
Envío de datos al servidor	19/04/2024	25/04/2024	5	20	Resultó bastante complejo debido a la escasez de documentación y a que los datos obtenidos en el análisis previo no facilitaron la operación. Además, las restricciones en el servidor representaron un gran problema.
Inicio de sesión	22/03/2024	25/03/2024	2	5	Fue una implementación básica y sencilla.
Registrar usuarios	25/03/2024	27/03/2024	2	5	Fue una implementación básica y sencilla.

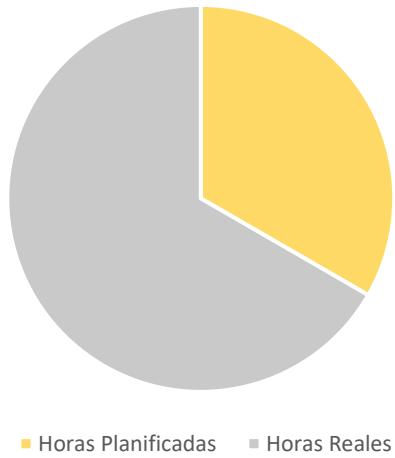
INTEGRACION DE NUEVAS FUNCIONALIDADES



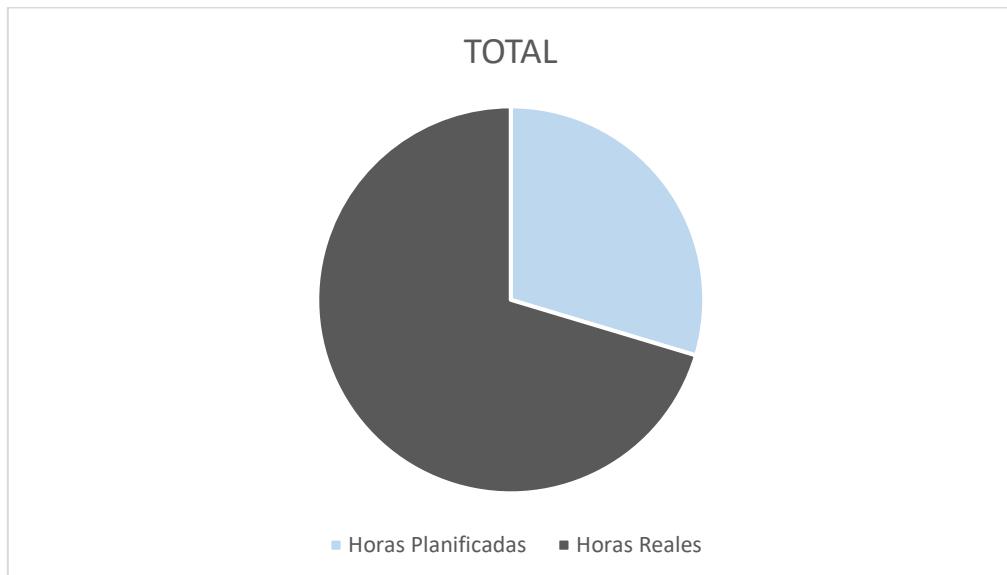
MEMORIA

Memoria Realizada	Fecha de inicio	Fecha de fin	Horas Planificadas	Horas Reales	Justificación
Memoria	06/03/2024	17/05/2024	20	40	La investigación de varios puntos de la misma prolongó su realización.

MEMORIA



TOTAL		
Total	Horas Planificadas	Horas Reales
Total	91	216



5. Diseño del proyecto.

5.1. Diseño de la BBDD.

Diagrama Modelo Relacional BBDD

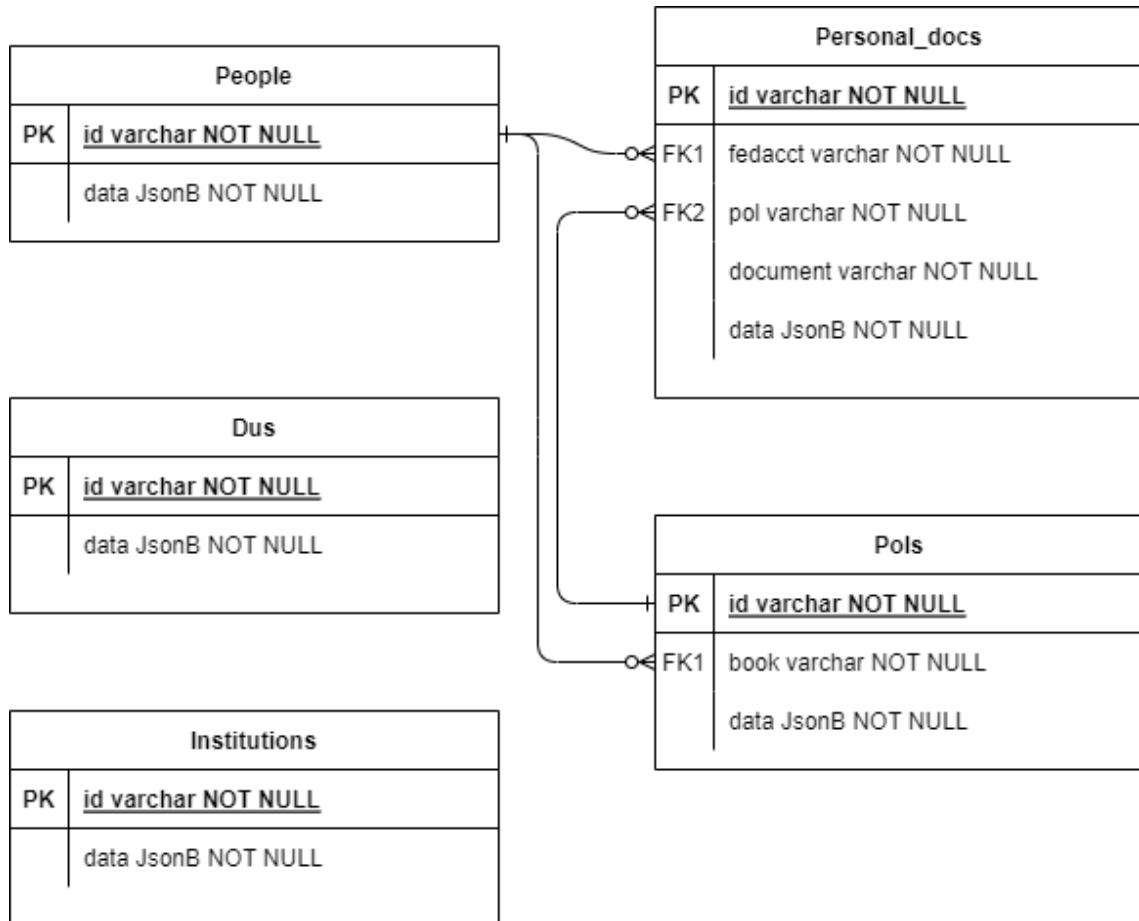
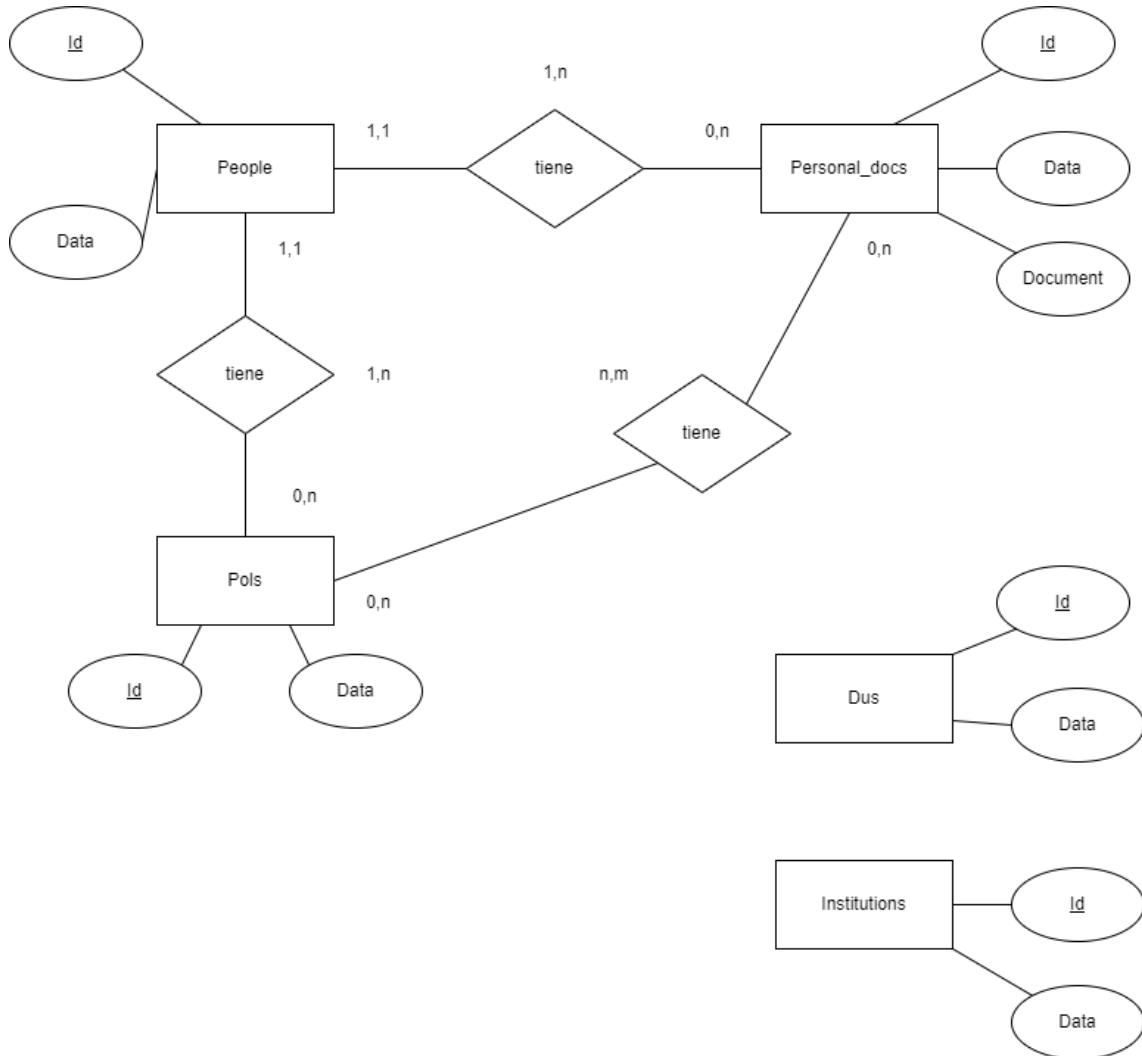


