Centro Universitario de la Defensa
en la Escuela Naval Militar

TRABAJO FIN DE GRADO

Sistema automatizado para el seguimiento médico de pacientes en evacuaciones militares

Grado en Ingeniería Mecánica

ALUMNO: Pablo García Rodríguez

DIRECTORES: Belén Barragáns Martínez
Pablo Sendín Raña

CURSO ACADÉMICO: 2023-2024

Universidad de Vigo

Centro Universitario de la Defensa en la Escuela Naval Militar

TRABAJO FIN DE GRADO

Sistema automatizado para el seguimiento médico de pacientes en evacuaciones militares

Grado en Ingeniería Mecánica

Intensificación en Tecnología Naval

Cuerpo General

Universidade Vigo

RESUMEN

La atención médica a heridos en zona de operaciones sigue el estándar de la OTAN que define los procedimientos para la atención de bajas en combate. Dicho estándar establece la información mínima (médica y personal de cada herido) recogida en la denominada *NATO Field Medical Card* (NFMC) que, en la actualidad, se rellena en papel y se vincula físicamente al herido.

La transmisión de la información del paciente por este medio presenta múltiples inconvenientes (degradación del soporte, dificultad de entender la caligrafía, etc.). Adicionalmente, la doctrina sanitaria en operaciones señala que la utilización de las TIC resulta crucial para asegurar la trazabilidad de la información de cada paciente a lo largo de la cadena asistencial.

Para dar solución a estas necesidades, este Trabajo Fin de Grado ha diseñado y desarrollado una aplicación móvil para dispositivos Android que genera la información de las NFMC y la transfiere a una tarjeta NFC (*Near Field Communication*) que portaría el herido, así como a un servidor (lo que permite que todas las unidades de atención médica puedan consultar dichos datos). Dicha aplicación se ha validado mediante un caso de uso que demuestra que satisface los requisitos de diseño establecidos al inicio de este TFG.

PALABRAS CLAVE

NFC (*Near Field Communication*), NFMC (*NATO Field Medical Card*), Android, Kotlin, Evacuaciones

AGRADECIMIENTOS

En primer lugar, me gustaría agradecer a mis padres y a mi hermana el apoyo incondicional que me han dado siempre que lo he necesitado, ya sea en relación con la elaboración de este trabajo o en cualquier aspecto de la vida. A lo largo de esta me he desviado del camino unas cuantas veces y ellos han estado siempre presentes para, no solo recomendarme lo que hacer, sino para escucharme cuando ya era demasiado tarde y, efectivamente, mi decisión no había sido la más adecuada.

Por otro lado, me gustaría agradecer a mis compañeros de promoción los buenos momentos que he pasado, ya que, a lo largo de estos cinco años, muchos de ellos más que compañeros se han convertido en familia. Durante este tiempo ha habido risas, lamentos, sustos, etc., sin embargo, algo que guardan todos estos recuerdos en común es que han sido compartidos prácticamente siempre con vosotros. De nuevo, gracias.

Finalmente, me gustaría agradecer la paciencia que han tenido mis tutores, Belén y Pablo, a la hora de realizar todas las reuniones y consultas que me iban surgiendo a lo largo del trabajo. Ambos han estado en todo momento buscando huecos en su apretada agenda para hacer este proyecto lo más cómodo y llevadero posible.

