



MARIO MUÑOZ PEQUEÑO

Tutor: Juan Antonio Carrasco

Contenido

Introducción.....	1
Análisis de la aplicación.....	2
Requisitos.....	2
▪ Requisitos funcionales.....	2
▪ Requisitos no funcionales.....	2
Recursos necesarios para el desarrollo.....	3
Recursos hardware.....	3
Recursos software.....	3
Metodología.....	4
Metodología de desarrollo.....	4
1º Ciclo.....	4
2º Ciclo.....	7
3º Ciclo.....	7
4ª Ciclo.....	9
5º Ciclo.....	10
Planificación.....	13
Diseño del proyecto.....	21
Diseño de la BBDD.....	21
Interacción con el usuario.....	22
▪ Casos de uso.....	23
▪ Diseño de la interfaz.....	24
▪ Diseño de la arquitectura.....	25
▪ Arquitectura del servidor.....	26
▪ Arquitectura del cliente.....	26
▪ Relación con otros sistemas.....	27
Detalles de implementación.....	27
Fase de pruebas.....	38
Pruebas de unidad.....	38
Pruebas de validación y aceptación.....	44
Pruebas de usabilidad.....	44
Pruebas de integración del sistema.....	45
Manual de usuario.....	46
Manual de Administrador.....	53
Trabajo futuro.....	55
Conclusiones.....	57

Bibliografía58

Introducción.

La digitalización en el ámbito de la salud es un proceso inevitable y fundamental para mejorar la eficiencia y la calidad de la atención médica. En este contexto, el proyecto se centra en desarrollar una versión para dispositivos Android de la aplicación móvil MyGNUHealth, originalmente creada en Kivy, con el objetivo de satisfacer las demandas crecientes de acceso rápido y fácil a la información de salud.

La decisión de migrar MyGNUHealth a la plataforma Android surge de las limitaciones y desafíos encontrados en el desarrollo con Kivy. Aunque Kivy es una herramienta versátil para crear aplicaciones multiplataforma, presenta ciertas restricciones en términos de rendimiento, compatibilidad y capacidad de integración con el ecosistema de Android. Por lo tanto, se opta por utilizar Kotlin como lenguaje de programación y Android Studio como entorno de desarrollo, para aprovechar las ventajas y las características específicas de Android, permitiendo una mejor optimización y una experiencia de usuario más fluida.

Además de la migración, se incorpora una nueva funcionalidad que permite la lectura de mediciones de temperatura desde un termómetro Bluetooth, mejorando la utilidad de la aplicación y la experiencia del usuario. Esta adición no solo simplifica el proceso de registro de datos, sino que también lo hace más eficiente, al permitir la transferencia directa de información a la aplicación mediante Bluetooth.

Tanto los datos introducidos manualmente como los obtenidos a través de la función Bluetooth se almacenan localmente en una base de datos SQLite. Sin embargo, se incluye la capacidad de enviar esta información a un servidor, lo que proporciona opciones adicionales para el análisis y la gestión de datos.

En resumen, el proyecto aborda las necesidades de digitalización en el sector de la salud mediante la migración y mejora de la aplicación MyGNUHealth para dispositivos Android, junto con la implementación de una nueva funcionalidad que amplía sus capacidades y beneficios para los usuarios.

Análisis de la aplicación

Requisitos.

■ Requisitos funcionales.

R1. Introducción de Datos Manualmente: La aplicación debe permitir a los usuarios introducir manualmente datos de salud, como la temperatura corporal, la presión arterial, etc.

R2. Lectura con Bluetooth: La aplicación debe ser capaz de conectarse y comunicarse con dispositivos Bluetooth, como un termómetro Bluetooth, para leer automáticamente los datos biométricos del usuario.

R3. Conexión al Servidor y Subida de Datos: La aplicación debe ser capaz de conectarse al servidor del instituto y subir los datos recopilados de los usuarios para su almacenamiento y análisis.

R4. Gestión de Usuarios (Inicio de Sesión): La aplicación debe incluir un sistema de registro e inicio de sesión para que los usuarios puedan autenticarse de manera segura y acceder a sus datos personales.

R5. Aplicación Traducida en Distintos Idiomas: La aplicación debe ser multilingüe y permitir a los usuarios cambiar el idioma de la interfaz de usuario según sus preferencias.

R6. Persistencias de datos: La aplicación debe utilizar el almacenamiento de manera local y remota en el dispositivo, permitiendo el acceso a la información incluso cuando no hay conexión a internet.

■ Requisitos no funcionales.

R1. Rendimiento: La aplicación debe ser rápida y eficiente, con tiempos de respuesta rápidos y una interfaz de usuario fluida, incluso en dispositivos con recursos limitados.

R2. Compatibilidad de hardware: La aplicación debe ser compatible con una amplia gama de dispositivos Android, desde versiones más antiguas hasta las más recientes, asegurando una experiencia uniforme para todos los usuarios a partir de la API 24.

R3. Adaptabilidad de Pantalla (Responsive): La interfaz de usuario de la aplicación debe ser adaptable y compatible con diferentes tamaños y resoluciones de pantalla, asegurando una experiencia consistente en dispositivos de distintas dimensiones, en formato vertical y con un aspecto de resolución 16:9 o 16:10, en formatos cuadrados como 4:3 no se mostrarán los datos correctamente.

R4. Desplazamiento Sencillo dentro de la APP: La aplicación debe tener una navegación intuitiva y fácil entre las diferentes pantallas y funciones, permitiendo a los usuarios acceder rápidamente a la información que necesitan.

R5. Facilidad de Uso: La aplicación debe ser fácil de usar y entender, con una interfaz de usuario intuitiva y guías claras para ayudar a los usuarios a realizar tareas específicas sin dificultad.

R6. Seguridad de Datos (Contraseñas Encriptadas): La aplicación debe garantizar la seguridad de los datos del usuario mediante la encriptación de las contraseñas y la implementación de medidas de seguridad robustas para proteger la privacidad de la información personal.

Recursos necesarios para el desarrollo.

Recursos hardware.

- Ordenador para el desarrollo que cumpla como mínimo, los requisitos de Android Studio:
 - 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.
- Dispositivos móviles para pruebas y depuración de la aplicación en diferentes configuraciones de hardware y versiones de Android.
- Se recomienda una conexión a Internet con una velocidad de al menos 10 Mbps para garantizar la eficiencia en la descarga de librerías y actualizaciones necesarias durante el desarrollo.
- Termómetro Beurer medical para comprobar la funcionalidad bluetooth de la aplicación.

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.
- **Photoshop 2022:** Programa de diseño con la que se han realizado el diseño de los botones de los aparatos bluetooth (Termómetro y tensiómetro)

Recursos software para la documentación:

Microsoft 365 y Documentos de Google para la creación de documentos, Figma para la creación de los esquemas/boceto.

Metodología.

Metodología de desarrollo.

Elegimos la **metodología en espiral** debido a que, por la naturaleza del proyecto, con los cambios y mejoras que se implementan constantemente, es útil la naturaleza iterativa y flexible de esta metodología, para ir añadiendo y comprobando constantemente los cambios.

1º Ciclo

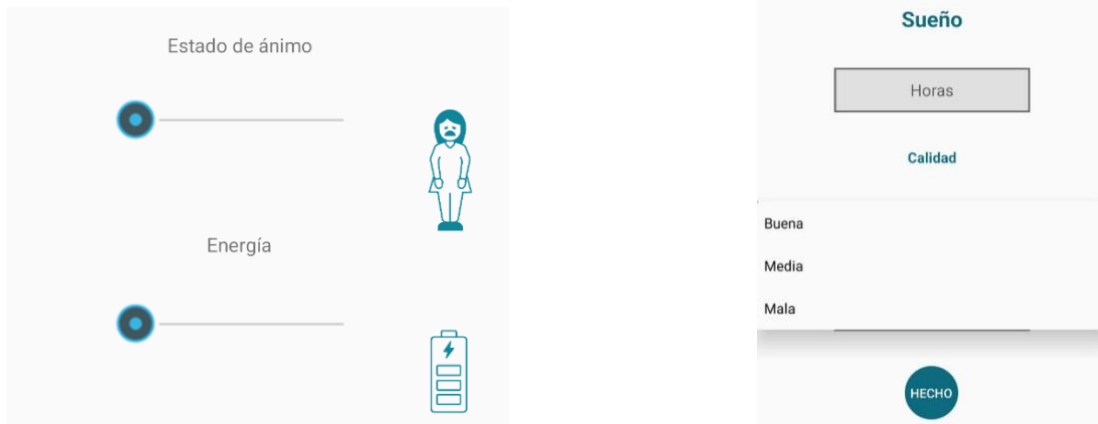
- **Fase 1: Análisis y requisitos**

Se analizan los requisitos iniciales de la aplicación, que son desarrollar la aplicación ya existente de MyGNUHealth, pero de manera nativa en Android, manteniendo las funcionalidades, y una coherencia visual, para que, pareciéndose a la anterior, se adapte al diseño vertical de un móvil, para ello tuvimos que analizar y comprender la aplicación ya existente

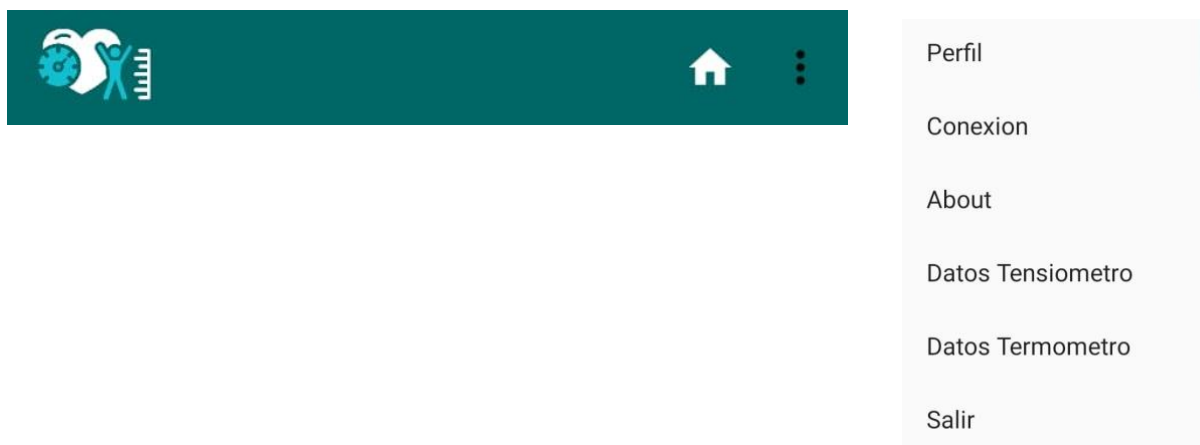
- **Fase 2: Diseño**

Además de las pantallas ya existentes en el programa original, decidimos agregar una toolbar con ciertas funcionalidades rápidas, como ir al menú principal, acceder al perfil, conexiones, una pestaña de "About" con información varia sobre GNU Health, o dos botones, para acceder directamente a las pestañas en las que realizar las mediciones del termómetro y del tensiómetro mediante bluetooth. Para toda esta fase inicial de diseño, antes de ponernos a hacer las pantallas en Android Studio, hicimos los esquemas en Figma

En este primer ciclo, en Kotlin únicamente programamos los elementos visuales, es decir, añadimos funcionalidades a las seekbars y los spinners, para que permitiesen al usuario interactuar correctamente, y que las respectivas imágenes de la derecha fuesen cambiando según el valor seleccionado.