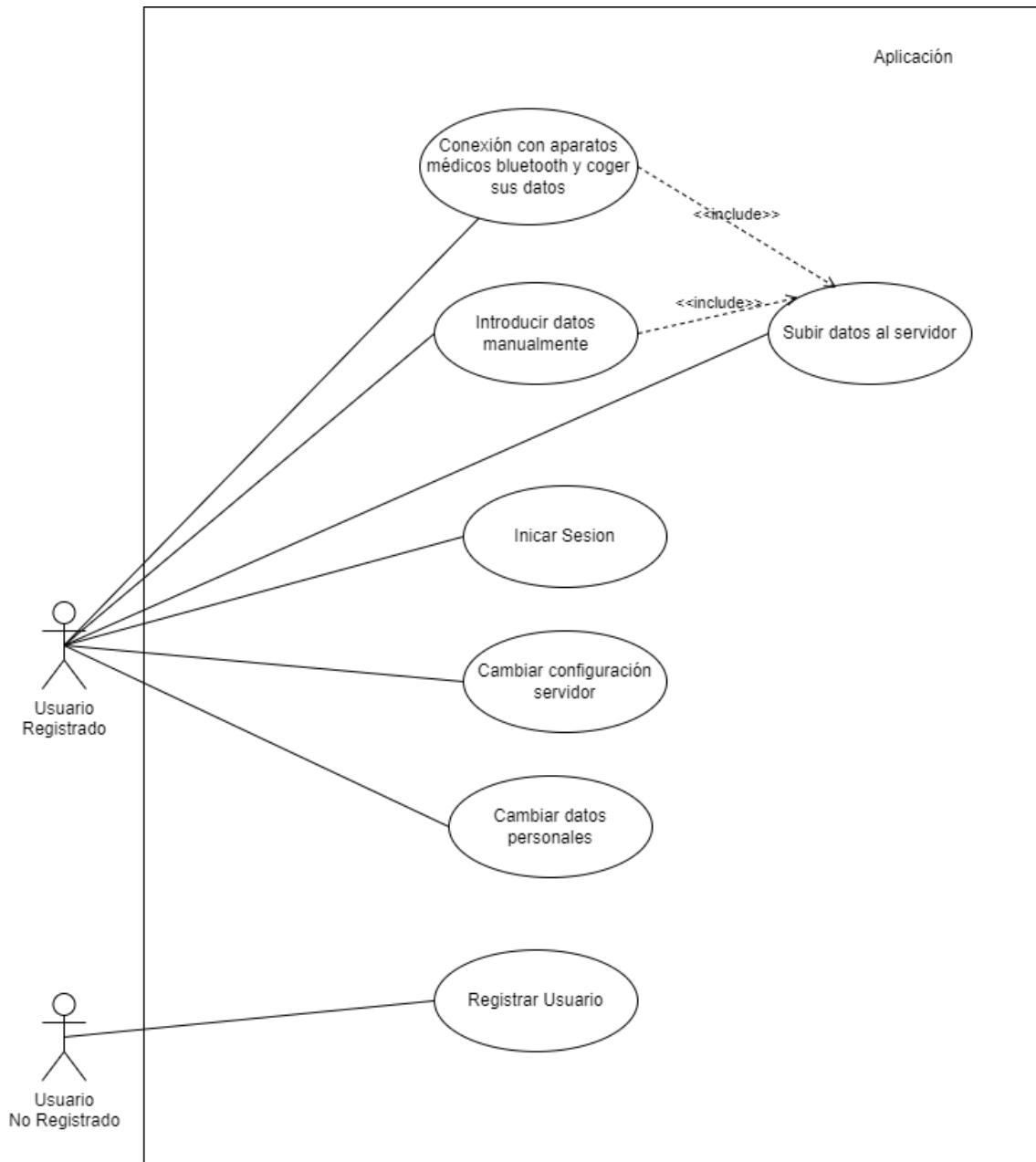
Diagrama Modelo Entidad Relación



5.2. Interacción con el usuario.

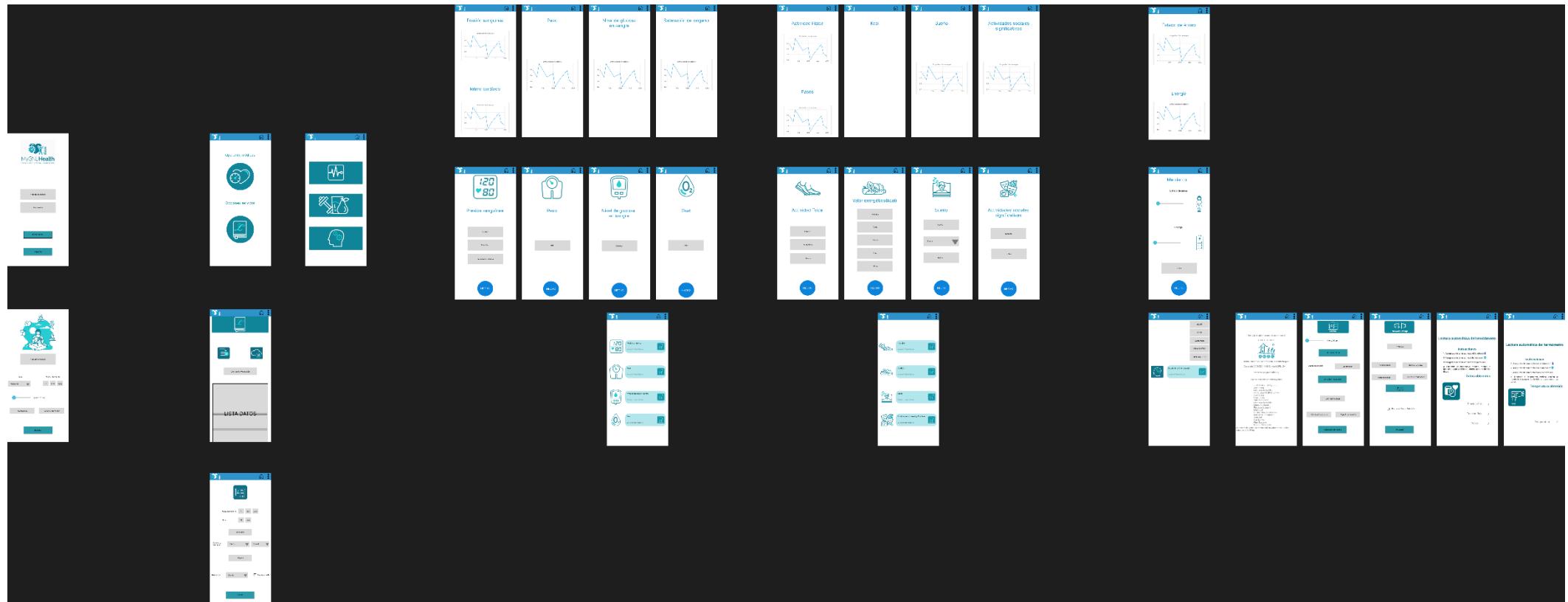
- **Iniciar sesión:**
 - El usuario puede iniciar sesión para acceder a sus datos previamente registrados.
 - Si no tiene una cuenta, el usuario puede registrar una nueva para empezar a utilizar la aplicación.
- **Introducción de datos:**
 - El usuario puede introducir datos médicos relevantes según su situación o necesidad, utilizando las opciones proporcionadas en la interfaz de la aplicación.
- **Conexión con dispositivo medico:**
 - Los usuarios pueden establecer conexión con dispositivos médicos compatibles a través de Bluetooth, utilizando las pantallas dedicadas diseñadas para gestionar la entrada y transferencia de datos desde los dispositivos.
- **Subir datos al servidor:**
 - Todos los datos, tanto los recopilados de dispositivos médicos como los introducidos manualmente, pueden subirse al servidor con botones de acción indicados en la interfaz.
- **Configuración de la aplicación:**
 - Los usuarios tienen la capacidad de configurar su perfil con información personal y detalles relevantes sobre sus condiciones médicas para un uso más personalizado de la aplicación.
 - Además, se proporciona la opción de configurar las conexiones con el servidor, permitiendo al usuario cambiar la configuración de subida del servidor según sea necesario en cualquier momento.

5.2.1. Casos de uso.



5.2.2. Diseño de la interfaz.

Para acceder al Figma con el diseño y la navegación de la aplicación pinche [Aquí](#).



5.2.3. Diseño de la arquitectura

La arquitectura de esta aplicación se basa en el modelo cliente-servidor, un enfoque ampliamente utilizado en el desarrollo de aplicaciones distribuidas. En este modelo, la aplicación se divide en dos partes principales: el cliente y el servidor, cada uno desempeñando un papel específico en el proceso de interacción y comunicación.

Cliente:

El cliente es la parte de la aplicación que interactúa directamente con el usuario final. En el contexto de esta aplicación móvil, el cliente es la interfaz de usuario que se ejecuta en el dispositivo del usuario, como un teléfono inteligente o una tableta. Sus responsabilidades incluyen:

- Recopilación de Entrada del Usuario: El cliente recoge la entrada proporcionada por el usuario.
- Procesamiento de Datos: Procesa la entrada del usuario para realizar acciones relevantes.
- Presentación Visual: Muestra los resultados de manera comprensible y visualmente atractiva.
- Intermediario con el Servidor: Actúa como intermediario entre el usuario y el servidor, enviando solicitudes al servidor y recibiendo respuestas para presentar al usuario.



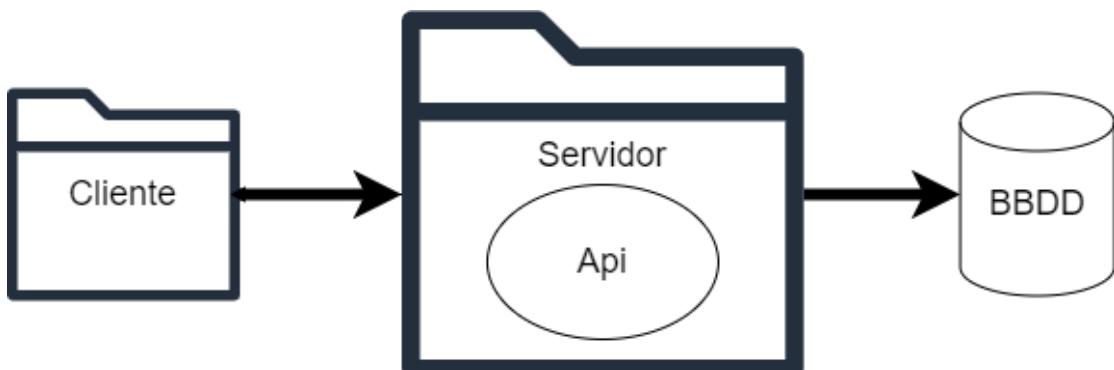
Servidor:

El servidor es la parte de la aplicación que gestiona y procesa los datos de manera centralizada. En el contexto de esta aplicación, el servidor se ejecuta en un sistema remoto y proporciona los servicios necesarios para la aplicación móvil. Sus funciones clave son:

- Almacenamiento y Administración de Datos: El servidor almacena y administra los datos de los usuarios.
- Procesamiento de Solicitudes: Procesa las solicitudes enviadas por el cliente.
- Respuestas a las Solicitudes: Proporciona información solicitada al cliente.



En resumen, el diseño de la arquitectura cliente-servidor permite una separación clara de responsabilidades entre el cliente y el servidor, lo que facilita la escalabilidad, la mantenibilidad y la eficiencia en el desarrollo y funcionamiento de la aplicación. Este enfoque distribuido también permite una mayor flexibilidad en la gestión de datos y recursos, proporcionando una experiencia de usuario más robusta y confiable.



5.2.4. Arquitectura del servidor.

El servidor es externo por lo que no tenemos constancia de cómo es su arquitectura

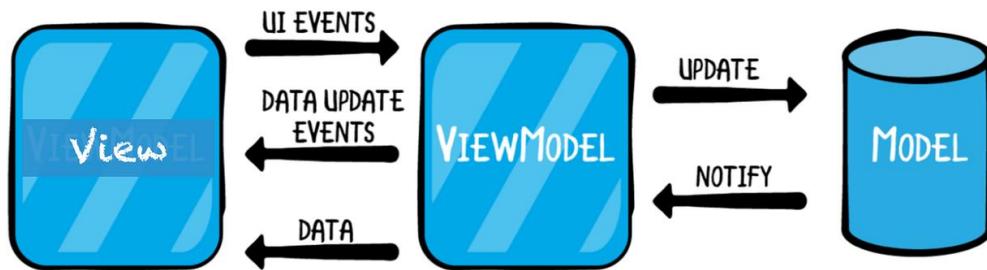
5.2.5. Arquitectura del cliente.

En cuanto a la arquitectura, hemos decidido desarrollar nuestra aplicación en MVVM con Clean Arquitecture, ya que considerábamos que era una muy buena opción.

MVVM (Model-View-ViewModel): El patrón MVVM se utiliza para separar claramente las responsabilidades de la interfaz de usuario, la lógica de presentación y la lógica de negocio. Los componentes Model, View y ViewModel interactúan entre sí para proporcionar una estructura modular y escalable.

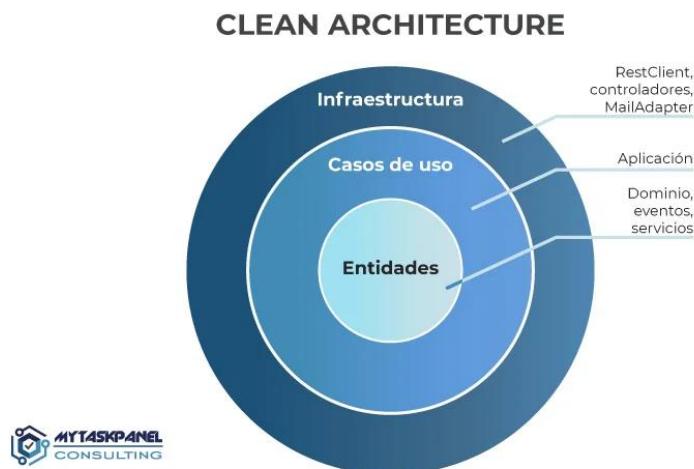
- **Model:** El modelo representa los datos y la lógica empresarial de la aplicación. En el caso de MVVM, el modelo es independiente de la vista y el modelo de vista. El modelo se comunica con el modelo de vista a través de interfaces.
- **View:** La vista es la interfaz de usuario de la aplicación. En el caso de MVVM, la vista es independiente del modelo y el modelo de vista. La vista se comunica con el modelo de vista a través de enlaces de datos.

- **ViewModel:** El modelo de vista es el intermediario entre la vista y el modelo. Se encarga de la lógica de presentación y se comunica con el modelo a través de interfaces. El modelo de vista también se comunica con la vista a través de enlaces de datos.



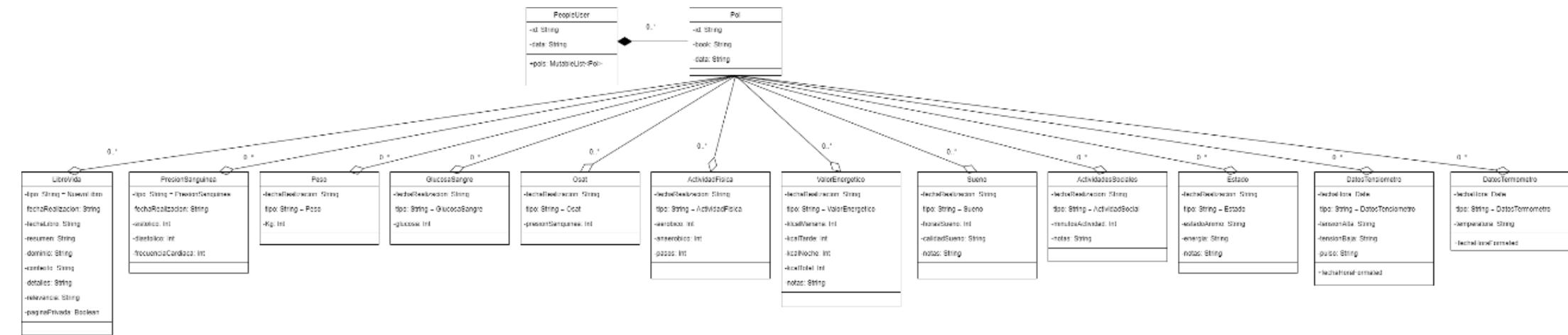
Arquitectura Clean: La arquitectura Clean es una arquitectura de software que se basa en la separación de capas y la dependencia en una sola dirección.

- **Separación de capas:** La arquitectura Clean se basa en la separación de capas. Cada capa tiene una responsabilidad única y no depende de las capas superiores o inferiores. Las capas se comunican entre sí a través de interfaces.
- **Dependencia en una sola dirección:** La arquitectura Clean se basa en la dependencia en una sola dirección. Las capas superiores no dependen de las capas inferiores. Las capas inferiores proporcionan interfaces que las capas superiores pueden utilizar.



Algo que se debe aclarar es que la arquitectura principal es la MVVM ya que la arquitectura Clean se puede aplicar a cualquier tipo de estructura, pero no es un orden como tal.

El diagrama de clases uml sería este:



Puedes acceder a la estructura de carpetas en este enlace [Aquí](#).