También se hizo la funcionalidad de la toolbar, para que permitiese desplazarse entre las pantallas.



▪ Fase 4: Pruebas

Esta parte, al ser mayoritariamente visual, no tuvo una gran fase de pruebas, pero se comprobó que los desplazamientos entre las distintas actividades y fragments fuesen correctos. También se probaron que las seekbar y spinners funcionasen bien, cambiando las imágenes correspondientes asociadas.

2º Ciclo

- **Fase 1: Análisis y requisitos**

En este segundo ciclo, con la mayor parte visual funcional, decidimos que lo siguiente que íbamos a añadir era la funcionalidad del bluetooth, tanto del termómetro como del tensiómetro. Ya teníamos los botones en la toolbar para acceder a las correspondientes activities, pero no estaba implementado todavía, así que teníamos que crearlas.

En estas activities se debían de poder conectarse, leer datos del dispositivo, y guardarlos, informando al usuario del resultado, tanto si había sido exitoso como si no.

- **Fase 2: Diseño**

Hicimos el diseño en Figma de las dos pantallas, que serían muy similares, debían de tener dos botones, uno para conectarse y leer los datos, y otro que los subiese al servidor, también debía de tener texto para informar al usuario del resultado.

- **Fase 3: Desarrollo**

Se realizó la parte de código correspondiente, y se integró con el ya existente, tanto los layout como los archivos Kotlin con la funcionalidad. Dado que los dispositivos envían todos los datos que tienen almacenadas, se mirará el parámetro de fecha, y se recibirá únicamente el más reciente

- **Fase 4: Pruebas**

Ya que la conexión y lectura del dispositivo es algo que lleva algo de tiempo, decidimos implementar una barra de progreso, para que el usuario tuviese información de lo que estaba ocurriendo, ya que, durante este proceso, podía parecer que la aplicación se había quedado bloqueada.

3º Ciclo

- **Fase 1: Análisis y requisitos**

En este quinto ciclo teníamos que implementar el apartado de las gráficas, para mostrar los datos del usuario de una manera más visual. Los requisitos son que, desde ambos menús, debe haber un botón que, para cada tipo de dato, nos lleve a su gráfica correspondiente, tomando los datos en local.

- **Fase 2: Diseño**

El diseño será simple, tendrá únicamente dos tipos de elemento, un texto, para indicar al usuario qué tipo de dato está viendo, y la gráfica o gráficas correspondientes. Las gráficas tendrán una leyenda, para que pueda asociarse cada color con su dato.

▪ Fase 3: Desarrollo

Para implementar las gráficas, necesitábamos una librería externa, ya que las nativas de Android no permitían desarrollar gráficas al nivel que necesitábamos, por eso encontramos una llamada "MPAndroidChart", como no estaban en el repositorio de Maven, tuvimos que añadir a los repositorios "jitpack.io" en el Gradle para expandir la búsqueda a ese repositorio, en el que se encontraba la librería de gráficas que necesitábamos.

Con la implementación hecha, ya podíamos poner los elementos correspondientes en el layout

```
<com.example.proyectoohospitalgambia.feature.CustomLineChart
    android:id="@+id/graficoLineas_ActividadFisica"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_weight="1"
    android:padding="16dp"
    app:layout_constraintBottom_toTopOf="@+id/tv_pasos"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/tv_actividadFisica" />

<com.example.proyectoohospitalgambia.feature.CustomLineChart
    android:id="@+id/graficoLineas_Pasos"
    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_constraintHorizontal_bias="0.0"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/tv_pasos" />
```

]. La librería viene preparada para hacer distintos tipos de gráfico, por lo que nos vale tanto para los gráficos de línea, que tiene la mayoría, como el de nutrición que es un gráfico de barras.

▪ Fase 4: Pruebas

1. **Pruebas de visualización de datos:** Se revisó que los datos mostrados en las gráficas fueran consistentes con los datos almacenados localmente y que se actualizarán correctamente en caso de nuevos registros.
2. **Pruebas de interacción:** Se verificó que las gráficas fueran interactivas, permitiendo al usuario realizar acciones como hacer zoom, desplazarse por los datos, y obtener información detallada al tocar los puntos de datos.
3. **Pruebas de compatibilidad:** Se probó la aplicación en diferentes dispositivos con variaciones en el tamaño de pantalla y resolución para asegurar que las gráficas se adaptaran correctamente a diferentes configuraciones de hardware.

4ª 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 rellenar 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

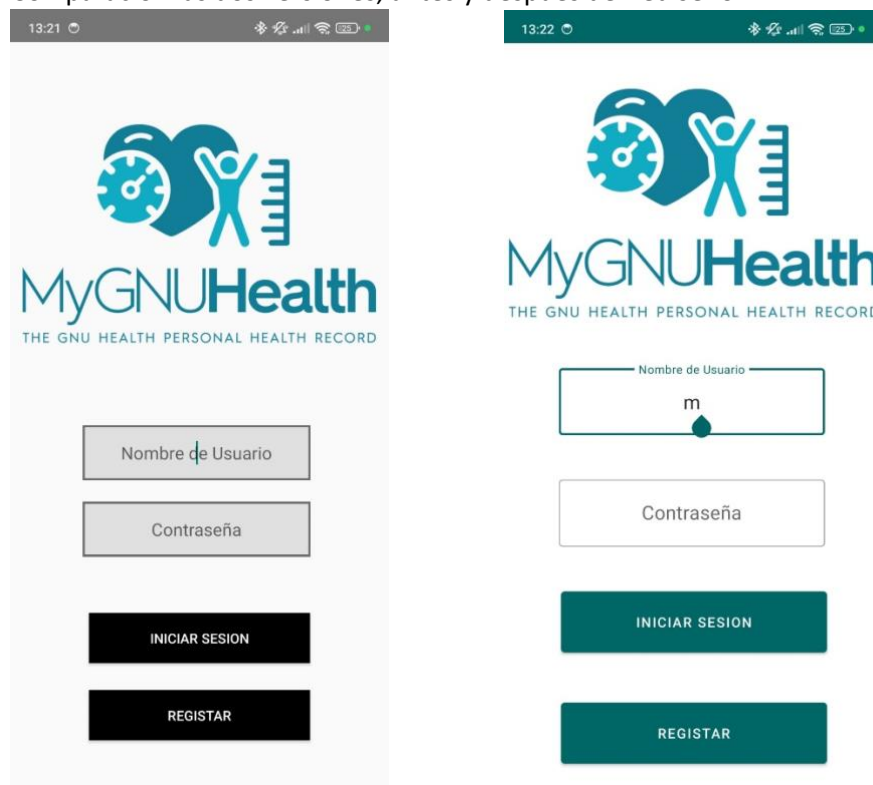
- **Fase 1: Análisis y requisitos**

Como último ciclo, la planificación eran cambiar los elementos de la interfaz gráfica, para que tuviese un aspecto más moderno, decidimos utilizar el "Material Design" como tema, implementado cuando fuese posible sus elementos.

- **Fase 2: Diseño**

La idea de diseño era cambiar el aspecto visual de los elementos y la aplicación, pero manteniendo la estructura de diseño, es decir, no desplazar los elementos de su lugar.

Comparación las dos versiones, antes y después del rediseño:





▪ Fase 3: Desarrollo

Para implementar el tema lo primero fue añadirlo al archivo correspondiente (themes.xml), eligiendo los colores primarios y secundarios correspondientes, para cambiar elementos como la notification bar del teléfono cuando la aplicación esté abierta. Como solo tenemos el diseño en modo claro, se eligen los mismos colores tanto para el archivo del modo claro como del oscuro, para que independientemente del que tenga elegido el usuario se vea como nosotros hemos decidido.

```
<resources xmlns:tools="http://schemas.android.com/tools">
  <!-- 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 era cambiar los elementos iniciales de los layout, por su versión mejorada de "Material Design", principalmente se sustituyeron todos los "Edit text" por "material.textInputLayout", que incluye elementos como animaciones en los "hint". Y se reajustaron los tamaños, tanto de los elementos como del texto, para que tuviesen una mayor cohesión con el nuevo tema.