La documentación KDoc del cliente se encuentra en el siguiente enlace [Aquí](#), pero es necesario tener Java 17 si vas a abrir el proyecto en local para que pueda funcionar.

5.2.6. Relación con otros sistemas.

Aparato Médico

Hay otros sistemas relacionados, pero son sistemas que no podemos controlar, ya que son los de los aparatos médicos facilitados, que la única gestión que hacen son el cálculo de los datos médicos necesarios, me parece el tensiómetro que saca la tensión alta, la tensión baja y el pulso. A parte de lo anterior lo que hacen es mandar los datos vía bluetooth a mi aplicación, la cual los recibe y hace sus funciones con ellos.

Cuando hace una transmisión de datos vía bluetooth a través de un escaneo por nombre y tiene muchos registros no pueden enviar todos los registros necesarios, indica un error 7 que en la transmisión de datos como indica su manual, ya que tampoco pueden enviar solo el último, por lo que necesitamos borrar los registros de datos anteriores.

Algo importante para tener en cuenta es que cada vez que el aparato médico se conecta a un nuevo dispositivo, se debe ingresar el código proporcionado por el aparato médico para permitir que el nuevo dispositivo se vincule con él por primera vez.

En conclusión, este sistema sería mejorable comprando o adquiriendo otro aparato con otro sistema que si sea capaz de realizar esta tarea igual, pero evitando esos problemas.

6. Detalles de implementación.

Implementación Bluetooth

Uno de los detalles más importantes es que para poder usar el escaneo de aparatos Bluetooth en todos los dispositivos necesitas controlar la API del móvil o tableta que lo está utilizando porque a partir de la API 31 se hace de la forma actual sin ningún método deprecated pero para poder usarlo en los dispositivos con un API anterior hay que usar este método deprecated:

```
override fun onCharacteristicChanged(  
    gatt: BluetoothGatt,  
    characteristic: BluetoothGattCharacteristic  
) {  
    // Handle the characteristic change  
    Log.d( tag: "Bluetooth2", msg: "viaV")  
    if (characteristic?.uuid == UUID.fromString( name: "00002a5e-0000-1000-8000-00805f9b34fb")) {  
        // Obtener los datos de la característica  
        val data = characteristic.value  
        if (data != null && data.size >= 4) {
```

También es importante controlar los permisos en función de la API porque antes de la API 31 hay permisos que no existían como son estos:

```
Manifest.permission.BLUETOOTH_SCAN,  
Manifest.permission.ACCESS_FINE_LOCATION,  
Manifest.permission.BLUETOOTH_CONNECT
```

Por lo que hay que idear este método para que no de error por el tema de los permisos:

```
if (Build.VERSION.SDK_INT > Build.VERSION_CODES.S) {  
    if (!hasBluetoothPermissions || !hasLocationPermissions || !hasBluetoothScanPermissions || !hasBluetoothConnectPermissions)  
        Log.d( tag: "Bluetooth2", msg: "Permisos denegados")  
    ActivityCompat.requestPermissions(  
        activity: this,  
        arrayOf(  
            Manifest.permission.BLUETOOTH,  
            Manifest.permission.BLUETOOTH_ADMIN,  
            Manifest.permission.BLUETOOTH_SCAN,  
            Manifest.permission.ACCESS_FINE_LOCATION,  
            Manifest.permission.BLUETOOTH_CONNECT  
, requestCode: 1  
)  
    return@setOnClickListener  
}  
} else {  
    if (!hasBluetoothPermissions) {  
        Log.d( tag: "Bluetooth2", msg: "Permisos denegados")  
        ActivityCompat.requestPermissions(  
            activity: this,  
            arrayOf(  
                Manifest.permission.BLUETOOTH,  
                Manifest.permission.BLUETOOTH_ADMIN  
, requestCode: 1  
)  
    return@setOnClickListener  
}
```

Algo que se debe saber es que esta operación se puede hacer de dos formas, una es escaneando los aparatos Bluetooth que hay alrededor y cogiendo el de nuestro aparato, o haciéndolo vía Mac, para nivel de usuario es mejor la segunda opción y es la que esta implementada, por petición, pero hemos comprobado que por Mac es más seguro ya que hace una conexión directa.

Trato y carga de datos de las conexiones Bluetooth

A nivel técnico, los datos devueltos por el dispositivo se almacenan en una lista y se selecciona el dato deseado para su visualización.

```
// Bucle forEach para recorrer todos los registros y encontrar el más reciente
var ultimoDatoTensiometro: DatosTensiometro? = null
datosTensiometroList.forEach { record ->
    if (ultimoDatoTensiometro == null || record.fechaHora > ultimoDatoTensiometro!!.fechaHora) {
        ultimoDatoTensiometro = record
    }
}
```

Una parte importante es la corutina que consigue que se carguen todos los datos y de tiempo a que solo saque el último dato requerido mostrando una barra de progreso.

```
// Mostrar la ProgressBar y quitar el boton
progressBar.visibility = View.VISIBLE
btMedicion.visibility = View.INVISIBLE

// Iniciar la corutina para simular el proceso de carga
viewModelJob = CoroutineScope(Dispatchers.Main).launch { this:CoroutineScope
    // Simular el progreso de carga durante 3 segundos
    for (progress in 0 .. 100) {
        progressBar.progress = progress
        delay( timeMillis: 300) // Cambia este valor para ajustar la velocidad de llenado de la ProgressBar
    }

    // Ocultar la ProgressBar
    progressBar.visibility = View.INVISIBLE
    btMedicion.visibility = View.VISIBLE

    cargarRegistros()
}
```

La parte más importante de este desarrollo es el tratamiento de datos en este caso para el termómetro, donde necesitamos la documentación del fabricante donde nos indique que bits son los que dan cada dato.

```
// Interpretar los datos recibidos
val tensionBaja = data[3]

val pulse = data[14]

val tensionAlta = data.sliceArray( indices: 1 ≤ until < 2).map { it.toInt() and 0xFF }
    .fold( initial: 0) { acc, byte ->
        (acc shl 8) + byte
    }

val bytesYear =
    data.sliceArray( indices: 7 ≤ until < 14) // Obtener los 7 bytes para la fecha y hora

val yearBuffer = ByteBuffer.wrap(bytesYear.copyOfRange(0, 2)).order(
    ByteOrder.LITTLE_ENDIAN
)
val year = yearBuffer.short.toInt()
```

```
val calendar = Calendar.getInstance()
calendar.set(Calendar.YEAR, year)
calendar.set(Calendar.MONTH, bytesYear[2].toInt() - 1)
calendar.set(Calendar.DAY_OF_MONTH, bytesYear[3].toInt())
calendar.set(Calendar.HOUR_OF_DAY, bytesYear[4].toInt())
calendar.set(Calendar.MINUTE, bytesYear[5].toInt())
calendar.set(Calendar.SECOND, bytesYear[6].toInt())

val fechaHora = calendar.time

val sdf = SimpleDateFormat(pattern: "dd/MM/yyyy HH:mm:ss", Locale.getDefault())
val fechaFormatada = sdf.format(fechaHora)
// Crear un objeto DataRecord con los datos recibidos
val datoNuevoTensiometro = DatosTensiometro(fechaHora, tensionAlta, tensionBaja, pulse)

// Si no existe, agregar un nuevo registro a la lista
datosTensiometroList.add(datoNuevoTensiometro)
```

Algo importante es la creación de un objeto para guardar los cuatro datos necesarios del tensiómetro.

```
└ sgg
data class DatosTensiometro (
    val fechaHora: Date,
    val tensionAlta: Int,
    val tensionBaja: Byte,
    val pulso: Byte
) {
    └ sgg
    val fechaHoraFormatted: String
        get() {
            val sdf = SimpleDateFormat(pattern: "dd/MM/yyyy HH:mm:ss", Locale.getDefault())
            return sdf.format(fechaHora)
        }
}
```

Escaneo de conexiones Bluetooth

Este método sería el que inicia el escaneo de los aparatos cercanos en caso de no tener los permisos correspondientes y haberlos aceptado posteriormente, lo cual iniciara el flujo.

```
override fun onRequestPermissionsResult(
    requestCode: Int,
    permissions: Array<out String>,
    grantResults: IntArray
) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults)
    // Verificar si la solicitud de permisos es para BLUETOOTH
    if (requestCode == 1 && grantResults.isNotEmpty() && grantResults.all { it == PackageManager.PERMISSION_GRANTED }) {
        // Si los permisos fueron concedidos, iniciar la búsqueda de dispositivos
        val deviceName = "BC54"
        val context: Context = this
        startDeviceSearch(context, deviceName)
    }
}
```

Con este método iniciamos el escaneo de los aparatos cercanos al móvil para iniciar la acción, que provoca la llamada al segundo método el cual hace una búsqueda de los aparatos Bluetooth cercanos y devuelve un resultado.

```
private fun startDeviceSearch(context: Context, deviceName: String) {  
    val bluetoothManager =  
        context.getSystemService(Context.BLUETOOTH_SERVICE) as BluetoothManager  
    bluetoothAdapter = bluetoothManager.adapter  
    bluetoothLeScanner = bluetoothAdapter.bluetoothLeScanner  
  
    if (bluetoothAdapter == null || !bluetoothAdapter.isEnabled) {  
        Log.d(tag: "Bluetooth2", msg: "Bluetooth is not enabled")  
        return  
    }  
  
    if (ContextCompat.checkSelfPermission(  
        context,  
        Manifest.permission.BLUETOOTH  
    ) != PackageManager.PERMISSION_GRANTED  
    || ContextCompat.checkSelfPermission(  
        context,  
        Manifest.permission.BLUETOOTH_ADMIN  
    ) != PackageManager.PERMISSION_GRANTED  
    ) {  
        ActivityCompat.requestPermissions(  
            context as Activity,  
            arrayOf(  
                Manifest.permission.BLUETOOTH,  
                Manifest.permission.BLUETOOTH_ADMIN  
            ), requestCode: 1  
        )  
    }  
}
```

```
// Llamar a scanLeDevice dentro de un bloque coroutine  
CoroutineScope(Dispatchers.Main).launch { this: CoroutineScope  
    val devices = scanLeDevice()  
    if (devices.isNotEmpty()) {  
        connectToDevice(devices[0])  
    } else {  
        Log.d(tag: "Bluetooth2", msg: "Device not found")  
    }  
}
```

```
private suspend fun scanLeDevice(): List<BluetoothDevice> {
    val devicesFound = mutableListOf<BluetoothDevice>()
    withContext(Dispatchers.IO) { thisCoroutineScope
        if (ContextCompat.checkSelfPermission(
            context,
            Manifest.permission.BLUETOOTH
        ) != PackageManager.PERMISSION_GRANTED
        || ContextCompat.checkSelfPermission(
            context,
            Manifest.permission.BLUETOOTH_ADMIN
        ) != PackageManager.PERMISSION_GRANTED
    ) {
        ActivityCompat.requestPermissions(
            context,
            arrayOf(
                Manifest.permission.BLUETOOTH,
                Manifest.permission.BLUETOOTH_ADMIN
            ), requestCode: 1
        )
    }
    if (!scanning) {
        bluetoothLeScanner.startScan(leScanCallback)
        scanning = true
        delay(SCAN_PERIOD)
        bluetoothLeScanner.stopScan(leScanCallback)
        scanning = false
        devicesFound.addAll(listadispositivos)
        listadispositivos.clear() // Limpiar la lista después de escanear
    }
}
return devicesFound
```

Este método te devuelve el dispositivo que requerimos por el nombre y lo devuelve en caso afirmativo para que el método anterior se quede con los dispositivos válidos.

```
private val leScanCallback = object : ScanCallback() {
    override fun onScanResult(callbackType: Int, result: ScanResult) {
        super.onScanResult(callbackType, result)
        if (ContextCompat.checkSelfPermission(
            context,
            Manifest.permission.BLUETOOTH
        ) != PackageManager.PERMISSION_GRANTED
        || ContextCompat.checkSelfPermission(
            context,
            Manifest.permission.BLUETOOTH_ADMIN
        ) != PackageManager.PERMISSION_GRANTED
        ) {
            ActivityCompat.requestPermissions(
                context,
                arrayOf(
                    Manifest.permission.BLUETOOTH,
                    Manifest.permission.BLUETOOTH_ADMIN
                ), requestCode: 1
            )
        }
        Log.d("tag: Bluetooth2", "msg: " + "Device found: ${result.device.name}")
        if(result.device.name != null && result.device.name.equals(deviceName)) {
            listadispositivos.add(result.device)
        }
    }
}
```

Gracias a este método conseguimos sacar los servicios de los aparatos Bluetooth para posteriormente poder analizarlos y tomar el que nos corresponda en ese momento.

```
private fun connectToDevice(device: BluetoothDevice) {

    if (checkSelfPermission(Manifest.permission.BLUETOOTH) != PackageManager.PERMISSION_GRANTED ||
        checkSelfPermission(Manifest.permission.BLUETOOTH_ADMIN) != PackageManager.PERMISSION_GRANTED
    ) {
        ActivityCompat.requestPermissions(
            activity: this,
            arrayOf(Manifest.permission.BLUETOOTH, Manifest.permission.BLUETOOTH_ADMIN),
            bluetoothPermissionRequestCode
        )
        return
    }
}
```

```
override fun onConnectionStateChange(gatt: BluetoothGatt?, status: Int, newState: Int) {
    if (newState == BluetoothProfile.STATE_CONNECTED) {
        if (ContextCompat.checkSelfPermission(
            context,
            Manifest.permission.BLUETOOTH
        ) != PackageManager.PERMISSION_GRANTED
        || ContextCompat.checkSelfPermission(
            context,
            Manifest.permission.BLUETOOTH_ADMIN
        ) != PackageManager.PERMISSION_GRANTED
    ) {
        ActivityCompat.requestPermissions(
            context as Activity,
            arrayOf(
                Manifest.permission.BLUETOOTH,
                Manifest.permission.BLUETOOTH_ADMIN
            ), requestCode: 1
        )
    }
    Log.d( tag: "Bluetooth2", msg: "Connected to ${device.name}")
    gatt?.discoverServices()
} else if (newState == BluetoothProfile.STATE_DISCONNECTED) {
    Log.d( tag: "Bluetooth2", msg: "Disconnected from ${device.name}")
}
}
```