```

<com.google.android.material.textfield.TextInputLayout
    android:id="@+id/et_systolic_layout"
    style="@style/Widget.MaterialComponents.TextInputLayout.OutlinedBox"
    android:layout_width="275dp"
    android:layout_height="75dp"
    android:hint="Sistólico"
    app:layout_constraintBottom_toTopOf="@+id/et_diastolic_layout"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/textView">

    <com.google.android.material.textfield.TextInputEditText
        android:id="@+id/et_systolic"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center"
        android:inputType="number"
        android:textSize="20sp" />

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

```

▪ Fase 4: Pruebas

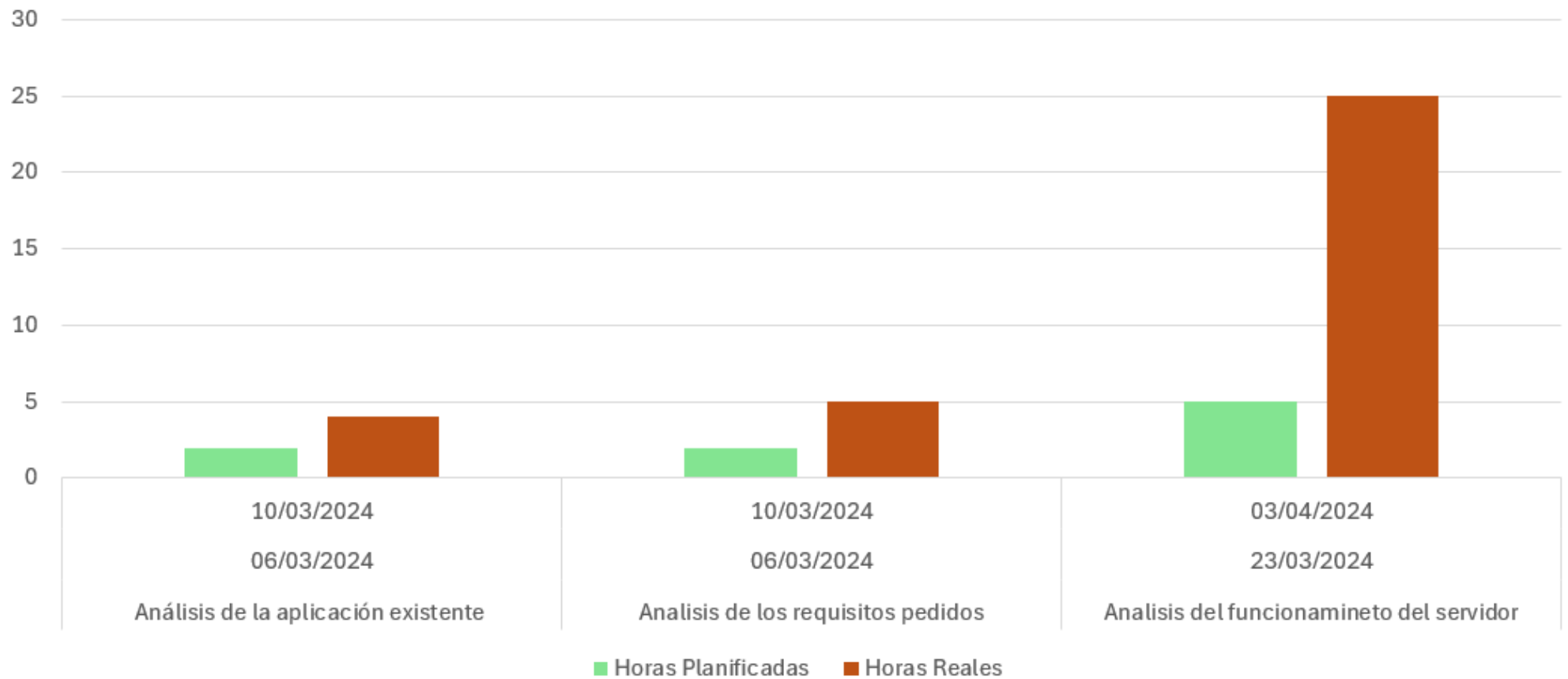
Ya que los cambios eran únicamente visuales, las pruebas debían consistir en utilizar la aplicación de manera habitual, y comprobar que funcionaba correctamente, y todos los elementos se mostraban de la manera correspondiente.

Planificación.

ANALISIS DE REQUISITOS Y APLICACIÓN

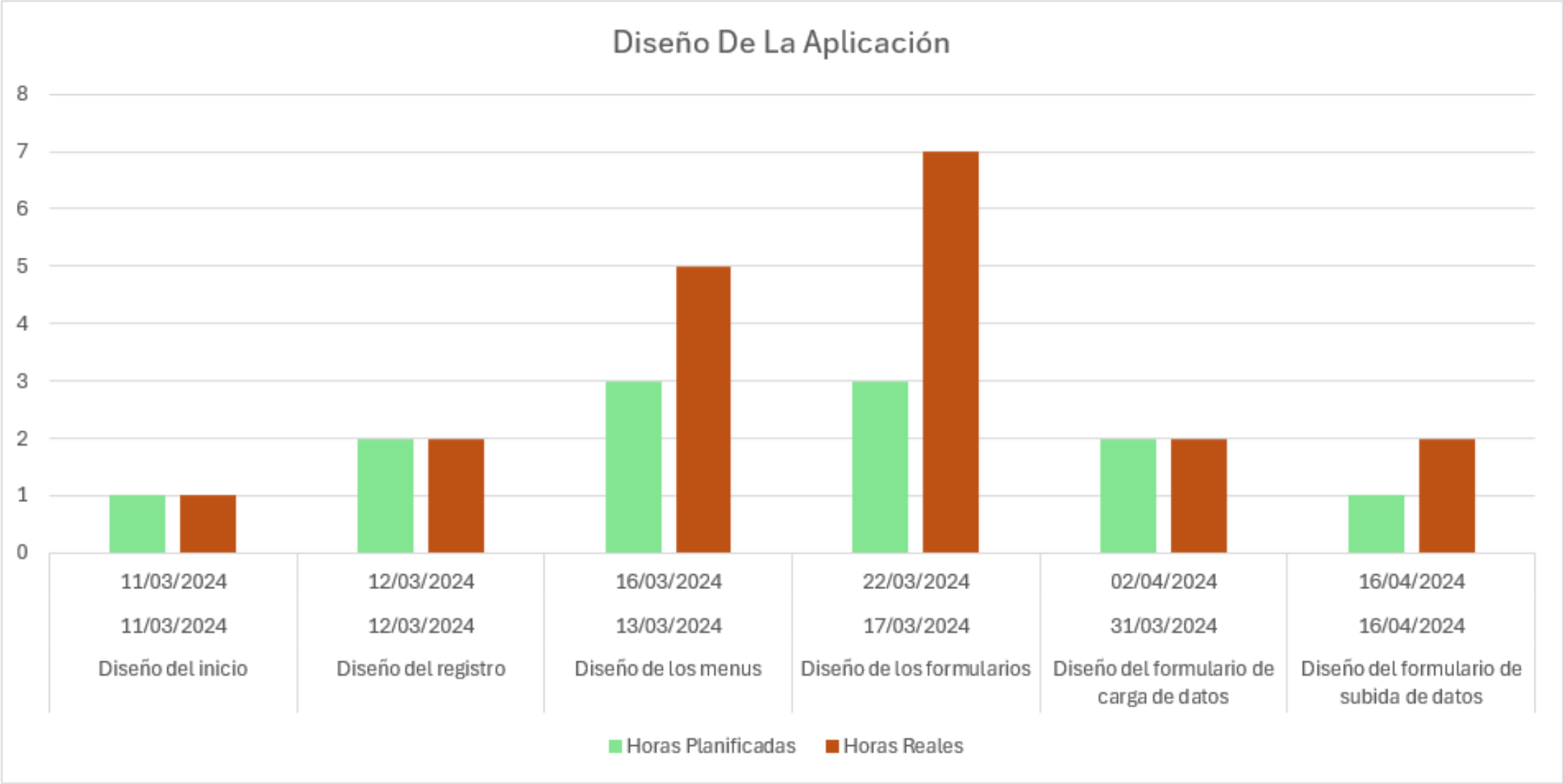
Analisis Realizado	Fecha de inicio	Fecha de fin	Horas Planificadas	Horas Reales	Justificación
Análisis de la aplicación existente	06/03/2024	10/03/2024	2	4	Hubo que analizar la aplicación existente para comprender su funcionamiento y flujo, el cual presentaba deficiencias y no facilitaba la captura de 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 funcionamineto 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 tambien hubo que sacar la estructura la BBDD, lo cual tambien fue complicado.

Analisis De Requisitos Y Aplicación



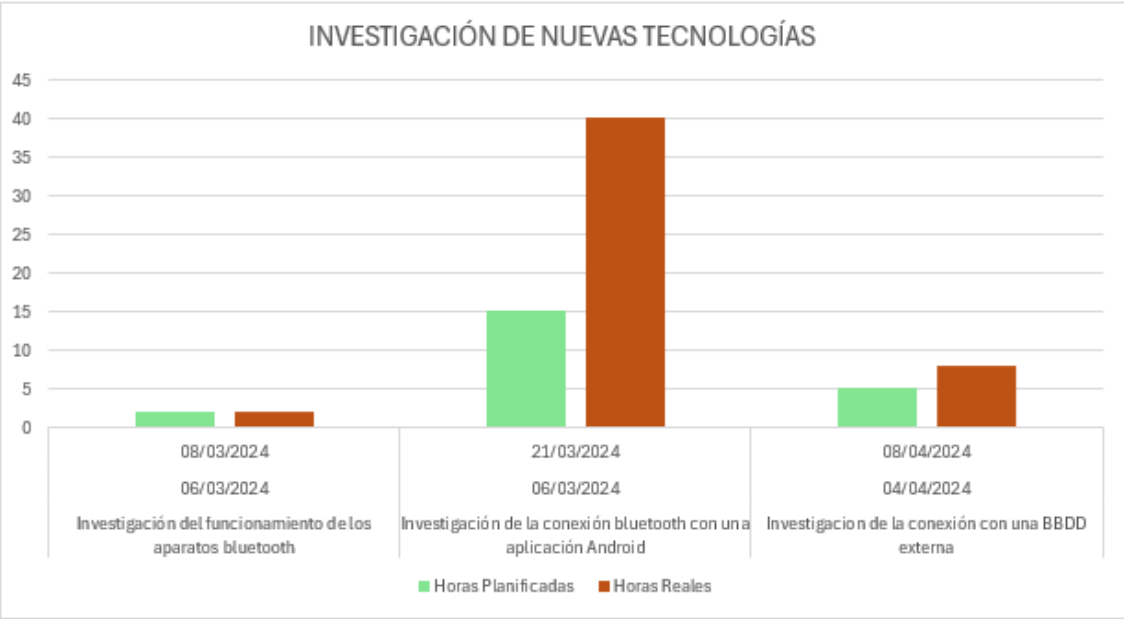
DISEÑO DE LA APLICACIÓN

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 menus	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 pocos elementos.
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.



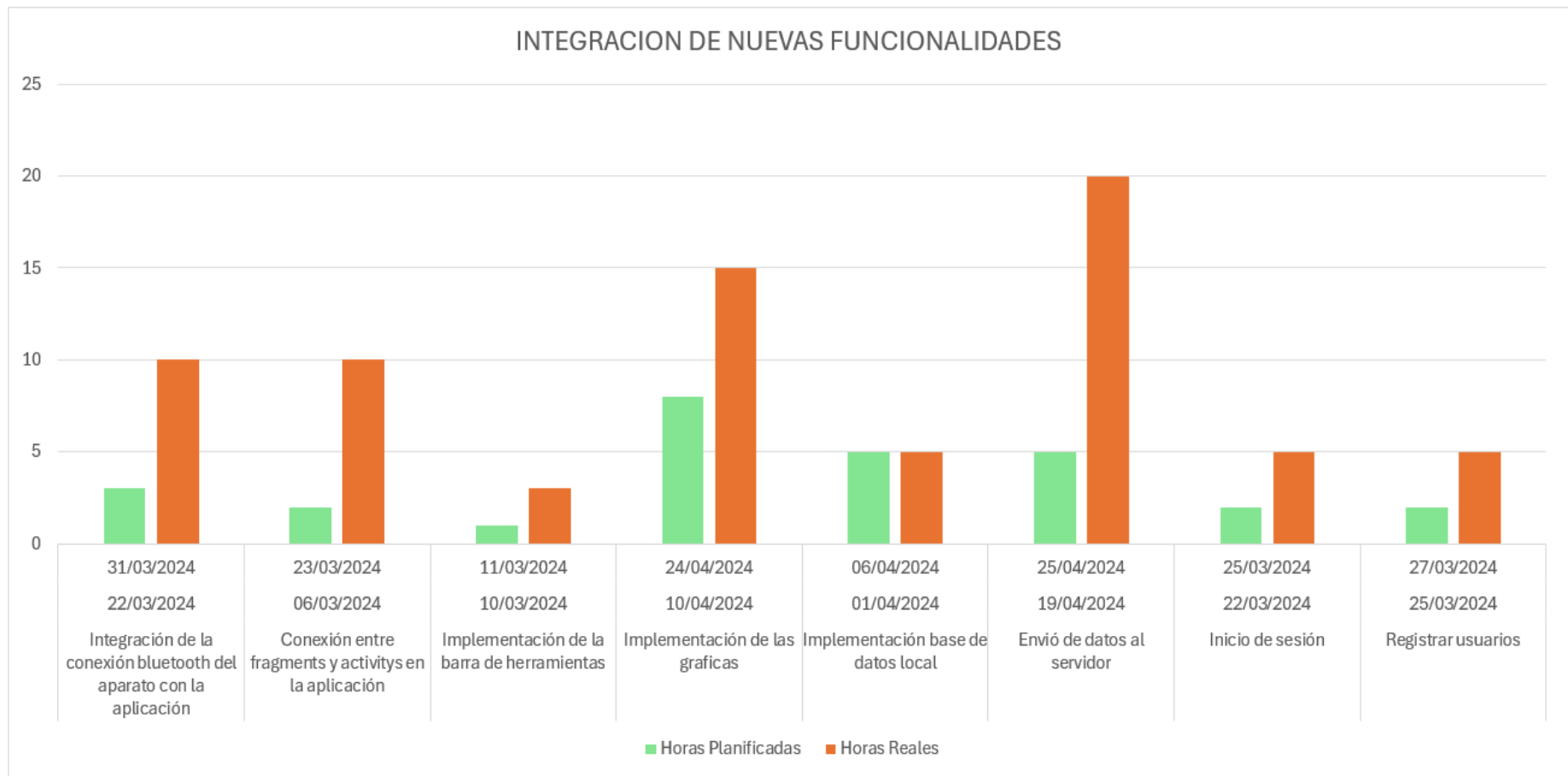
INVESTIGACIÓN DE NUEVAS TECNOLOGÍAS

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.
Investigacion 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.



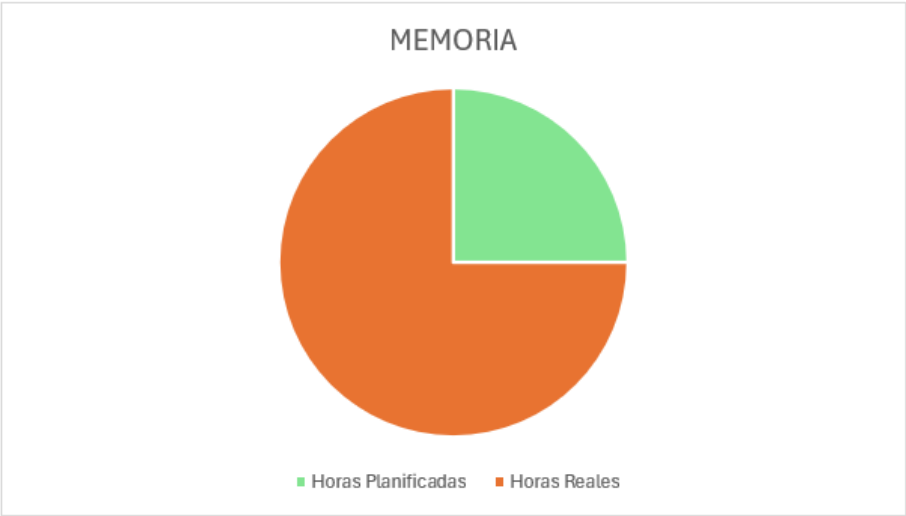
INTEGRACIÓN DE NUEVAS FUNCIONALIDADES

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 activitis 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 graficas	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
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.



MEMORIA

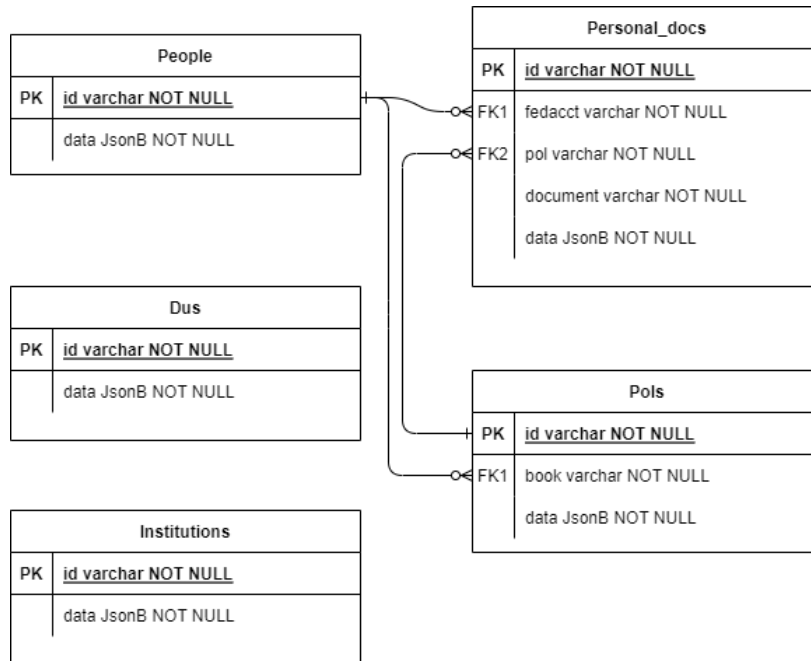
Memoria Realizada	Fecha de inicio	Fecha de fin	Horas Planificadas	Horas Reales	Justificación
Memoria	12/03/2024	17/05/2024	10	30	La investigación de varios puntos de la misma prolongó su realización.



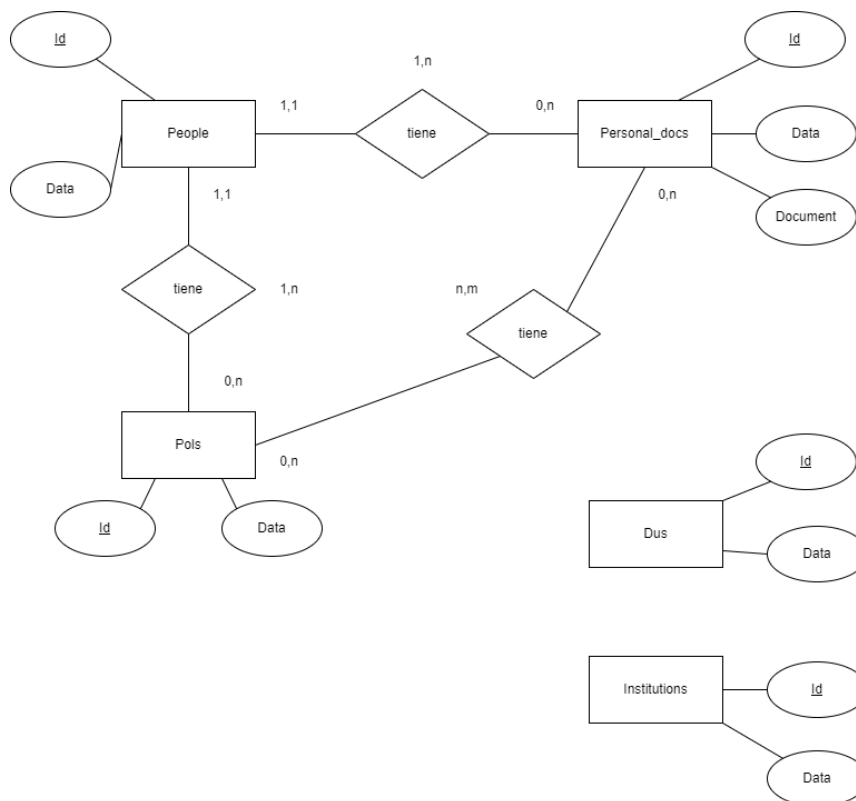
Diseño del proyecto.

Diseño de la BBDD.

Relacional