También es necesario que se suscriba antes de iniciar el escaneo para para poder recorrer todos los servicios y quedarnos con el que nos interesa para después poder enviar los datos del servicio a nuestra aplicación.

```
override fun onServicesDiscovered(gatt: BluetoothGatt?, status: Int) {
    super.onServicesDiscovered(gatt, status)
    if (ContextCompat.checkSelfPermission(
        context,
        Manifest.permission.BLUETOOTH
    ) != PackageManager.PERMISSION_GRANTED
    || ContextCompat.checkSelfPermission(
        context,
        Manifest.permission.BLUETOOTH_ADMIN
    ) != PackageManager.PERMISSION_GRANTED
) {
    ActivityCompat.requestPermissions(
        context as Activity,
        arrayOf(
            Manifest.permission.BLUETOOTH,
            Manifest.permission.BLUETOOTH_ADMIN
        ), requestCode: 1
    )
}
Log.d( tag: "Bluetooth2", msg: "Services discovered")
```

```
    val characteristic = gatt?.getService(servicio)?.getCharacteristic(caracteristica)
    Log.d( tag: "Bluetooth2", characteristic?.uuid.toString())
    gatt?.setCharacteristicNotification(characteristic, enable: true)
    gatt?.readCharacteristic(characteristic)
    characteristic?.let { it: BluetoothGattCharacteristic
        lifecycleState = BLELifecycleState.ConnectedSubscribing
        subscribeToIndications(it, gatt)
    } ?: run { this:<no name provided>
        lifecycleState = BLELifecycleState.Connected
    }
}
```

Una vez se suscribe analizando la API del dispositivo ya pasara al método de la trata de los datos que realizamos en otra de las partes explicadas para que sean legibles los datos.

```
private fun subscribeToIndications(
    characteristic: BluetoothGattCharacteristic,
    gatt: BluetoothGatt
) {
    if (ContextCompat.checkSelfPermission(
        context,
        Manifest.permission.BLUETOOTH
    ) != PackageManager.PERMISSION_GRANTED
        || ContextCompat.checkSelfPermission(
            context,
            Manifest.permission.BLUETOOTH_ADMIN
        ) != PackageManager.PERMISSION_GRANTED
    ) {
        ActivityCompat.requestPermissions(
            context as Activity,
            arrayOf(
                Manifest.permission.BLUETOOTH,
                Manifest.permission.BLUETOOTH_ADMIN
            ), requestCode: 1
        )
    }
}
```

```
val cccdUuid = UUID.fromString(CCC_DESCRIPTOR_UUID)
characteristic.getDescriptor(cccdUuid)?.let { cccDescriptor ->
    if (!gatt.setCharacteristicNotification(characteristic, enable: true)) {
        return
    }
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
        gatt.writeDescriptor(cccDescriptor, BluetoothGattDescriptor.ENABLE_INDICATION_VALUE) ^let
    } else {
        cccDescriptor.value = BluetoothGattDescriptor.ENABLE_INDICATION_VALUE
        gatt.writeDescriptor(cccDescriptor) ^let
    }
}
```

Implementación Gráficas

Lo más importante para la implementación de las gráficas y que más problemas puede dar es la implementación de la librería ya que en este caso la que mejor nos venia y más fácil no ha parecido de implementar ha sido la librería de MPAndroidChart pero para que funcione hay que seguir estos pasos:

Primero en  build.gradle.kts (Module :app) hay que implementar esta librería:

```
implementation ("com.github.PhilJay:MPAndroidChart:v3.1.0")
```

Pero lo más importante y que más problemas ha dado es que en

 build.gradle.kts (Project: ProyectoHospitalGambia)

hay que hacer esto para que baje la librería y se inserte de forma correcta

```
// Top-level build file where you can add configuration options common to all sub-projects/modules.
plugins { this:PluginDependenciesSpecScope
    id("com.android.application") version "8.1.1" apply false
    id("org.jetbrains.kotlin.android") version "1.9.0" apply false
    id("androidx.navigation.safeargs.kotlin") version "2.7.1" apply false
}

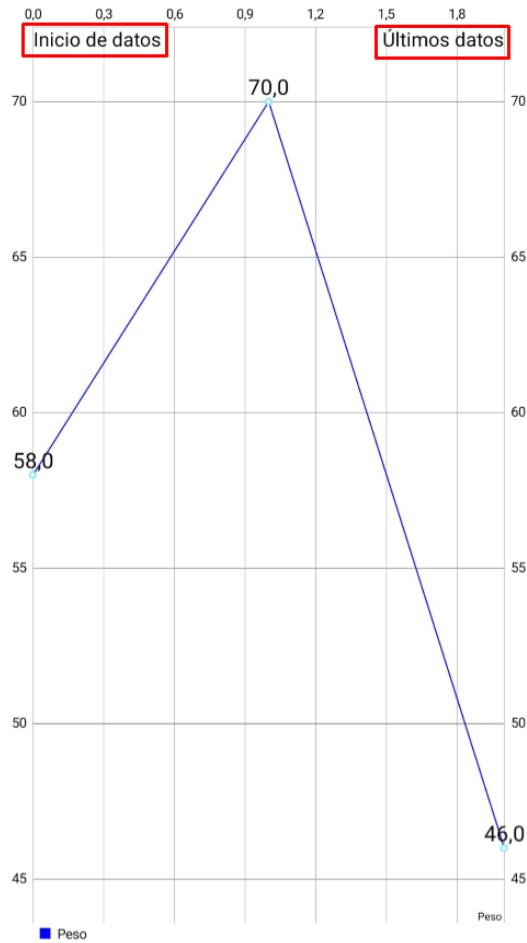
buildscript { this:ScriptHandlerScope
    repositories { this:RepositoryHandler
        google()
        mavenCentral()
        maven { url = uri("https://jitpack.io") }
        maven { url = uri("https://maven.google.com") }
    }
    dependencies { this:DependencyHandlerScope
        classpath ("com.android.tools.build:gradle:7.0.4")
        classpath ("org.jetbrains.kotlin:kotlin-gradle-plugin:1.6.10")
        classpath ("com.google.gms:google-services:4.3.10")
    }
}
```

Una vez completado este paso, puedes integrar la librería de la misma manera que lo haces con otras.

Una vez instalada, puedes utilizar sus componentes en los diseños de la interfaz. Necesitábamos utilizar los elementos de línea para todas las gráficas, excepto para la de nutrición, que era una gráfica de barras. Sin embargo, encontramos algunos problemas al mostrar la fecha en el eje x de las gráficas. Para solucionar esto, creamos una clase personalizada que hereda de la clase que hemos implementado anteriormente.

```
class CustomLineChart(context: Context, attrs: AttributeSet) : LineChart(context, attrs) {  
  
    private val textPaint = Paint().apply { this: Paint  
        color = Color.BLACK  
        textSize = 40f  
        textAlign = Paint.Align.LEFT  
    }  
  
    override fun onDraw(canvas: Canvas) {  
        super.onDraw(canvas)  
  
        val y = viewPortHandler.contentTop() + 40f  
  
        // Dibujar "Inicio de datos" a la izquierda  
        val xInicio = viewPortHandler.contentLeft()  
        canvas.drawText( text: "Inicio de datos", xInicio, y, textPaint)  
  
        // Dibujar "Últimos datos" a la derecha  
        val xFinal = viewPortHandler.contentRight() - textPaint.measureText( text: "Últimos datos")  
        canvas.drawText( text: "Últimos datos", xFinal, y, textPaint)  
    }  
}
```

Esto nos permite mostrar el texto "Inicio de datos" en la parte superior izquierda de la gráfica, y "Fin de datos" en la parte superior derecha.



En los diseños, empleamos esta clase personalizada en lugar de la original, lo que nos permite aprovechar todas sus funciones ya que hereda de ella y nos ofrece el mismo conjunto de características prácticas.

```
<com.example.proyectohospitalgambia.app.utils CustomLineChart  
    android:id="@+id/graficoLineas_Suenno"  
    android:layout_width="0dp"  
    android:layout_height="0dp"  
    android:layout_weight="1"  
    android:padding="16dp"  
    app:layout_constraintBottom_toBottomOf="parent"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toBottomOf="@+id/txt_suenno" />
```

Tomaré como ejemplo la gráfica de Peso para explicar su funcionamiento. En estas gráficas, se muestran exclusivamente los datos del usuario activo en ese momento. Por lo tanto, el primer paso en la clase será identificar qué usuario tiene la sesión iniciada, recuperar todos los datos del tipo de gráfica específica y almacenarlos en una lista.

```
val chartOsat = view.findViewById<LineChart>(R.id.graficoLineas_SaturacionOxigeno)

val idUsuarioActual = MainActivity.usuario?.id
```

Una vez obtenemos la lista con los datos, los almacenamos en el tipo "Entry", el cual es utilizado por la librería de gráficas para representar los datos posteriormente.

```
// Crear las entradas de la gráfica a partir de los datos de osat
val entriesOsat = datosOsat?.mapIndexed { index, osat ->
    Entry(index.toFloat(), osat.presionSanguinea.toFloat())
}
```

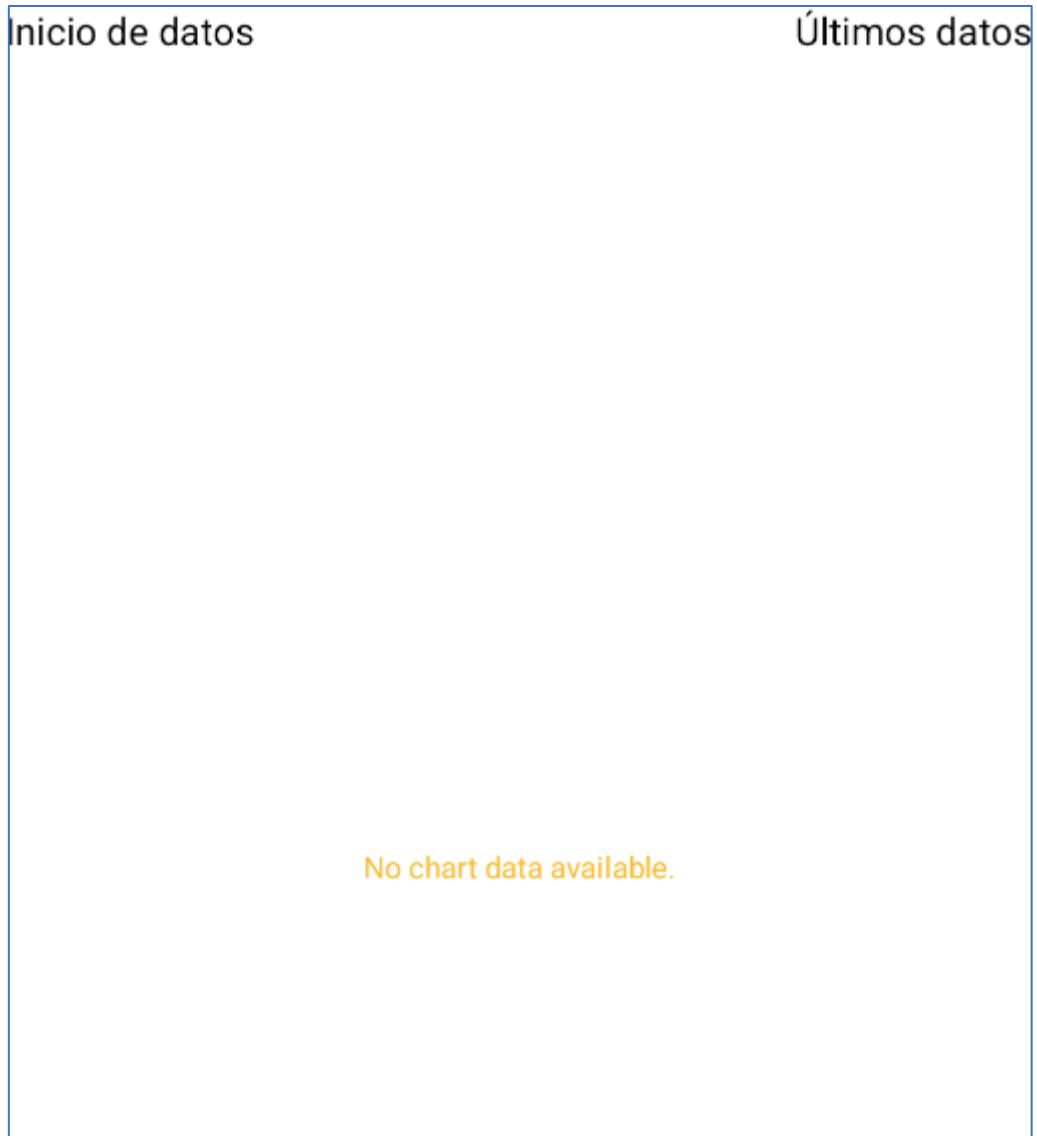
La variable entriesPeso es una lista de objetos "Entry", cada uno de ellos contiene un par de datos: su valor en el eje x y su valor en el eje y.

Después de obtener la lista con los datos, configuramos varias propiedades del gráfico, como el color de la línea, el color del texto, su tamaño, la capacidad de hacer zoom o desplazarse por la gráfica con gestos táctiles, y finalmente invocamos el método "Invalidate" para recargar los datos.

```
// Crear el conjunto de datos y personalizarlo
val dataSetOsat = LineDataSet(entriesOsat, label: "Osat")
dataSetOsat.color = Color.BLUE
dataSetOsat.valueTextColor = Color.BLACK
dataSetOsat.valueTextSize = 16f

// Crear el gráfico de líneas y personalizarlo
val dataOsat = LineData(dataSetOsat)
chartOsat.data = dataOsat
chartOsat.setTouchEnabled(true)
chartOsat.setPinchZoom(true)
chartOsat.description.text = "Osat"
chartOsat.setNoDataText("No hay datos disponibles")
chartOsat.invalidate()
```

También se puede configurar una propiedad para mostrar un texto si no hay datos disponibles.



Implementación Barra de Navegación

Algo importante también es la implantación de la barra de herramientas donde está el menú para acceder a ciertas pantallas, se ha realizado a base de intents pero se quitaron las animaciones para mejorar la navegación:

```
└─ sgg
    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        // Handle item selection
        return when (item.itemId) {
            R.id.mn_menu -> {
                // Creamos un Intent para iniciar VistaSeleccionPartida.
                val intent = Intent( packageContext: this, MainActivity::class.java)

                intent.addFlags(Intent.FLAG_ACTIVITY_NO_ANIMATION)

                // Iniciamos la actividad sin esperar un resultado.
                startActivity(intent)

                true
            }
            R.id.mn_perfil -> {
                // Creamos un Intent para iniciar VistaSeleccionPartida.
                val intent = Intent( packageContext: this, ProfileView::class.java)

                intent.addFlags(Intent.FLAG_ACTIVITY_NO_ANIMATION)

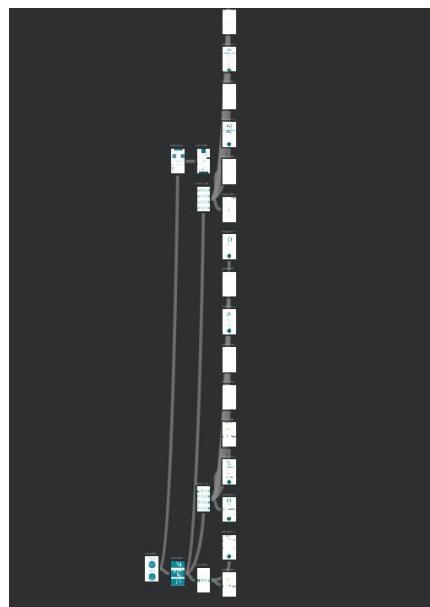
                // Iniciamos la actividad sin esperar un resultado.
                startActivity(intent)

                true
            }
        }
    }
```

Implementación Navegación entre Fragments

Una de las principales implementaciones para navegar entre los fragments es a través del navigation que permite de forma fácil o pasar de un fragment a otro.

El navigation sería esto:



Y se utilizan de esta forma, lo cual hace que a través de las acciones se vaya de un fragment a otro:

```
// Agrega OnClickListener al botón btnJugarlocal
btnOpcionAlimentacion.setOnClickListener { it: View! 
    // Navega al fragmento de vistaTableroView cuando se hace clic en el botón
    findNavController().navigate(R.id.action_menuOpcionesMedicasView_to_menu_deporte_sueno)
}
```

Las acciones que genera el navigation para poder utilizarlo como se muestra anteriormente serían estas:

```
<fragment
    android:id="@+id/menuOpcionesMedicasView"
    android:name="com.example.proyectohospitalgambia.feature.vistaMenuOpcionesMedicas.MenuOpcionesMedicasView"
    android:label="fragment_menu_opciones_medicas_view"
    tools:layout="@layout/fragment_menu_opciones_medicas_view" >
    <action
        android:id="@+id/action_menuOpcionesMedicasView_to_menu_Introducir_Manual"
        app:destination="@+id/menu_Introducir_Manual" />
    <action
        android:id="@+id/action_menuOpcionesMedicasView_to_menu_deporte_suenio"
        app:destination="@+id/menu_deporte_suenio" />
    <action
        android:id="@+id/action_menuOpcionesMedicasView_to_menuMoodAndEnergyView"
        app:destination="@+id/menuMoodAndEnergyView" />
</fragment>
```

Implementación Recogida de Datos

A través del botón se recogen los datos de los elementos que hay en el formulario, creamos una Pol a través del id del usuario, el id de la Pol auto generado y el Json que generamos con los datos del formulario, una vez creada la Pol a través del objeto del formulario que pertenezca, asignamos la Pol al usuario y terminamos la acción.

```
btnGuardar.setOnClickListener { it: View ->

    val usuarioActivo = MainActivity.usuario

    // Obtener los datos del formulario
    val datosFormulario = obtenerDatosFormulario()
    // Verificar si se obtuvieron los datos del formulario correctamente
    if (datosFormulario != null) {

        // Mostrar un mensaje de éxito
        Toast.makeText(context, "Datos guardados con éxito", Toast.LENGTH_SHORT).show()

        // Generar IDs aleatorios como strings
        val idPols = generarIdAleatorio()
        val idBook = MainActivity.usuario?.id.toString() // Asumiendo que MainActivity.idUsuario es un Long o un Int

        val pol = Pol(idPols, idBook, datosFormulario.toString(), isSubido: "false")

        usuarioActivo?.polos?.add(pol)

        // Llamar al método del ViewModel para insertar datos
        val resultado = viewModel.insertarDatosEnBaseDeDatos(pol)

        if (resultado){
            // Navegar hacia atrás
            requireActivity().supportFragmentManager.popBackStack()
        } else {
            Toast.makeText(requireContext(), "Ha ocurrido un error, datos no guardados", Toast.LENGTH_SHORT).show()
        }
    }

    private fun generarIdAleatorio(): String {
        return UUID.randomUUID().toString()
    }

    // Método para obtener los datos del formulario y crear el JSON
    ▲ sgg +1
    private fun obtenerDatosFormulario(): JSONObject? {
        // Obtener los valores de los EditText
        val pesoText = edtPeso.text.toString()

        // Verificar si algún campo está vacío
        if (pesoText.isEmpty()) {
            // Mostrar un Toast indicando que algún campo está vacío
            Toast.makeText(context, "Por favor, complete todos los campos", Toast.LENGTH_SHORT).show()
            return null // Devolver null para indicar que no se han completado todos los campos
        }

        // Convertir los valores a tipos numéricos
        val pesoKg = pesoText.toInt()

        // Obtener la fecha y hora actual
        val currentDateAndTime = SimpleDateFormat("yyyy-MM-dd HH:mm:ss", Locale.getDefault()).format(
            Date()
        )

        val peso = Peso(
            fechaRealizacion = currentDateAndTime,
            kg = pesoKg
        )
    }
}
```

```
// Crear el objeto JSON con los datos del formulario
val jsonObject = JSONObject()
jsonObject.put("name: "TipoPol", peso.tipoPol)
jsonObject.put("name: "FechaInsercion", peso.fechaRealizacion)
jsonObject.put("name: "kg", peso.kg)

Log.d(tag: "JSON Data", jsonObject.toString())

// Limpiar los elementos del formulario después de obtener los datos si son correctos
edtPeso.text.clear()

return jsonObject
}
```

También a través de este método insertamos la Pol en la BBDD para que se guarde en local, ya que la aplicación siempre trabaja en local.

```
👤 sgg
fun insertarDatosEnBaseDeDatos(
    pol: Pol
): Boolean {
    return databaseHelper.insertFormData(pol)
}
```

Implementación Subida de Datos

Uno de los principales problemas a resolver era definir la estructura del JSON requerida para realizar la solicitud al servidor, por lo que creé la estructura JSON necesaria.

```
try {

    // Recuperar las pols
    val pols = viewModel.recuperarDatos()

    // Crear el encabezado del JSON
    val jsonString2 = """
        {
            "creation_info": {
                "node": "mygnuhealth",
                "timestamp": "2024-03-26T18:13:46.604899",
                "user": "ESPGNU7770RG"
            },
            "domain": "medical",
            "fsynced": true,
            "genetic_info": null,
            "id": "2bb7e529-c583-4d6e-ad24-e858196f98af",
            "measurements": [
                {
                    "hr": 12444218888
                }
            ]
        }
    """.trimIndent()
```

Una vez hemos realizado lo anterior, integramos la subida de los datos al servidor en una corutina para evitar que bloquee la pantalla, considerando que esta operación puede tardar unos segundos.

```
// Iniciar una corutina para manejar las solicitudes secuencialmente
viewModelJob = CoroutineScope(Dispatchers.IO).launch { this: CoroutineScope
    var fallo = false

    for (pol in pols) {
        if (usuarioEncontrado != null && usuarioEncontrado.id == pol.book && pol.isSubido.equals(
            other: "false",
            ignoreCase = true
        )
    ) {
        val jsonConcatenar =
            "${pol.data.trimEnd(...chars: '}')}, ${jsonString2.trimStart(...chars: '{')}"

            .trimIndent() // Eliminar espacios en blanco adicionales

        // Realizar la solicitud y esperar a que se complete antes de continuar
        val result = viewModel.insertarDatosEnServidorAsync(jsonConcatenar)

        // Verificar si la solicitud fue exitosa
        if (!result) {
            // Si falla alguna solicitud, actualizar la variable de fallo
            fallo = true
        } else {
            // Si la solicitud fue exitosa, actualizar el estado en la base de datos
            viewModel.actualizarEstadoSubidoEnBD(pol.idPol, nuevoEstado: "true")
        }
    }
}
```

Este método se encarga de subir los datos al servidor a través de la URL proporcionada y devuelve un valor booleano (true o false) para indicar si se realizó la subida correctamente o no. Es fundamental manejar todos los posibles errores de manera adecuada para garantizar un funcionamiento estable del sistema.

```

fun insertarDatosEnServidorAsync(polJsonString: String): Boolean {
    return runBlocking { thisCoroutineScope {
        try {
            val idPagina = generarIdAleatorio()
            val url = MainActivity.url + "/pols/ESPGNU7770RG/" + idPagina

            val request = Request.Builder()
                .url(url)
                .post(polJsonString.toRequestBody(jsonMediaType))
                .header("Authorization", Credentials.basic(username: "ESPGNU7770RG", password: "gnusolidario"))
                .build()

            val response = withContext(Dispatchers.IO) { thisCoroutineScope {
                client.newCall(request).execute()
            }

            if (!response.isSuccessful) {
                println("Error al enviar la solicitud: ${response.message}")
                return@runBlocking false
            }

            val resultado = response.body?.string()
            val fallo = resultado == null || resultado.equals(other: "null", ignoreCase = true)
            return@runBlocking !fallo
        } catch (e: IllegalArgumentException) {
            // Captura la excepción y muestra un mensaje de error
            println("Error al construir la URL: ${e.message}")
            e.printStackTrace() // Imprime la traza de la excepción en Logcat
            return@runBlocking false
        } catch (e: Exception) {
            // Captura la excepción y muestra un mensaje de error
            println("Error al construir la URL: ${e.message}")
            e.printStackTrace() // Imprime la traza de la excepción en Logcat
            return@runBlocking false
        }
    }
}
}

```

Una vez completado el proceso anterior, se muestra un diálogo en otro hilo para informar al usuario sobre si la subida de los datos se ha realizado correctamente o no. Además, se actualiza la lista para mostrar los datos que han sido correctamente subidos al servidor del usuario que está actualmente logueado.

```

// Mostrar el diálogo apropiado en el hilo principal después de que todas las solicitudes se hayan completado
withContext(Dispatchers.Main) { thisCoroutineScope {

    // Hacer visible el ProgressBar
    progresBarSubirDatos.visibility = View.INVISIBLE

    if (fallo) {
        mostrarDialogoPolsNoSubidos()
    } else {
        mostrarDialogoPolsSubidos()
    }

    val listView = federacionServidoresView.findViewById<ListView>(R.id.lst_conexionesServidor)
    val polList = viewModel.recuperarDatos().filter { pol ->
        usuarioEncontrado != null && usuarioEncontrado.id == pol.book && pol.isSubido == "true"
    }
    val adapter = PolAdapter(polList, requireContext())
    listView.adapter = adapter
}
}

```

También es crucial controlar la disponibilidad de conexión a internet para evitar posibles fallos, ya que sin conexión no se puede acceder al servidor. Utilizamos un método para verificar si la aplicación tiene conexión a internet, devolviendo un valor booleano que indica si hay o no conexión.

```
if (!isNetworkAvailable(requireContext())) {  
    // Mostrar un mensaje de que no hay conexión a Internet  
    Toast.makeText(requireContext(), text: "No hay conexión a Internet", Toast.LENGTH_SHORT).show()  
    return@setOnClickListener  
}
```

```
private fun isNetworkAvailable(context: Context): Boolean {  
    val connectivityManager = context.getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager  
    val network = connectivityManager.activeNetwork  
    val capabilities = connectivityManager.getNetworkCapabilities(network)  
    return capabilities?.hasCapability(NetworkCapabilities.NET_CAPABILITY_INTERNET) == true  
}
```

Gracias a esta variable, podemos controlar el hilo y, en caso de que se destruya o pause la vista, se cancela el hilo para evitar que continúe realizando la operación. Esto es útil para prevenir que la aplicación se bloquee, aunque puede resultar en la pérdida de información sobre si se han subido los datos o no. Este enfoque también se utiliza en la parte de Bluetooth para evitar problemas similares con los hilos.

```
private var viewModelJob: Job? = null
```

```
└ sgg
override fun onDestroyView() {
    super.onDestroyView()
    viewModelJob?.cancel()
    progresBarSubirDatos.visibility = View.INVISIBLE
}

└ sgg
override fun onPause() {
    super.onPause()
    viewModelJob?.cancel()
    progresBarSubirDatos.visibility = View.INVISIBLE
}

└ sgg
override fun onResume() {
    super.onResume()
    viewModelJob?.cancel()
    progresBarSubirDatos.visibility = View.INVISIBLE
}
```