Entidad relación



Interacción con el usuario.

Cuenta de usuario:

- El usuario puede crearse una cuenta, en la cual posteriormente podrá iniciar sesión, y tendrá todos sus datos asociados a ella.

Gestión de datos manualmente:

- El usuario puede introducir, de manera manual, mediante campos de texto, spinners y seekbars, distintos tipos de datos médicos, que haya obtenido mediante mediciones manuales.

Gestión de datos automáticos:

- El usuario puede establecer una conexión con su dispositivo médico (Termómetro/Tensiómetro) para que, con un simple botón y de manera automática, la medición del dispositivo se comunique hacia la aplicación.

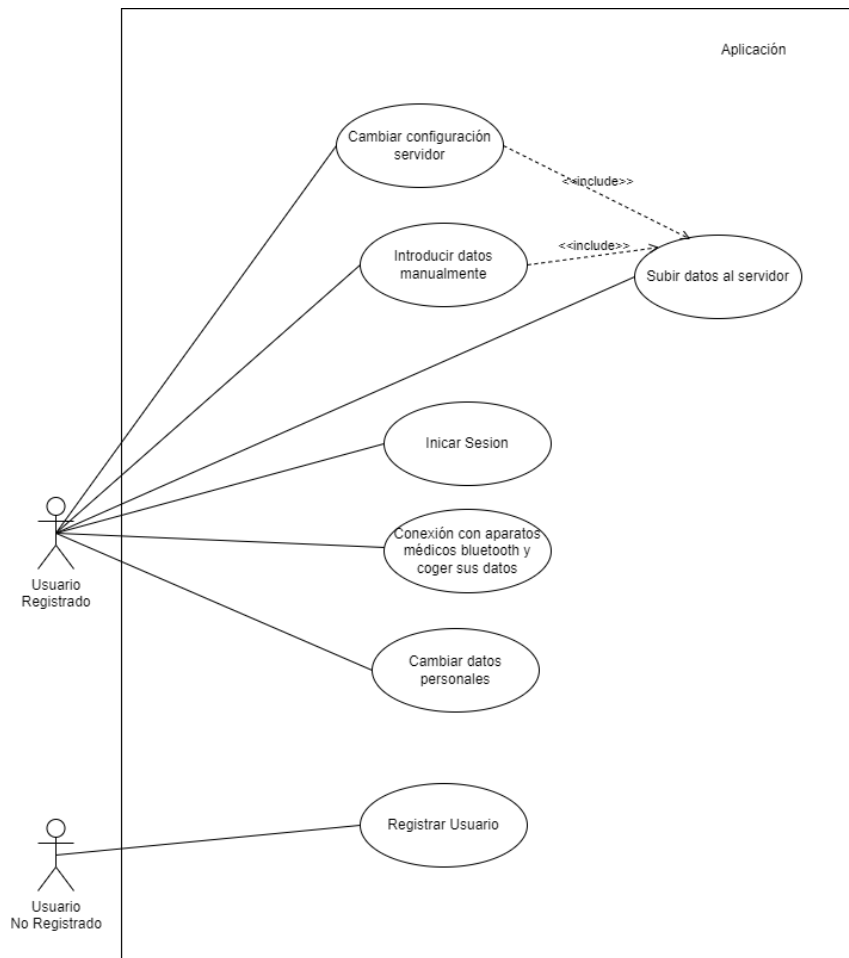
Datos en la nube:

- El usuario podrá enviar sus datos médicos, tanto los introducidos manualmente como los que se recogen de manera automática, a un servidor, mediante la simple pulsación de un botón.

Configuración:

- El usuario puede configurar los datos de su cuenta, como la altura o la cuenta de la federación, para obtener una experiencia más personalizada.
- El usuario puede configurar los datos de la conexión del servidor.

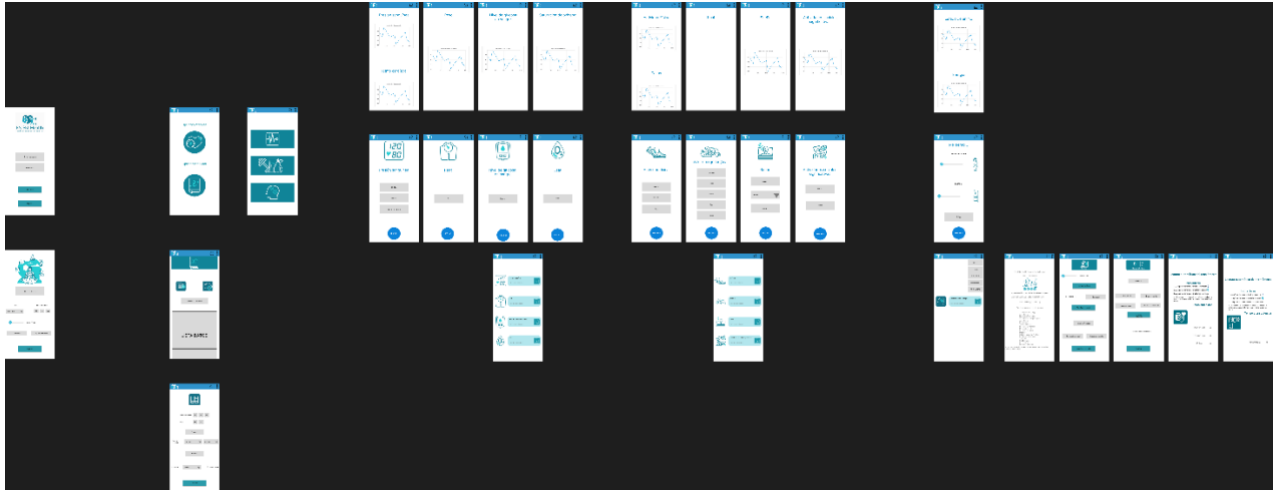
■ Casos de uso.



- Diseño de la interfaz.

Se mantienen los diseños originales de las imágenes, y las añadidas nuevas se intentan parecer a las ya existentes, para la gente que usase la aplicación de Kivy lo sienta familiar, el azul y el blanco son los colores corporativos (Predominantes)

Menús muy simples para navegar, sencillez para que sea fácil e intuitivo utilizar la aplicación, para acceder al Figma con el diseño de la aplicación y navegar de manera interactiva, pulse [aquí](#).



▪ Diseño de la arquitectura.

La arquitectura del proyecto se basa en el modelo cliente servidor, dado que el proyecto se basa en un cliente, que es el que recogerá los datos en la aplicación, y un servidor, que será el que va a recogerlos.

Ciente

El cliente es la parte del proyecto con la que el usuario interactúa directamente, utilizando la aplicación, desde un teléfono móvil, las características del cliente son las siguientes:

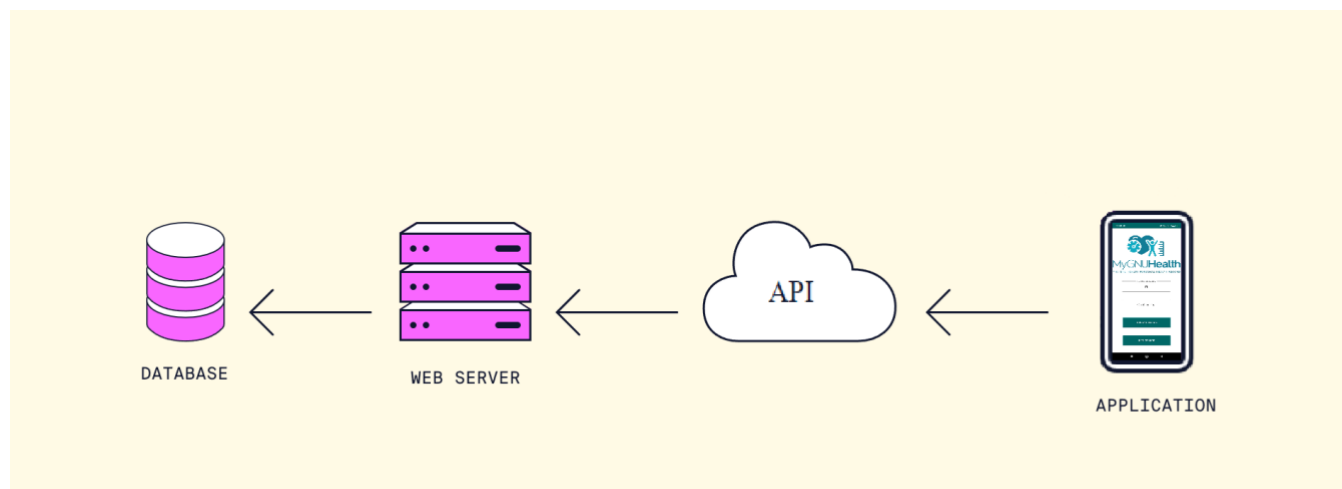
- Recolección de datos: Se obtendrán los datos que introduzca el usuario, tanto de manera manual como automática
- Procesamiento y almacenamiento de datos: Los datos se procesarán, y se almacenarán de manera local, para que, sin necesidad de servidor, los datos persistan incluso saliendo de la aplicación.
- Interfaz gráfica: Se muestran los datos y las opciones de una manera simple e intuitiva.
- Conexión con el servidor: Realiza las solicitudes correspondientes con el servidor, para el manejo de los datos.

Servidor

El servidor es la parte del proyecto que gestiona y almacena datos de una manera centralizada. No se ejecuta en el dispositivo que utiliza el usuario, es decir, es un servidor remoto, no local, tiene una serie de funciones:

- El servidor almacena todos los datos de todos los usuarios.
- Procesa las peticiones que envía el cliente y proporciona respuestas a esas solicitudes.

Resumen: Esta arquitectura permite separar de una manera clara las responsabilidades de cada parte, separando al cliente, mejorando el rendimiento y facilitando futuros mantenimientos. La estructura cliente-servidor facilita la implementación de nuevas funcionalidades y la integración con otros sistemas, asegurando una experiencia de usuario consistente y fiable.



- Arquitectura del servidor.

Desconocemos la arquitectura del servidor

- **Arquitectura del cliente.**

La arquitectura del cliente es un Model-View-ViewModel (MVVM), se utiliza este patrón para separar, de una manera clara, las funciones de cada parte, cada una tiene la siguiente función:

Arquitectura Clean:

Es una arquitectura que tiene dos características principales:

- Separación de capas: Es la característica principal, cada capa tiene una responsabilidad única, y son independientes entre sí, no depende las unas de las otras.
- Dependencia en una sola dirección: Las capas superiores no dependen de las capas inferiores, pero las inferiores, pueden proporcionar interfaces para que las utilicen las superiores.

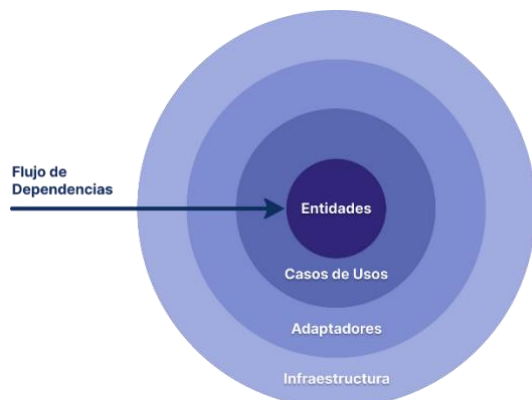
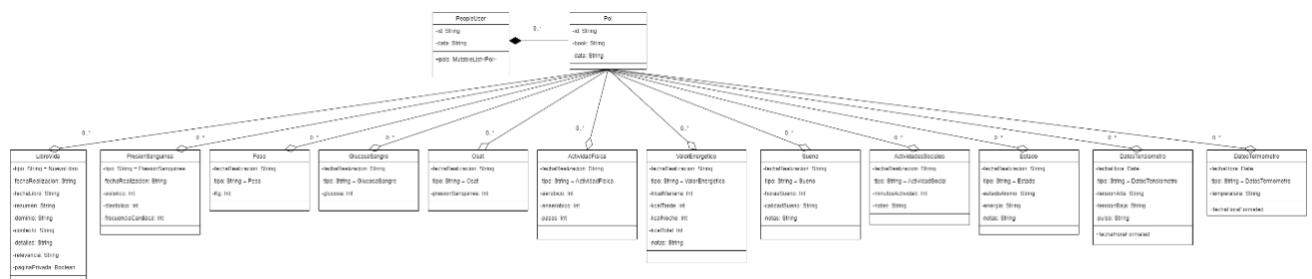


Diagrama UML de las clases de la aplicación:



La estructura de las carpetas del proyecto se puede ver en esta [imagen](#).

La documentación KDoc del cliente se encuentra alojada en este [página](#) web

- Relación con otros sistemas.

La aplicación se relaciona con los aparatos médicos facilitados, que son el termómetro y el tensiómetro mediante bluetooth, para que la relación se produzca correctamente, deben de ser el modelo exacto con el que nosotros hemos trabajado, ya que cuando realizamos la búsqueda de los dispositivos, necesitamos que coincida el nombre, o de otra manera la búsqueda no será satisfactoria. El termómetro envía las últimas 60 mediciones realizadas, pero como nos interesante únicamente la última, borramos el resto.

Cuando se conecta a un nuevo dispositivo, aparece un código de vinculación

Detalles de implementación.

Implementación Bluetooth (Termómetro)

A partir de la API 31, necesitas tener la ubicación activada para poder realizar el escaneo de los dispositivos bluetooth cercanos, además de solicitar una serie de permisos extra, por lo tanto, hay que incluirlos en el manifest. Los permisos bluetooth Scan y Connect son los nuevos a partir de la API 31, y los de "Location" son los relacionados con la ubicación