Métodos Destacados DatabaseHelper

Obtenemos todas las Pols:

```
fun obtenerPols(): List<Pol> {
    val polsList = mutableListOf<Pol>()
    val db = readableDatabase
    val cursor = db.rawQuery( sql: "SELECT * FROM $TABLE_POLS", selectionArgs: null)

    cursor.use { cursor ->
        // Verificar si el cursor tiene columnas
        if (cursor.columnCount > 0) {
            // Obtener los índices de las columnas
            val idIndex = cursor.getColumnIndex(KEY_POLS_ID)
            val bookIndex = cursor.getColumnIndex(KEY_POLS_BOOK)
            val dataIndex = cursor.getColumnIndex(KEY_POLS_DATA)
            val subidoIndex = cursor.getColumnIndex(KEY_POLS_SUBIDO)

            // Iterar sobre el cursor
            while (cursor.moveToNext()) {
                // Obtener los valores de cada columna por su índice
                val id = cursor.getString(idIndex)
                val book = cursor.getString(bookIndex)
                val data = cursor.getString(dataIndex)
                val subido = cursor.getString(subidoIndex)

                // Crear un objeto Pol con los datos obtenidos de la base de datos
                val pol = Pol(id, book, data, subido)
                polsList.add(pol)
            }
        }
    }
}
```

Insertamos todas las Pols:

```
✉ sgg
fun insertFormData(pol: Pol): Boolean {
    val db = writableDatabase
    val values = ContentValues().apply { this: ContentValues
        put(KEY_POLS_ID, pol.idPol)
        put(KEY_POLS_BOOK, pol.book)
        put(KEY_POLS_DATA, pol.data)
        put(KEY_POLS_SUBIDO, pol.isSubido)
    }
    val newRowId = db.insert(TABLE_POLS, nullColumnHack: null, values)
    db.close()
    return newRowId != -1L
}
```

Comprobamos que el usuario que se va a introducir tiene las credenciales y los datos requeridos para dar su acceso:

```

fun verificarCredenciales(nombreUsuario: String, contraseniaUsuario: String): PeopleUser? {
    val db = readableDatabase
    val selection = "${DatabaseHelper.KEY_PEOPLE_DATA} LIKE ?"
    val selectionArgs = arrayOf("%\"name\": \"$nombreUsuario\"%")
    val cursor = db.query(
        DatabaseHelper.TABLE_PEOPLE,
        arrayOf(DatabaseHelper.KEY_PEOPLE_ID, DatabaseHelper.KEY_PEOPLE_DATA),
        selection,
        selectionArgs,
        groupBy: null,
        having: null,
        orderBy: null
    )

    var usuario: PeopleUser? = null
    cursor.use { it // Utilizamos use para asegurar que el cursor se cierre correctamente al finalizar
        if (cursor.moveToFirst()) {
            val idIndex = cursor.getColumnIndex(DatabaseHelper.KEY_PEOPLE_ID)
            val dataIndex = cursor.getColumnIndex(DatabaseHelper.KEY_PEOPLE_DATA)
            if (idIndex != -1 && dataIndex != -1) { // Verificamos que los indices de las columnas sean válidos
                val id = cursor.getString(idIndex)
                val data = cursor.getString(dataIndex)
                val jsonObject = JSONObject(data)
                val contraseniaAlmacenada = jsonObject.getString("password")
            }
        }
    }

    // Verificar si la contraseña proporcionada coincide con la contraseña almacenada
    if (BCrypt.checkpw(contraseniaUsuario, contraseniaAlmacenada)) {
        usuario = PeopleUser(id, data)
    } else {
        Log.e("VerificarCredenciales", "Contraseña incorrecta para el usuario: $nombreUsuario")
    }
} else {
    // Las columnas no fueron encontradas en el cursor
    // Puede ser un error en los nombres de las columnas
    // o las columnas no están presentes en la tabla
    Log.e("VerificarCredenciales", "Las columnas no fueron encontradas en el cursor.")
}
} else {
    Log.e("VerificarCredenciales", "No se encontró ningún usuario con el nombre: $nombreUsuario")
}

db.close()

return usuario
}
}

```

Gracias a este método podemos insertar los usuarios:

```

fun insertarPersona(peopleUser: PeopleUser): Boolean {
    val db = writableDatabase
    val values = ContentValues().apply { this[KEY_PEOPLE_ID] = peopleUser.id
        this[KEY_PEOPLE_DATA] = peopleUser.data
    }
    val newRowId = db.insert(TABLE_PEOPLE, nullColumnHack: null, values)
    return newRowId != -1L
}

```

7. Fase de pruebas.

7.1. Pruebas de unidad.

Se han utilizado las librerías JUnit y Mockito para las pruebas unitarias. Mockito es una herramienta que nos permite crear mocks, que son objetos simulados que imitan el comportamiento de objetos reales en un entorno controlado. En las pruebas unitarias, los mocks se emplean para simular el comportamiento de las dependencias de una clase, lo que permite aislar la clase que estamos probando y verificar su comportamiento de forma independiente.

El funcionamiento resumido de Mockito se puede describir en los siguientes pasos:

1. Configuración del mock: Se define el comportamiento esperado del mock para ciertos métodos o acciones.
2. Uso del mock: Durante la ejecución de la prueba, se utiliza el mock en lugar de la implementación real.
3. Verificación del comportamiento: Se verifica que el objeto bajo prueba interactúe correctamente con el mock según lo esperado.

Esta estructura básica se sigue en cada prueba que utiliza Mockito. A continuación, se proporciona una explicación más detallada de cada fase de la prueba:

Configuración del mock: En esta fase, se configura el comportamiento esperado del mock para los métodos o acciones específicas que se van a simular durante la prueba. Esto se logra utilizando métodos como `when()` de Mockito para definir el comportamiento esperado del mock.

Uso del mock: Durante la ejecución de la prueba, se utiliza el mock en lugar de la implementación real de la clase bajo prueba. Esto nos permite simular el comportamiento de las dependencias y centrarnos en probar el comportamiento específico de la clase bajo prueba.

Verificación del comportamiento: Una vez completada la ejecución de la prueba, se verifica que el objeto bajo prueba haya interactuado correctamente con el mock según lo esperado. Esto se logra utilizando métodos de verificación de Mockito, como `verify()`.

En resumen, estas pruebas unitarias se han utilizado para validar el funcionamiento de los métodos en la clase DatabaseHelper.

testInsertarPersona()

- Propósito: Verifica que el método insertarPersona() funcione correctamente al insertar una persona en la base de datos.
- Descripción:
 - Crea un objeto PeopleUser para la prueba.
 - Configura el comportamiento del mock para que devuelva true al llamar insertarPersona() con el objeto creado.
 - Llama al método insertarPersona() y verifica que devuelve true.
 - Verifica que el método insertarPersona() fue llamado con el objeto PeopleUser.

```
@Test
fun testInsertarPersona() {
    val peopleUser = PeopleUser( id: "1" , data: "data")
    every { databaseHelper.insertarPersona(peopleUser) } returns true
    val result = databaseHelper.insertarPersona(peopleUser)
    assert(result)
    verify { databaseHelper.insertarPersona(peopleUser) }
    println("La prueba 'testInsertarPersona' se ejecutó correctamente")
}
```

testInsertarPersonaConDatosInvalidos()

- Propósito: Verifica que el método insertarPersona() maneje correctamente datos inválidos.
- Descripción:
 - Crea un objeto PeopleUser con datos inválidos para la prueba.
 - Configura el comportamiento del mock para que devuelva false al llamar insertarPersona() con el objeto de datos inválidos.
 - Llama al método insertarPersona() y verifica que devuelve false.
 - Verifica que el método insertarPersona() fue llamado con el objeto de datos inválidos.

```
@Test  
fun testInsertarPersonaConDatosInvalidos() {  
    val peopleUserInvalido = PeopleUser( id: "", data: "")  
    every { databaseHelper.insertarPersona(peopleUserInvalido) } returns false  
    val result = databaseHelper.insertarPersona(peopleUserInvalido)  
    assert(!result)  
    verify { databaseHelper.insertarPersona(peopleUserInvalido) }  
    println("La prueba 'testInsertarPersonaConDatosInvalidos' se ejecutó correctamente")  
}
```

testVerificarCredenciales()

- Propósito: Verifica que el método `verificarCredenciales()` funcione correctamente al verificar las credenciales de un usuario.
- Descripción:
 - Crea un nombre de usuario y una contraseña para la prueba.
 - Configura el comportamiento del mock para que devuelva un objeto `PeopleUser` al llamar `verificarCredenciales()` con los datos de usuario.
 - Llama al método `verificarCredenciales()` y verifica que devuelve el objeto `PeopleUser` esperado.
 - Verifica que el método `verificarCredenciales()` fue llamado con los datos de usuario.

```
@Test  
fun testVerificarCredenciales() {  
    val nombreUsuario = "testUser"  
    val contraseniaUsuario = "testPassword"  
    val peopleUser = PeopleUser(nombreUsuario, contraseniaUsuario)  
    every { databaseHelper.verificarCredenciales(nombreUsuario, contraseniaUsuario) } returns peopleUser  
    val result = databaseHelper.verificarCredenciales(nombreUsuario, contraseniaUsuario)  
    assertEquals(peopleUser, result)  
    verify { databaseHelper.verificarCredenciales(nombreUsuario, contraseniaUsuario) }  
    println("La prueba 'testVerificarCredenciales' se ejecutó correctamente")  
}
```

testObtenerTodosLosDatosPesoCorrecto()

- Propósito: Verifica que el método `obtenerTodosLosDatosPeso()` devuelva correctamente los datos de peso para un usuario.
- Descripción:
 - Crea datos de prueba.
 - Configura el comportamiento del mock para que devuelva los datos de prueba al llamar `obtenerTodosLosDatosPeso()` con el ID de usuario.

- Llama al método obtenerTodosLosDatosPeso() y verifica que devuelve los datos de prueba.
- Verifica que el método obtenerTodosLosDatosPeso() fue llamado con el ID de usuario correcto.

```
@Test
fun testObtenerTodosLosDatosPesoCorrecto() {
    val peso1 = Peso( tipoPol: "Peso", fechaRealizacion: "2022-01-01", kg: 70)
    val peso2 = Peso( tipoPol: "Peso", fechaRealizacion: "2022-01-02", kg: 71)
    val datosPrueba = listOf(peso1, peso2)
    every { databaseHelper.obtenerTodosLosDatosPeso( idUsuario: "1" ) } returns datosPrueba
    val result = databaseHelper.obtenerTodosLosDatosPeso( idUsuario: "1" )
    assertEquals(datosPrueba, result)
    verify { databaseHelper.obtenerTodosLosDatosPeso( idUsuario: "1" ) }
    println("La prueba 'testObtenerTodosLosDatosPesoCorrecto' se ejecutó correctamente")
}
```

testObtenerTodosLosDatosPesoIncorrecto()

- Propósito: Verifica que el método obtenerTodosLosDatosPeso() maneje correctamente el caso de no encontrar datos de peso para un usuario.
- Descripción:
 - Configura el comportamiento del mock para que devuelva una lista vacía al llamar obtenerTodosLosDatosPeso() con el ID de usuario.
 - Llama al método obtenerTodosLosDatosPeso() y verifica que devuelve una lista vacía.
 - Verifica que el método obtenerTodosLosDatosPeso() fue llamado con el ID de usuario correcto.

```
@Test
fun testObtenerTodosLosDatosPesoIncorrecto() {
    every { databaseHelper.obtenerTodosLosDatosPeso( idUsuario: "1" ) } returns emptyList()
    val result = databaseHelper.obtenerTodosLosDatosPeso( idUsuario: "1" )
    assertTrue(result.isEmpty())
    verify { databaseHelper.obtenerTodosLosDatosPeso( idUsuario: "1" ) }
    println("La prueba 'testObtenerTodosLosDatosPesoIncorrecto' se ejecutó correctamente")
}
```

testInsertFormData()

- Propósito: Verifica que el método insertFormData() funcione correctamente al insertar datos de un formulario.
- Descripción:
 - Crea un objeto Pol para la prueba.

- Configura el comportamiento del mock para que devuelva true al llamar insertFormData() con el objeto creado.
- Llama al método insertFormData() y verifica que devuelve true.
- Verifica que el método insertFormData() fue llamado con el objeto Pol.

```
@Test
fun testInsertFormData() {
    val pol = Pol(idPol: "1", book: "book", data: "data", isSubido: "subido")
    every { databaseHelper.insertFormData(pol) } returns true
    val result = databaseHelper.insertFormData(pol)
    assertTrue(result)
    verify { databaseHelper.insertFormData(pol) }
    println("La prueba 'testInsertFormData' se ejecutó correctamente")
}
```

testListarPols()

- Propósito: Verifica que el método obtenerPols() devuelva correctamente una lista de pols.
- Descripción:
 - Crea una lista de pols de prueba.
 - Configura el comportamiento del mock para que devuelva la lista de pols al llamar obtenerPols().
 - Llama al método obtenerPols() y verifica que la lista no sea nula, tenga el tamaño esperado y contenga los elementos esperados.

```
@Test
fun testListarPols() {
    val expectedPols = listOf(Pol(idPol: "id1", book: "book1", data: "data1", isSubido: "subido"),
        Pol(idPol: "id2", book: "book2", data: "data2", isSubido: "subido"))
    every { databaseHelper.obtenerPols() } returns expectedPols
    val pols = databaseHelper.obtenerPols()
    Assert.assertNotNull(pols)
    Assert.assertEquals( message: "El tamaño de la lista de pols no es el esperado", expectedPols.size, pols.size)
    Assert.assertTrue( message: "La lista de pols no contiene los elementos esperados", pols.containsAll(expectedPols))
    println("La prueba 'testListarPols' se ejecutó correctamente")
}
```

Al ejecutar la clase que contiene las pruebas, todas se completan correctamente y proporcionan el resultado esperado.

```
La prueba 'testObtenerTodosLosDatosPesoIncorrecto' se ejecutó correctamente
La prueba 'testVerificarCredenciales' se ejecutó correctamente
La prueba 'testInsertarPersona' se ejecutó correctamente
La prueba 'testObtenerTodosLosDatosPesoCorrecto' se ejecutó correctamente
La prueba 'testListarPols' se ejecutó correctamente
La prueba 'testInsertFormData' se ejecutó correctamente
La prueba 'testInsertarPersonaConDatosInvalidos' se ejecutó correctamente
BUILD SUCCESSFUL in 12s
24 actionable tasks: 10 executed, 14 up-to-date
```

¿Por qué solo se han realizado pruebas de la clase DatabaseHelper?

La clase DatabaseHelper no tiene asociados elementos visuales, por lo que no pertenece a ninguna actividad ni fragmento. Esto nos ha permitido realizar pruebas en ella sin problemas. Sin embargo, nos hemos enfrentado a dificultades al intentar realizar pruebas para el resto de las clases que sí contienen elementos visuales.

¿Por qué ocurre esto?

- Dependencia de Android: Las actividades y fragmentos requieren acceso a recursos y componentes específicos de Android, lo que dificulta la generación de pruebas unitarias.
- Ciclo de vida complicado: El ciclo de vida de las actividades y fragmentos, gestionado por Android, añade complejidad a las pruebas de unidad al tener que simular adecuadamente estos estados.

Una solución posible sería implementar uno de los siguientes frameworks, creados específicamente para trabajar con clases que tienen elementos visuales:

1. Robolectric: Aunque se utiliza principalmente con Java, también es compatible con Kotlin y permite ejecutar pruebas unitarias en un entorno de JVM simulando el comportamiento de Android. Es útil para probar clases que dependen del ciclo de vida de la actividad o fragmento.
2. Espresso: Es el framework más común para pruebas de interfaz de usuario en Android, y es totalmente compatible con Kotlin. Espresso permite escribir pruebas concisas para interactuar con los elementos de la interfaz de usuario de la aplicación.
3. UI Automator: Si necesitas realizar pruebas que abarquen múltiples aplicaciones o componentes del sistema, UI Automator es una opción sólida. También es compatible con Kotlin y te permite interactuar con elementos de la interfaz de usuario en diferentes aplicaciones.

7.2. Pruebas de validación y aceptación.

Tipo de Usuario	Resultado Esperado	Resultado Obtenido	Observaciones
Usuario Desarrollador	Uso exitoso de todas las funcionalidades sin errores	Uso exitoso de todas las funcionalidades sin errores	Todo funcionó como se esperaba
Usuario Avanzado	Uso efectivo de la aplicación sin errores	Uso efectivo de la aplicación sin errores	Todo funcionó como se esperaba
Usuario Medio-Alto	Uso de todas las funcionalidades con posibles dificultades en los menús	Uso de todas las funcionalidades con algunas dificultades en los menús	Se encontraron problemas de navegación en los menús
Usuario Medio	Dificultades para acceder a ciertas opciones y realizar tareas específicas	Problemas al subir datos y conectar Bluetooth	Se encontraron problemas de conectividad y subida de datos
Usuario Bajo	Dificultades para utilizar la aplicación y posibles errores no controlados	Dificultades encontradas y errores durante la transmisión de datos	Se encontraron errores durante la transmisión de datos y se necesitó asistencia para tareas básicas
Usuario sin Experiencia	Dificultades para utilizar la aplicación sin asistencia	Dificultades encontradas, pero pudo utilizar la aplicación con asistencia	Se encontraron problemas de usabilidad y se necesitó asistencia constante

7.3. Pruebas de usabilidad.

Para realizar las pruebas de usabilidad hemos valorado distintos apartados:

- **Número de clics:**
 - Para tareas como llenar y enviar un formulario, el número de clics necesarios es mayor de lo deseado debido a la presencia de múltiples menús y una interfaz confusa, lo que dificulta la accesibilidad y la navegación.
- **Facilidad de memorización:**
 - La aplicación carece de flujo intuitivo, lo que dificulta su memorización y la hace menos atractiva para los usuarios. Los menús son complejos y poco intuitivos, lo que dificulta su uso.
- **Gestión visual de errores:**

- La aplicación proporciona una gestión visual detallada de errores, lo que ayuda a los usuarios a comprender y solucionar los problemas que puedan surgir durante su uso.
- **Tiempo de tareas:**
 - El tiempo necesario para completar tareas como la subida de datos es razonablemente rápido.
 - Sin embargo, la toma de datos de los dispositivos Bluetooth puede ser más lenta debido a la transmisión de datos innecesarios por parte de los dispositivos, lo que puede afectar la eficiencia del proceso.
- **Accesibilidad:**
 - La aplicación no cumple con los estándares de accesibilidad para personas con discapacidad, ya que no incorpora funcionalidades diseñadas para ayudar a este grupo de usuarios. Además, su dificultad de uso puede suponer un obstáculo adicional para las personas con discapacidad.

7.4. Pruebas de integración del sistema.

La aplicación ha sido probada en esta serie de dispositivos:

- **Xiaomi Redmi note 10 pro**
 - La aplicación se adapta correctamente a su pantalla de 6,43 pulgadas.
 - Se conecta adecuadamente a los dispositivos médicos mediante Bluetooth, ya que su versión de Android es la 11, lo que la hace lo suficientemente moderna y fiable para recibir los datos de manera efectiva.
 - No presenta problemas para subir los datos al servidor, ya que su conexión es bastante estable.
 - Su manejo es sencillo por lo que no supone problemas a la hora de llenar los formularios y navegación entre las distintas pantallas.
- **Xiaomi Redmi note 12 pro**
 - La aplicación se adapta correctamente a su pantalla de 6.67 pulgadas.
 - Se conecta adecuadamente a los dispositivos médicos mediante Bluetooth, ya que su versión de Android es la 14, lo que la hace lo suficientemente moderna y fiable para recibir los datos de manera efectiva.
 - No presenta problemas para subir los datos al servidor, ya que su conexión es bastante estable.
 - Su manejo es sencillo por lo que no supone problemas a la hora de llenar los formularios y navegación entre las distintas pantallas.

- **Un emulador Píxel_3a_Api34**

- La aplicación se adapta correctamente a su pantalla de 5,6 pulgadas.
- Se conecta adecuadamente a los dispositivos médicos mediante Bluetooth, ya que su versión de Android es la 9.0 Pie, lo que la hace lo suficientemente moderna y fiable para recibir los datos de manera efectiva.
- No presenta problemas para subir los datos al servidor, ya que su conexión es bastante estable.
- Su manejo es sencillo por lo que no supone problemas a la hora de llenar los formularios y navegación entre las distintas pantallas.

- **Xiaomi Redmi Pad SE**

- La aplicación se adapta correctamente a su pantalla de 11 pulgadas.
- Se conecta adecuadamente a los dispositivos médicos con Bluetooth, ya que su versión de Android es la 13, pero puede tener problemas al recibir los datos ya que es más lenta por su bajo procesador y poca RAM.
- No presenta problemas para subir los datos al servidor, ya que su conexión es bastante estable, pero su subida se puede demorar más ya que su tratamiento de datos es más lento.
- Su manejo no es tan sencillo por su tamaño, por lo que cuesta llenar los formularios y navegación entre las pantallas.

- **Smart TV**

- La aplicación no se adapta correctamente a una pantalla de 50 pulgadas y hay algún problema, pero sigue permitiendo el funcionamiento de la aplicación.
- Se conecta adecuadamente a los dispositivos médicos con Bluetooth, pero puede tener problemas al recibir los datos ya que es más lenta por su bajo procesador y poca RAM.
- No presenta problemas para subir los datos al servidor, ya que su conexión es bastante estable, pero su subida se puede demorar más ya que su tratamiento de datos es más lento.
- Su manejo no es tan sencillo por su tamaño, por lo que cuesta llenar los formularios y navegación entre las pantallas.

- **Huawei y6 2018**

- La aplicación se adapta correctamente a su pantalla de 5,7 pulgadas.
- Se conecta adecuadamente a los dispositivos médicos mediante Bluetooth, ya que su versión de Android es la 8.0 Oreo, pero se hace con el método deprecated ya que es más antiguo de lo aceptado.
- No presenta problemas para subir los datos al servidor, ya que su conexión es bastante estable.
- Su manejo es sencillo por lo que no supone problemas a la hora de llenar los formularios y navegación entre las distintas pantallas.

8. Trabajo futuro.

El desarrollo de la aplicación móvil abre numerosas posibilidades de expansión y mejora en diversas áreas. A continuación, se presentan algunas de las posibles ampliaciones que podrían considerarse en futuras iteraciones del proyecto:

- **Conexión con más Dispositivos Médicos:**
 - Se podría explorar la posibilidad de ampliar la compatibilidad de la aplicación para conectarse con una variedad más amplia de dispositivos médicos que permitan medir diferentes condiciones físicas, lo que enriquecería la experiencia del usuario y aumentaría la utilidad de la aplicación.
- **Gestión de Usuarios con Distintos Permisos:**
 - Implementar un sistema de gestión de usuarios más robusto que permita definir diferentes roles y permisos de acceso, por ejemplo, otorgando a los usuarios médicos la capacidad de modificar los datos de sus pacientes, garantizando así una mejor colaboración entre profesionales de la salud y una atención más personalizada.
- **Sistema de Citas Online:**
 - Integrar una funcionalidad de solicitud de citas médicas online, que permita a los usuarios programar citas con sus médicos correspondientes de manera conveniente a través de la aplicación, optimizando así la gestión de citas y mejorando la accesibilidad a la atención médica.
- **Mejoras en la Seguridad de Acceso:**
 - Reforzar la seguridad de la aplicación mediante la implementación de autenticación multifactorial, que añade capas adicionales de seguridad al proceso de inicio de sesión, protegiendo así los datos sensibles de los usuarios de manera más efectiva, y que la contraseña deba tener una complejidad mínima requerida.
- **Mejoras en la Interfaz de Usuario:**
 - Realizar mejoras en la interfaz de usuario para hacerla más intuitiva y fácil de usar, teniendo en cuenta los comentarios de los usuarios y las tendencias de diseño actuales, lo que contribuiría a una experiencia de usuario más satisfactoria y atractiva.
- **Mejoras en la Subida de Datos al servidor:**
 - Se debería realizar una mejora clave en la subida de datos, utilizando la librería Retrofit para su implementación. Sin embargo, esto requiere mejorar el servidor y su gestión de las subidas.
 - Además, es necesario mejorar el tratamiento de los datos JSON, ya que su estructura está codificada de forma rígida, lo cual no debería ser así. Esta mejora también depende de una mejor recepción de datos por parte del servidor.

- Asimismo, se debería mejorar la conexión con el servidor y habilitar tanto Path como Get para mantener los mismos datos en local y en el servidor, aunque estas mejoras no son posibles debido a limitaciones en el servidor.
- Arreglar la cancelación de la subida de datos al servidor para evitar la duplicación.

9. Conclusiones.

Durante el desarrollo de este proyecto, se han experimentado diversas sensaciones y aprendizajes significativos. Uno de los aspectos más destacados ha sido el crecimiento personal y profesional al enfrentarse a nuevas tecnologías y desafíos técnicos.

En primer lugar, el proceso de investigación y adquisición de conocimientos sobre nuevas tecnologías, como la conexión con dispositivos médicos y la gestión de servidores, ha sido fundamental para el desarrollo exitoso de la aplicación. Esta experiencia ha contribuido significativamente al desarrollo de habilidades de resolución de problemas y a la capacidad de adaptarse a entornos tecnológicos en constante evolución.

Además, el trabajo con herramientas como Git y otras nuevas tecnologías ha proporcionado una base sólida para el crecimiento profesional continuo y la adaptabilidad en futuros proyectos. La experiencia adquirida en la gestión de versiones y colaboración en equipo a través de Git ha sido especialmente valiosa y representa un paso importante en el desarrollo de habilidades de desarrollo de software colaborativo.

En resumen, este proyecto ha sido una experiencia enriquecedora que ha permitido no solo el desarrollo de habilidades técnicas, sino también la oportunidad de contribuir positivamente a una causa benéfica. La colaboración con una ONG para abordar necesidades de salud en países necesitados ha sido una experiencia gratificante y significativa, destacando el potencial del desarrollo tecnológico para generar un impacto positivo en la sociedad.



MyGNUHealth
THE GNU HEALTH PERSONAL HEALTH RECORD

10. Manuales

10.1. Manual de Usuario

Si es la primera vez que se inicia la aplicación, la única acción disponible es la creación de un nuevo usuario para acceder a las funciones restantes. Esto se hace fácilmente al hacer clic en el botón de registro.



Sergio García Gómez

En la pantalla de registro, puedes ingresar los datos del usuario que deseas crear. El nombre y la contraseña que elijas aquí serán los que se utilicen para acceder a la aplicación.



The registration form consists of several input fields and a central button:

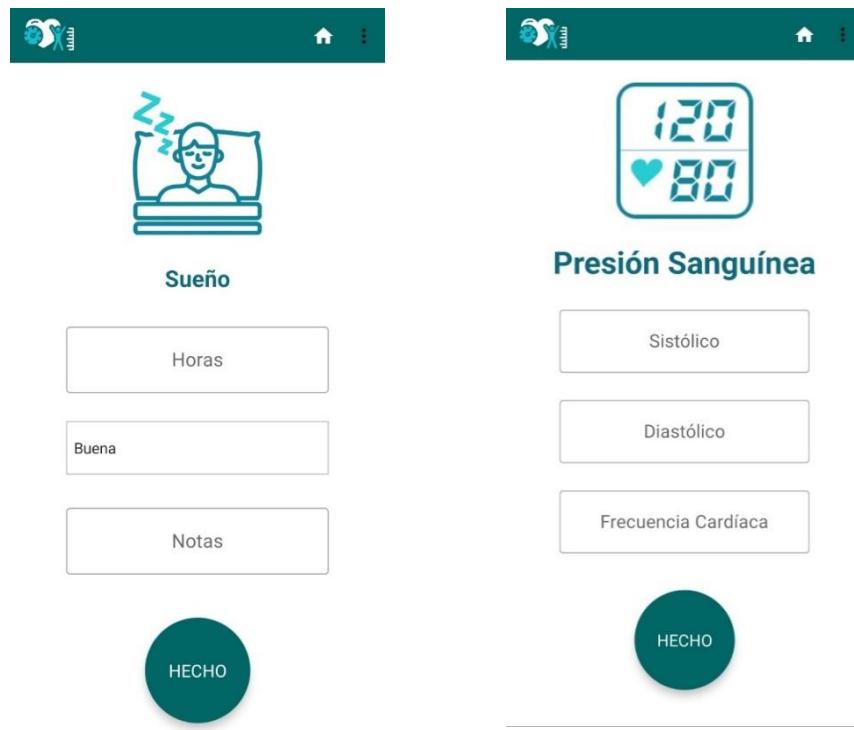
- A decorative header image featuring a person meditating with various icons (clock, DNA, bag) floating around them.
- An input field labeled "Nombre de Usuario" (User Name).
- Two input fields for "Sexo" (Gender) and "Fecha Nacimiento" (Birth Date), each with dropdown menus for "Masculino" (Male), "dd" (day), "mm" (month), and "yyyy" (year).
- A slider for "Altura: 50 cm" (Height: 50 cm).
- Two input fields for "Contraseña" (Password) and "Repetir Contraseña" (Repeat Password).
- A large teal button labeled "REGISTRARSE" (Register).