```
<uses-permission android:name="android.permission.BLUETOOTH_CONNECT" />
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.BLUETOOTH_SCAN" />
<uses-permission android:name="android.permission.INTERNET" />
```

Por lo tanto, hay que controlar el pedir unos permisos, u otros, dependiendo de si es anterior o posterior a la API 31

```
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
    }
}
```

Hay dos maneras de realizar la búsqueda, uno es por MAC, y otro es por nombre de dispositivo, hemos creído más conveniente la búsqueda por nombre ya que, buscando por MAC, que es un identificador único, si el dispositivo cambia, dejaría de encontrarlo, pero si se realiza la búsqueda por nombre, siempre y cuando sea el mismo modelo, seguirá encontrándolo, aunque el usuario cambie de dispositivo.

Trato y carga de datos de las conexiones Bluetooth

Por cómo funciona el termómetro, no envía solamente la última medición, esto es algo que puede llevar un tiempo, por lo que tenemos una barra de progreso, en lo que reciben las últimas 60 temperaturas.

El envío de datos se realiza como un array de bytes, tuvimos que mirar en foros y la página oficial del fabricante, para saber a qué dato correspondía cada valor del array.

El primer dato del array es uno que se utiliza comúnmente en protocolos bluetooth a modo de “FLAG”, luego, los 4 siguientes son la temperatura, dos para los valores enteros, y dos para los decimales, y los siguientes 7 son la fecha, por lo tanto, de esta manera, ya tenemos separados los datos que nos interesan, la temperatura y cuando se realizó la medición.

```
val temperatureBytes = data.copyOfRange(1, 5)
val timestampBytes = data.copyOfRange(5, 12)
```

No hemos averiguado realmente en que orden se envían los datos, pero no es por fecha de medición, así que tenemos que ordenarlo nosotros manualmente, y quedarnos con el más reciente.

Implementación Graficas

Las librerías que vienen incluidas por defecto en Android sobre gráficas eran demasiado simples, y no nos permitían mostrar los datos de la manera adecuada. Por lo tanto, decidimos utilizar una externa, “MPAndroidChart”, esta librería no se encuentra en el repositorio Maven, por lo tanto, hay que añadir “Jitpack.io” a la lista de repositorios en los que busque librerías, como se muestra en la imagen:

```

1  pluginManagement { this: PluginManagementSpec
2      repositories { this: RepositoryHandler
3          google()
4          mavenCentral()
5          gradlePluginPortal()
6          maven { url = uri(path: "https://jitpack.io") }
7          maven { url = uri(path: "https://maven.google.com") }
8      }
9  }
10 dependencyResolutionManagement { this: DependencyResolutionManagement
11     repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
12     repositories { this: RepositoryHandler
13         google()
14         mavenCentral()
15         maven { url = uri(path: "https://jitpack.io") }
16         maven { url = uri(path: "https://maven.google.com") }
17     }
18 }
19
20 rootProject.name = "ProyectoHospitalGambia"
21 include( ...projectPaths: ":app")
22

```

Una vez hecho esto, ya puedes implementar la librería de igual manera que se hace con el resto.

Cuando ya está instalada la librería, puedes usar sus elementos en los layouts, necesitábamos las de líneas para todas las gráficas, excepto la de nutrición que era una gráfica de barras, como tuvimos una serie de problemas para mostrar la fecha en el eje x de las gráficas, creamos una clase personalizada, que hereda de la que hemos implementado:

```

import com.github.mikephil.charting.charts.LineChart

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 que se muestre el texto “inicio de datos” arriba a la izquierda de la gráfica, y “Fin de datos” arriba a la derecha.



Así que, en los layout, utilizamos esta clase personalizada en vez de la otra, que a efectos prácticos nos servirá y podremos utilizar todas sus funciones, ya que hereda de ella

```
<com.example.proyectohospitalgambia.feature.CustomLineChart Clase personalizada
    android:id="@+id/graficoLineas_PresionSanguinea"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_weight="1"
    android:padding="16dp"
    app:layout_constraintBottom_toTopOf="@+id/tv_presionSanguinea"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/tv_BloodPressure" />
```

Tomaré de ejemplo la gráfica de Peso, para explicar cómo es su funcionamiento. En las gráficas se muestran únicamente los datos del usuario que está activo en ese momento, por lo tanto, en la clase lo primero será identificar que usuario tiene la sesión iniciada, obtener todos los datos del tipo de gráfica específica y guardarlos en una lista.

```
val idUsuarioActual = MainActivity.usuario?.id
val datosPeso = MainActivity.databaseHelper?.obtenerTodosLosDatosPeso(idUsuarioActual!!)
```

Una vez que tenemos la lista con los datos, lo guardaremos en el tipo "Entry" que es un tipo de dato que utiliza la librería que estamos usando de las gráficas para representar posteriormente los datos.

```
// Crear las entradas de la gráfica a partir de los datos de peso
val entriesPeso = datosPeso?.mapIndexed { index, peso ->
    Entry(index.toFloat(), peso.kg.toFloat())
}
```

La variable `entriesPeso` será una lista de “Entry”, cada uno de ellos contiene dos datos, su valor de eje x y su valor en el eje y.

Una vez que tenemos la lista con los datos, se ponen una serie de propiedades del gráfico, como el color de la línea, el color del texto, su tamaño, si se puede hacer zoom o desplazarte por la gráfica utilizando los dedos y finalmente el “`Invalidate`” para que recargue los datos.

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

// Crear el gráfico de líneas y personalizarlo
val dataWeight = LineData(dataSetPeso)
chartWeight.data = dataWeight
chartWeight.setTouchEnabled(true)
chartWeight.setPinchZoom(true)
chartWeight.description.text = "Peso"
chartWeight.setNoDataText("No hay datos disponibles")
chartWeight.invalidate()
```

También puedes elegir una propiedad para que muestre un texto en caso de que no haya datos disponibles

Inicio de datos

Últimos datos

No chart data available.

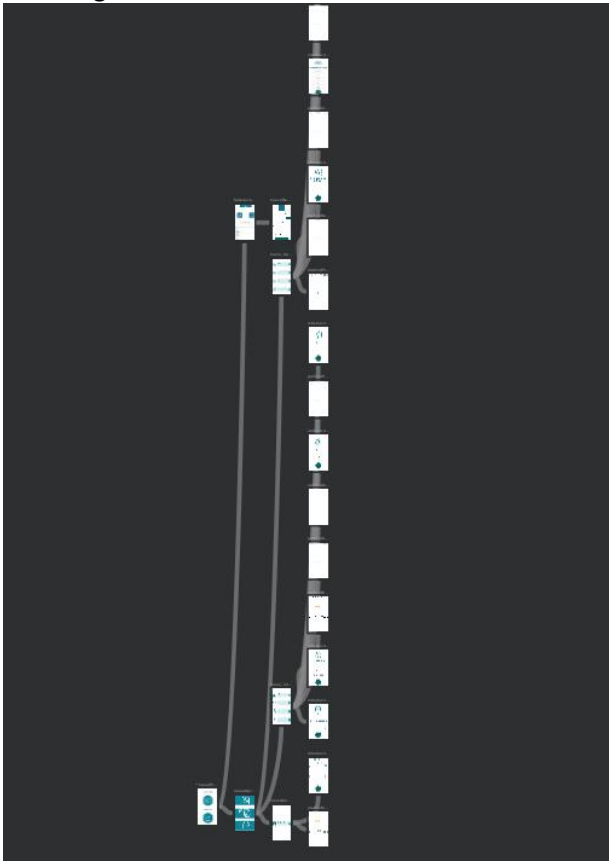
Implementación Barra de Navegación

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:

```
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, MainActivity::class.java)  
  
            intent.addFlags(Intent.FLAG_ACTIVITY_NO_ANIMATION)  
  
            // Iniciamos la actividad sin esperar un resultado.  
            startActivity(intent)  
  
            true  
        }  
    }  
}
```

Implementación de la navegación entre Fragments:

La manera en la que nos desplazamos entre los distintos fragments de nuestro proyecto es con el "navigation":



En el cual se establecen las relaciones de todos los fragments, indicando qué fragments están conectados entre ellos, para que luego, al utilizar, se pueda hacer de una manera sencilla:

```
btnOpcionMedica.setOnClickListener { it: View!
    findNavController().navigate(R.id.action_menuPrincipalView_to_menuOpcionesMedicasView)
}

btnOpcionServidor.setOnClickListener { it: View!
    findNavController().navigate(R.id.action_menuPrincipalView_to_federacionServidoresView)
}
```

Para que esto funcione correctamente, se generan una serie de acciones de manera automática al implementar este método:

```
<fragment
    android:id="@+id/menuOpcionesMedicasView"
    android:name="com.example.proyectohospitalgambia.feature.vistaMenuOpcionesMedicas.MenuOpcionesMedicasView"
    android:label="@string/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 de la recogida de Datos

Cada fragment para introducir los datos de manera manual, tiene un botón para enviar, lo que hace es recoger los datos de los "editText" rellenados. Con esos datos se crea una Pol, teniendo en cuenta el ID del usuario que está con la sesión iniciada en ese momento.

```
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?.pols?.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()
    }
} else {
    // Mostrar un mensaje de error
    Toast.makeText(context, "Por favor complete todos los campos", Toast.LENGTH_SHORT).show()
}
```

Y con esas “Pols” que tenemos, se suben a la BBDD local

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

Implementación de la Subida de Datos

Hubo un problema inicial a la hora de realizar la petición de subida de datos al servidor, y es que la rechazaba, si al inicio no seguía una estructura clara, la petición iba a ser rechazada, por lo que tuvimos que incluir esa estructura, y luego ya pudimos incluir nuestros datos:

```
// 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()
```

Para subir los datos, se realiza en una corrutina, ya que si subes muchos datos es una operación que puede tomar un poco de tiempo, con una corrutina se evita que se pueda quedar bloqueada la pantalla.

```
viewModelJob = CoroutineScope(Dispatchers.IO).launch { this: CoroutineScope
    var fallo = false

    for (pol in polys) {
        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 sube los datos según la URL especificada, manejando los escenarios en los que pueda dar error, para que devuelva false.

```

fun insertarDatosEnServidorAsync(polJsonString: String): Boolean {
    return runBlocking { this: CoroutineScope
        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("ESPGNU7770RG", "gnusolidario"))
                .build()

            val response = withContext(Dispatchers.IO) { this: CoroutineScope
                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("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
        }
    }
}

```

Cuando ese método termina, se muestra un diálogo en otro hilo, para que el usuario tenga la información correspondiente sobre si se ha realizado o no con éxito, y se actualiza la lista, para que se muestran todos los datos del usuario que tienen la sesión activa.

```

// Mostrar el diálogo apropiado en el hilo principal después de que todas las solicitudes se hayan completado
withContext(Dispatchers.Main) { this: CoroutineScope
    // Hacer visible el ProgressBar
    progressBarSubirDatos.visibility = View.INVISIBLE

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

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

```

Un añadido que hicimos posteriormente fue controlar la disponibilidad de acceso a internet, ya que, al ser un servidor remoto, se necesita internet para comunicarte con él, por lo que hacemos la comprobación en este método:

```
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
}

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
}
```

También tenemos una variable para controlar el hilo, ya que podía dar errores al salir de la aplicación (OnPause) mientras se realizaba la subida de datos

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

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

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

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


Fase de pruebas.

Pruebas de unidad.

Para las pruebas unitarias se han usado las librerías de Junit y Mock, un mock es un objeto simulado que imita el comportamiento de un objeto real en un entorno controlado. En las pruebas unitarias, los mocks se utilizan para simular el comportamiento de las dependencias de una clase, permitiendo así aislar la clase que estamos probando y verificar su comportamiento de forma independiente.

Funcionamiento resumido de “Mock”

- **Configuración del mock:** Se define el comportamiento esperado del mock para ciertos métodos o acciones.
- **Uso del mock:** Se utiliza el mock en lugar de la implementación real durante la ejecución de la prueba.
- **Verificación del comportamiento:** Se verifica que el objeto bajo prueba interactúe correctamente con el mock según lo esperado

Esa es la estructura básica que sigue una prueba que utiliza “Mock”, pero ahora se va a explicar más en profundidad cada una de las pruebas:

Esta clase contiene pruebas unitarias para validar el funcionamiento de los métodos en la clase **DatabaseHelper**.

```

└ Mario Muñoz
@Test
fun testInsertarPersona() {
    // Crear un objeto PeopleUser para la prueba
    val peopleUser = PeopleUser(id: "1", data: "data")

    // Configurar el comportamiento del mock
    // Aquí estamos diciendo que cuando el método insertarPersona() sea llamado con el objeto peopleUser, debe devolver true
    every { databaseHelper.insertarPersona(peopleUser) } returns true

    // Llamar al método que estamos probando
    val result = databaseHelper.insertarPersona(peopleUser)

    // Verificar que el resultado es el esperado
    assert(result)

    // Verificar que el método insertarPersona() fue llamado con el objeto peopleUser
    verify { databaseHelper.insertarPersona(peopleUser) }
}

```

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**.

```
new *
@Test
fun testInsertarPersonaConDatosInvalidos() {
    // Crear un objeto PeopleUser con datos inválidos para la prueba
    val peopleUserInvalido = PeopleUser(id: "", data: "")

    // Configurar el comportamiento del mock
    // Aquí estamos diciendo que cuando el método insertarPersona() sea llamado con el objeto peopleUserInvalido, debe devolver false
    every { databaseHelper.insertarPersona(peopleUserInvalido) } returns false

    // Llamar al método que estamos probando
    val result = databaseHelper.insertarPersona(peopleUserInvalido)

    // Verificar que el resultado es el esperado
    assert(!result)

    // Verificar que el método insertarPersona() fue llamado con el objeto peopleUserInvalido
    verify { databaseHelper.insertarPersona(peopleUserInvalido) }
}
```

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.

```
new *
@Test
fun testVerificarCredenciales() {
    // Crear un nombre de usuario y una contraseña para la prueba
    val nombreUsuario = "testUser"
    val contraseniaUsuario = "testPassword"

    // Aquí estamos diciendo que cuando el método verificarCredenciales() sea llamado con el nombreUsuario y contraseniaUsuario
    // debe devolver un objeto PeopleUser
    val peopleUser = PeopleUser(nombreUsuario, contraseniaUsuario)
    every { databaseHelper.verificarCredenciales(nombreUsuario, contraseniaUsuario) } returns peopleUser

    // Llamar al método que estamos probando
    val result = databaseHelper.verificarCredenciales(nombreUsuario, contraseniaUsuario)

    // Verificar que el resultado es el esperado
    assertEquals(peopleUser, result)

    // Verificar que el método verificarCredenciales() fue llamado con el nombreUsuario y contraseniaUsuario
    verify { databaseHelper.verificarCredenciales(nombreUsuario, contraseniaUsuario) }
}
```

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.

```
new *
@Test
fun testObtenerTodosLosDatosPesoCorrecto() {
    // Crear datos de prueba
    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)

    // Configurar el comportamiento del mock
    every { databaseHelper.obtenerTodosLosDatosPeso( idUsuario: "1") } returns datosPrueba

    // Llamar a la función que estamos probando
    val result = databaseHelper.obtenerTodosLosDatosPeso( idUsuario: "1")

    // Verificar que el resultado es el esperado
    assertEquals(datosPrueba, result)

    // Verificar que la función obtenerTodosLosDatosPeso() fue llamada con el id de usuario correcto
    verify { databaseHelper.obtenerTodosLosDatosPeso( idUsuario: "1") }
}
```

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.

```

new *
@Test
fun testObtenerTodosLosDatosPesoIncorrecto() {
    // Configurar el comportamiento del mock para devolver una lista vacía
    every { databaseHelper.obtenerTodosLosDatosPeso( idUsuario: "1" ) } returns emptyList()

    // Llamar a la función que estamos probando
    val result = databaseHelper.obtenerTodosLosDatosPeso( idUsuario: "1" )

    // Verificar que el resultado es una lista vacía
    assertTrue(result.isEmpty())

    // Verificar que la función obtenerTodosLosDatosPeso() fue llamada con el id de usuario correcto
    verify { databaseHelper.obtenerTodosLosDatosPeso( idUsuario: "1" ) }
}

```

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.

```

new *
@Test
fun testInsertFormData() {
    val pol = Pol( idPol: "1", book: "book", data: "data", isSubido: "subido" )

    // Configurar el comportamiento del mock
    every { databaseHelper.insertFormData(pol) } returns true

    // Llamar al método que estamos probando
    val result = databaseHelper.insertFormData(pol)

    // Verificar que el resultado es el esperado
    assertTrue(result)

    // Verificar que el método insertFormData() fue llamado con el objeto pol
    verify { databaseHelper.insertFormData(pol) }
}

```

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 devuelva **true**.
- Verifica que el método **insertFormData()** fue llamado con el objeto **Pol**.

```
new *
@Test
fun testListarPols() {
    // Crear una lista de pols de prueba
    val expectedPols = listOf(Pol(idPol: "id1", book: "book1", data: "data1", isSubido: "subido"), Pol(idPol: "id2", book: "book2", data: "data2", isSubido: "subido"))

    // Configurar el comportamiento del mock
    every { databaseHelper.obtenerPols() } returns expectedPols

    // Llamar a la función que estamos probando
    val pols = databaseHelper.obtenerPols()

    // Verificar que la lista de pols no sea nula
    Assert.assertNotNull(pols)

    // Verificar que la lista de pols tiene el tamaño esperado
    Assert.assertEquals("El tamaño de la lista de pols no es el esperado", expectedPols.size, pols.size)

    // Verificar que la lista de pols contiene los elementos esperados
    Assert.assertTrue("La lista de pols no contiene los elementos esperados", pols.containsAll(expectedPols))
}
```

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.

Al ejecutar la clase que contiene las pruebas, todas se realizan de manera correcta, y dan el resultado esperado.