MyGNUHealth
The GNU Health Personal Health Record

Si ya tienes un usuario, puedes iniciar sesión para acceder al resto de funciones de la aplicación. Al iniciar sesión, primero se mostrará una pantalla para elegir entre datos médicos o datos del servidor. Nos centraremos primero en los datos médicos. Este es uno de los menús desde el cual puedes acceder tanto a las pantallas para introducir los datos manualmente, pulsando los iconos de la izquierda, como a la visualización de gráficas, pulsando en los iconos de la derecha.



A continuación, se presentan un par de pantallas de ejemplo en las cuales puedes introducir los datos manualmente.



Los datos introducidos pueden visualizarse de manera gráfica en el menú mencionado anteriormente. Al pulsar el botón del tipo de dato que deseas visualizar, los datos se mostrarán en forma de gráfica interactiva.



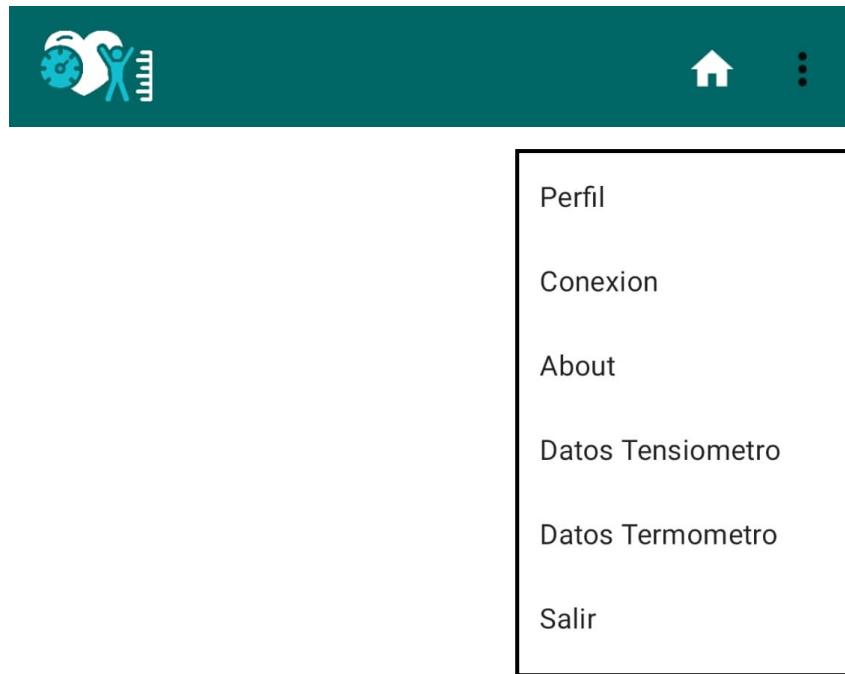
En las opciones del servidor, al pulsar el botón de la izquierda, se mostrará una pantalla relacionada con el libro de vida. Además, encontrarás un botón para sincronizar los datos, que se encarga de subir los datos locales que no se hayan subido aún, y una lista que muestra los datos locales disponibles.



La pantalla correspondiente al libro de vida permite visualizar los registros médicos almacenados en la base de datos. En esta pantalla, puedes encontrar una lista de registros con detalles como la fecha, el tipo de registro y una breve descripción.



La toolbar de la aplicación cuenta con un botón en forma de casa, que te lleva de vuelta a la pantalla de inicio, y otro botón que te da acceso a un conjunto de opciones adicionales.



La opción de perfil te permite modificar los datos del usuario, como la altura o la contraseña.



La opción de conexión te permite modificar los parámetros del servidor y realizar una prueba de conexión.



La opción "About" muestra información sobre la aplicación y las personas que la han creado.



Finalmente, las opciones de termómetro y tensiómetro te permiten realizar la conexión con el dispositivo médico correspondiente y leer sus datos.

Lectura termómetro



Lectura automática del termómetro

Instrucciones

1. Asegúrate de tener activado el Bluetooth
2. Asegúrate de tener activada la ubicación
3. Asegúrate de tener activados los permisos
4. Enciende el dispositivo médico, realiza la medición y pulsa el botón inferior para recibir los datos

Temperatura obtenida



Temperatura 0

Lectura tensiómetro



Lectura automática del tensiómetro

Instrucciones

1. Asegúrate de tener activado el Bluetooth
2. Asegúrate de tener activada la ubicación
3. Asegúrate de tener activados los permisos
4. Enciende el dispositivo médico, realiza la medición y pulsa el botón inferior para recibir los datos

Datos obtenidos



Tension Alta 0

Tension Baja 0

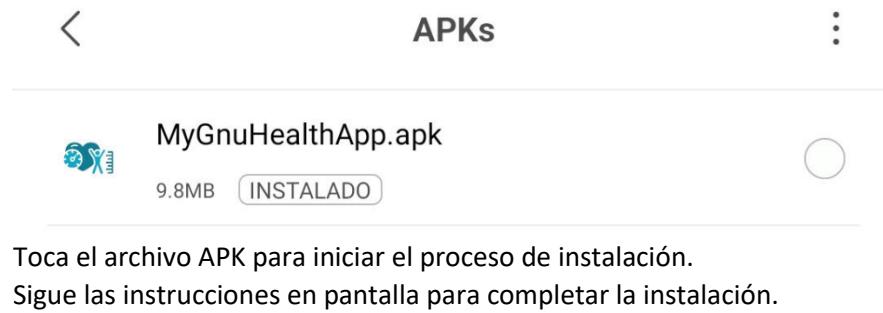
Pulso 0

10.2. Manual de Administrador

- Instalación:
 - Descarga la aplicación APK desde la ubicación proporcionada, en el apartado 11.



- Abre la aplicación de administración de archivos en tu dispositivo Android.
- Navega hasta la ubicación donde se descargó el archivo APK.



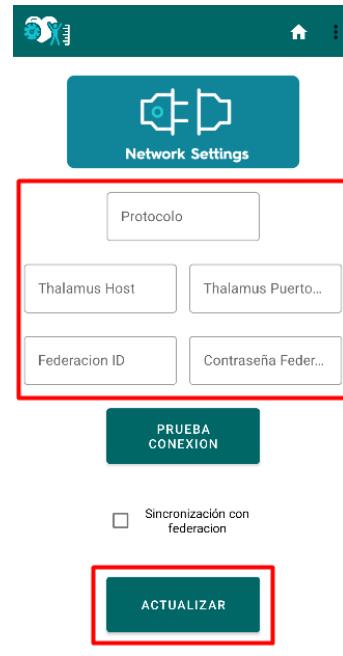
- Configuración del Servidor:
 - La configuración al servidor ya viene configurada al servidor principal, pero sería posible cambiarla de la siguiente manera:
 - Toca el ícono de configuración en la esquina superior derecha de la pantalla, para llegar a la pestaña de conexión.



Opciones Servidor

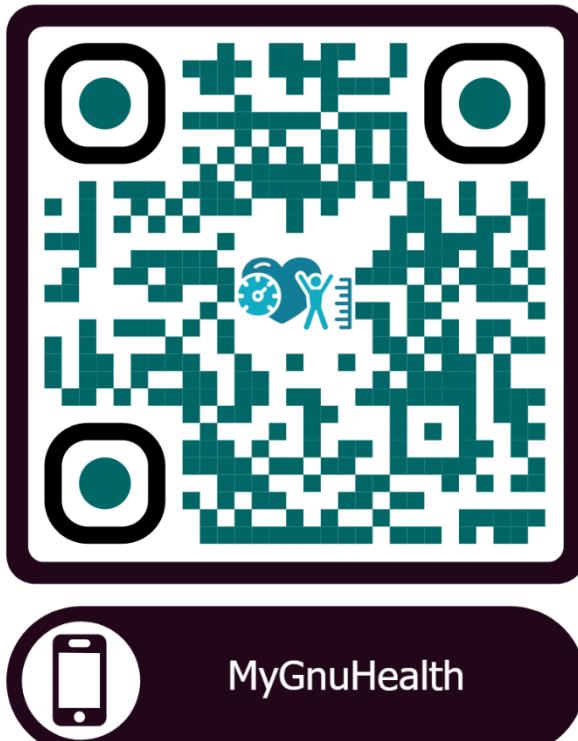


- Introduce la nueva ruta del servidor en el campo proporcionado.
- Selecciona la opción "Guardar" y la aplicación estará configurada para enviar datos al nuevo servidor.



11. Descargar aplicación

Puedes descargar la aplicación a través de este código QR:



También puedes entrar en este enlace y descargar la aplicación:

<https://mariomunpeq.github.io/DescargaApkGnuHealth/>

12. Bibliografía

- AristiDevs. (29 de Marzo de 2024). *MVVM en ANDROID*. Obtenido de Youtube: <https://youtu.be/hhhSMXi0R3E?si=d1Mx1CFyrZZ-g4Rj>
- AristiDevs. (20 de Marzo de 2024). *Navega en Android de forma eficiente*. Obtenido de Youtube: <https://www.youtube.com/watch?v=1N6xmCHZexo>
- Coders, E. O. (5 de Abril de 2024). *Android Charts / Bar Chart / MP Android Chart*. Obtenido de Youtube: <https://www.youtube.com/watch?v=WdsmQ3Zyn84>
- Coders, E. O. (20 de Abril de 2024). *Custom Buttons Design*. Obtenido de Youtube: <https://www.youtube.com/watch?v=OyCVFFse4cs>
- dev.xcheko51x. (5 de Abril de 2024). *Crear Gráfica de Barras usando MPAndroidChart*. Obtenido de Youtube: <https://www.youtube.com/watch?v=cOy8SRs7Rs8>
- Dimas, J. (2 de 5 de 2024). *Youtube*. Obtenido de Youtube: https://youtu.be/Z_5o9tVQOpU?si=huW625D0ekp7bYVi
- Emil. (1 de Abril de 2024). *Android BLE characteristics getValue was deprecated in API level 33*. Obtenido de StackOverflow: <https://stackoverflow.com/questions/76166150/android-ble-characteristics-getvalue-was-deprecated-in-api-level-33-whats-corr>
- Español, L. S. (2 de 5 de 2024). *Youtube*. Obtenido de Youtube: https://youtu.be/Z0yLerU0g-Q?si=P_4qozdao9M1Smb1
- GitHub. (2 de 5 de 2024). *GitHub Foro*. Obtenido de GitHub Foro: <https://github.com/topics/foro>
- Hernández, J. J. (04 de Abril de 2024). *josejuansanchez.org*. Obtenido de Unidad 03. Del modelo conceptual al modelo relacional: <https://josejuansanchez.org/bd/unidad-03-teoria/index.html>
- InnovaDomotics. (25 de Marzo de 2024). *Comunicación Bluetooth con Android Studio*. Obtenido de <https://www.youtube.com/watch?v=7hlYihvpaPQ>
- InnovaDomotics. (26 de Marzo de 2024). *Comunicación Bluetooth con Android Studio Soporte - Parte 1*. Obtenido de Youtube: <https://www.youtube.com/watch?v=dK07j4XOPiE>
- jsgalarraga. (1 de Mayo de 2024). *android studio error Unable to locate adb*. Obtenido de StackOverflow: <https://es.stackoverflow.com/questions/206207/android-studio-error-unable-to-locate-adb>
- Mike, J. (3 de Marzo de 2024). *Wi-Fi Debug (ADB)*. Obtenido de StackOverflow: <https://stackoverflow.com/questions/71221039/wi-fi-debug-adb-there-was-an-error-pairing-the-device>
- MooreDev. (26 de Marzo de 2024). *Cómo CREAR el ICONO de tu APP desde ANDROID STUDIO*. Obtenido de Youtube: <https://www.youtube.com/watch?v=oI6WgrHwsuI>
- Nicolas. (2 de 5 de 2024). *Youtube*. Obtenido de Youtube: <https://youtu.be/fJa3cshrFWs?si=TLipyW8EmV2cp7t6>

Overflow, S. (2 de 5 de 2024). *stackoverflow*. Obtenido de stackoverflow:
<https://es.stackoverflow.com/questions/572152/problema-al-visualizar-gr%C3%A1ficas-en-r>

Programador, L. c. (2 de 5 de 2024). Youtube. Obtenido de Youtube:
<https://youtu.be/WkcnmUuLbV4?si=SOh88sDL1EtRNelk>

Soporte oficial Android. (2 de Abril de 2024). *Android Developers*. Obtenido de BluetoothGattCharacteristic:
<https://developer.android.com/reference/android/bluetooth/BluetoothGattCharacteristic>

Soporte oficial Android. (20 de Marzo de 2024). *Android Developers*. Obtenido de Capture and read bug reports: <https://developer.android.com/studio/debug/bug-report>

13. Enlace al código

Enlace para acceder al código desde GitHub:

<https://github.com/Sergioggvm/ProyectoHospitalGambia.git>