```
✓ Tests passed: 7 of 7 tests – 2 sec 432 ms
> Task :app:testDebugUnitTest
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 5s
24 actionable tasks: 2 executed, 22 up-to-date
```

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

La clase DatabaseHelper no tiene asociada ningún elemento visual en ella, no pertenece a ninguna activity ni a ningún fragment, lo cual nos ha permitido realizar pruebas sin problemas en ella. Sin embargo, nos encontramos con problemas a la hora de realizar pruebas para el resto de clases que sí contienen elementos visuales.

¿Por qué sucede?

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

Una posible solución podría ser implementar uno de los siguientes frameworks, que están creados específicamente para trabajar con clases que tienen elementos visuales.

1. **Robolectric:** Aunque principalmente se usa con Java, también es compatible con Kotlin y te 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.

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

Pruebas de usabilidad.

Aspecto	Observaciones
Número de clics	Para tareas como rellenar 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, dificultando 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, dificultando 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.

Pruebas de integración del sistema.

La aplicación final se ha probado en los siguientes dispositivos:

- **Xiaomi Redmi note 10 pro - API 30**
 - La aplicación se adapta correctamente a la pantalla de 6,43 pulgadas
 - No ha habido ningún problema a la hora de conectarse con el dispositivo bluetooth
- **Xiaomi Redmi note 12- API 34**
 - La aplicación se muestra correctamente en la pantalla de 6,67 pulgadas.
 - No tiene problemas al conectarse con los dispositivos bluetooth, pero al ser moderno requiere tener la ubicación activada
- **Emulador Píxel 31- API 34**
 - La aplicación se adapta correctamente a la pantalla de 5,6 pulgadas.
 - No tiene problemas al conectarse con los dispositivos bluetooth, pero al ser moderno requiere tener la ubicación activada
- **OPPO A9 2020- API 30**
 - La aplicación se adapta bien, pero al tener un ratio de resolución distinto, algunos elementos están demasiado cerca de los bordes de la pantalla
 - Al conectarse con el termómetro, salta el aviso, indicando que necesita un número de emparejamiento, ese número se muestra en la pantalla del termómetro


Manual de usuario

Si es la primera vez que se inicia la aplicación, la única opción disponible es crear un nuevo usuario para acceder al resto de funciones, dando al botón de registrarse.



The login screen features the MyGNUHealth logo at the top, which includes a stylized heart with a clock face and a person icon. Below the logo, the text "MyGNUHealth" is displayed in a large, bold font, followed by the tagline "THE GNU HEALTH PERSONAL HEALTH RECORD" in a smaller font. The form consists of two input fields: "Nombre de Usuario" (Username) and "Contraseña" (Password). Below these fields are two buttons: "INICIAR SESION" (Login) and "REGISTRARSE" (Register).

En la pantalla de registro, introduces los datos del usuario que quieras crear, el nombre y la contraseña serán los que se usarán para entrar a la aplicación



The registration screen features a large illustration at the top showing a person meditating, surrounded by various health-related icons like a heart, a brain, a DNA helix, and a person. Below the illustration is a form with several fields: "Nombre de Usuario" (Username), "Sexo" (Gender) with a dropdown menu showing "Masculino", "Fecha Nacimiento" (Date of Birth) with three input fields for "dd", "mm", and "yyyy", and "Altura: 50 cm" (Height) with a slider and a text input. Below these fields are two input fields for "Contraseña" (Password) and "Repetir Contraseña" (Repeat Password). At the bottom is a "REGISTRARSE" (Register) button and the MyGNUHealth logo with the tagline "The GNU Health Personal Health Record".

Si ya tienes un usuario, puedes iniciar sesión y acceder al resto de funciones de la aplicación, primero se mostrará una pantalla para elegir datos médicos o datos del servidor, hablaremos primero de los datos médicos, este es uno de los menús con los que puedes acceder tanto a las pantallas para introducir los datos de manera manual pulsando los iconos de la izquierda, como a la visualización de gráficas pulsando en los iconos del a derecha:



A continuación, se muestran un par de pantallas de ejemplo en las que introducir los datos de manera manual



Sueño

Horas

Buena

Notas

HECHO



Presión Sanguínea

Sistólico

Diastólico

Frecuencia Cardíaca

HECHO

Estos datos introducidos se pueden ver de una manera gráfica, en el menú mencionado anteriormente, pulsando el botón del tipo de dato que quieras visualizar, y los datos se verán a modo de gráfica interactiva:




Por otro lado, en las opciones del servidor se verá esta pantalla, en la que pulsando el botón de la izquierda veremos una pantalla relacionada con el libro de vida, un botón para sincronizar los datos, que lo que hará será subir los que no se hayan subido, y mostrar en la lista de abajo lista los datos locales:

The screenshot shows a server options screen with a header bar, a 'Libro de Vida' button, a 'Sincronizar' button, a 'Contraseña Federación' input field, and a table of local data.

Fecha	Evento	Valor
2024-05-04 23:33:46	PresionSanguinea	Sistolico:4 Diastolico:4 FrecuenciaCardiaca:4
2024-05-04 23:42:45	PresionSanguinea	Sistolico:4 Diastolico:4 FrecuenciaCardiaca:7
2024-05-05 11:33:15	temperatura	temperatura:36.7

La pantalla mencionada anteriormente sobre el libro de vida es la siguiente:







Fecha

dd

mm

yyyy

Hora

hour

min

Resumen

Dominio y contexto

Medico

Normal

Detalles


Relevancia



Condición de s...

☐ Pagina privada

GUARDAR

La aplicación tiene una toolbar, con un botón con forma de casa, para volver a la pantalla de inicio, y un botón que permite a acceder a una serie de opciones:





Perfil

Conexion

About

Datos Tensiometro

Datos Termometro

Salir

La opción perfil para modificar los datos de usuario, como la altura o la contraseña



Profile

Altura: 50cm

ACTUALIZAR ALTURA

Cuenta Federacion

Cuenta Federacio...

ACTUALIZAR FEDERACION

Contraseña Personal Actual

Repetir Contraseñ...

Contraseña Perso...

ACTUALIZAR CONTRASEÑA

La opción conexión, que permite modificar los parámetros del servidor y realizar una prueba de conexión



Network Settings

Protocolo

Thalamus Host

Thalamus Puerto...

Federacion ID

Contraseña Feder...

PRUEBA CONEXION

☐ Sincronización con federacion

ACTUALIZAR

La opción About para mostrar información sobre la aplicación, y la personas que la han creado



Finalmente, las opciones de termómetro y tensiómetro, que son las que te permiten realizar la conexión con el dispositivo médico 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

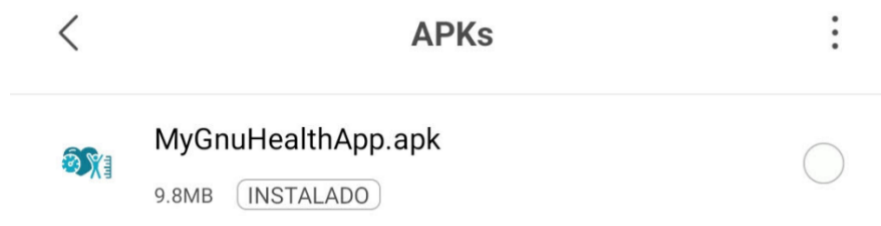
Pulso 0

Manual de Administrador

- Instalación:
 - Descarga la aplicación APK desde la ubicación proporcionada, a través del código QR.



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



- Toca el archivo APK para iniciar el proceso de instalación.
 - Sigue las instrucciones en pantalla para completar la instalación.
- 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.

Trabajo futuro.

El desarrollo móvil avanza constantemente, solo hay que ver cómo, con tan solo retroceder unas pocas versiones, los métodos empiezan a quedar obsoletos, por lo tanto, sería necesario una actualización constante en la aplicación, para asegurarse que no se queda obsoleta, y comprobar que la funcionalidad no falla. Además, podrían hacerse una serie de añadidos:

- **Conexión con más dispositivos médicos:** Actualmente la mayoría de datos se introducen de manera manual, y los únicos dispositivos permitidos para obtener datos de manera automática son el termómetro y tensiómetro, por lo que sería interesante que introducirlos de manera manual sea algo secundario, y el usuario tenga la posibilidad de utilizar la aplicación únicamente con distintos dispositivos médicos, permitiendo todas las mediciones de manera automática.
- **Mejorar gestión de usuarios:** Tener un sistema de usuarios con diferentes roles, que tengan acceso a diferentes ventanas, por ejemplo, pacientes, personal médico, gestores, etc. Para que el sistema tenga una mejor organización.
- **Sistema de citas online:** Permitir a los usuarios programar citas con su médico, para aprovechar el sistema de usuarios, y que, si alguna medición está en un nivel peligroso, que se notifique de manera automática a su centro de salud.
- **Mejoras de seguridad de acceso:** Ya que los datos médicos son sensibles, habría que mejorar la seguridad, tanto del acceso a la aplicación como de la BBDD
- **Mejoras en la interfaz de usuario:** Realizar mejoras ya que, algunas pantallas podrían no ser todo lo intuitivas que deberían

Acceso a la descarga de la aplicación



Acceso al código fuente del proyecto

[GitHub](#)

Conclusiones.

En colaboración con otro compañero, he experimentado un viaje de aprendizaje y crecimiento, tanto personal como profesional, durante el desarrollo de este proyecto solidario. En este camino, nos hemos enfrentado a diversos desafíos tecnológicos que han sido fundamentales para nuestro desarrollo.

Desde el principio, la investigación y la adquisición de conocimientos sobre nuevas tecnologías, como la conexión con dispositivos médicos y la gestión de servidores, han sido cruciales para el éxito de la aplicación. Esta experiencia ha fortalecido nuestras habilidades de resolución de problemas y nuestra capacidad para adaptarnos a entornos tecnológicos en constante cambio.

Además, el trabajo en equipo y la colaboración estrecha han sido elementos esenciales en el proceso. La gestión de versiones y la colaboración en equipo a través de Git han sido especialmente valiosas y representan un paso importante en nuestro desarrollo como desarrolladores de software colaborativo.

En resumen, este proyecto solidario ha sido una experiencia transformadora que no solo ha contribuido al desarrollo de nuestras habilidades técnicas, sino que también nos ha brindado la oportunidad de impactar positivamente en la sociedad. La colaboración con una ONG para abordar necesidades de salud en un país necesitado ha sido gratificante y ha resaltado el potencial del desarrollo tecnológico para generar un cambio significativo en el mundo.

Bibliografía

Youtube

(AristiDevs, Navega en Android de forma eficiente, 2024)

(AristiDevs, MVVM en ANDROID, 2024)

(InnovaDomotics, Comunicación Bluetooth con Android Studio, 2024)

(InnovaDomotics, Comunicación Bluetooth con Android Studio Soporte - Parte 1, 2024)

(dev.xcheko51x, 2024)

(Coders, Android Charts | Bar Chart | MP Android Chart, 2024)

(Coders, Custom Buttons Design, 2024)

(MoureDev, 2024)

Foros

(Emil, 2024)

(Soporte oficial Android, 2024)

(Hernández, 2024)

(jsgalarraga, 2024)

(Mike, 2024)

(Soporte oficial Android, 2024)