CONTENIDO

Contenido 1
Índice de Figuras 3
Índice de Tablas..........7
1 Introducción y objetivos..........9
1.1 Contexto y motivación..........9
1.2 Objetivos del TFG..........10
1.3 Organización de la memoria 11
2 Estado del arte 13
2.1 Contexto de la sanidad militar 13
2.1.1 Historia de la sanidad militar..........13
2.1.2 Sanidad militar en operaciones..........14
2.2 Estándares que rigen las evacuaciones militares..........17
2.2.1 Protocolo TCCC 17
2.2.2 Tarjetas TCCC..........18
2.2.3 NATO Field Medical Card (NFMC) 19
2.3 Tecnología NFC..........21
2.3.1 Historia 21
2.3.2 Características..........22
2.3.3 Uso de la tecnología NFC..........22
2.3.4 Modos de funcionamiento 22
2.3.5 Tarjetas NFC..........23
2.4 Selección de tecnologías (hardware y software) a emplear 23
2.4.1 Desarrollo de una aplicación móvil 23
2.4.2 Selección del sistema operativo..........24
2.4.3 Empleo del IDE de desarrollo y lenguaje de programación 26
2.4.4 Recursos hardware..........30
2.5 Trabajos previos..........32
2.5.1 Proyecto e-SafeTag..........32
3 Desarrollo del TFG..........35
3.1 Requisitos de la aplicación..........35
3.1.1 Análisis de soluciones planteadas..........35
3.2 Diseño de la aplicación 38
3.3 Desarrollo de la aplicación..........39
3.3.1 Estructura del proyecto Android..........41

3.3.2 Conexión con el servidor 48
3.3.3 Implementación de las diferentes actividades 49
4 Resultados y validación de la aplicación 77
4.1 Descripción del escenario de prueba 77
4.1.1 Conjunto de pruebas realizadas 78
4.2 Capacidad de las tarjetas NFC 90
5 Conclusiones y líneas futuras 93
5.1 Conclusiones 93
5.2 Líneas futuras 94
6 Bibliografía 95
Anexo I: Implicaciones sociales, económicas, y ambientales 99
Anexo II: Reflexiones éticas y sociales 101
Anexo III: Código de la aplicación DIGITAL NFMC 103

ÍNDICE DE FIGURAS

Figura 1-1. Ejemplo de NFMC [3]....10
Figura 2-1. Competencias asociadas a cada nivel de capacitación [11] 15
Figura 2-2. Composición del CoTCCC en 2002 [1] 18
Figura 2-3. Tarjeta TCCC [13] 19
Figura 2-4. *NATO Field Medical Card* [6] 21
Figura 2-5. Usos de NFC [17]....22
Figura 2-6. Gráfico comparativo del uso de diferentes sistemas operativos para dispositivos móviles [19] 24
Figura 2-7. Capas de Android [20] 26
Figura 2-8. Vista de Android en Android Studio [fuente propia] 27
Figura 2-9. Vista Project [fuente propia] 28
Figura 2-10. Ciclo de vida de una actividad [23]....29
Figura 3-1. Esquema de la primera configuración propuesta [fuente propia] 36
Figura 3-2. Esquema de la segunda configuración propuesta [fuente propia]....36
Figura 3-3. Esquema de la tercera configuración propuesta [fuente propia] 37
Figura 3-4. Esquema que muestra el flujo de información entre actividades [fuente propia] 38
Figura 3-5. Pantallas de Inicio de sesión y menú principal [fuente propia]....39
Figura 3-6. Asociación de las versiones de Android con sus niveles API [31] 40
Figura 3-7. Gráfico que representa el porcentaje de uso de las diferentes versiones del sistema operativo Android [32]....40
Figura 3-8. Contenido del fichero *AndroidManifest.xml* de la aplicación [fuente propia] 42
Figura 3-9. Actividades de la aplicación [fuente propia]....44
Figura 3-10. Vista diseño del *layout* de *InicioSesion* [fuente propia]....46
Figura 3-11. Parte del código del *layout* de *InicioSesion* [fuente propia]....46
Figura 3-12. Dependencias de la aplicación [fuente propia] 48
Figura 3-13. Dependencias añadidas para garantizar la conexión con el servidor [fuente propia] 48
Figura 3-14. Código añadido para conceder acceso a Internet a la aplicación [fuente propia] 48
Figura 3-15. Configuración de la seguridad de la red [fuente propia] 49
Figura 3-16. Código relacionado con la interfaz API [fuente propia] 49
Figura 3-17. Código asociado a la actividad de *InicioSesion* [fuente propia] 50
Figura 3-18. Parte del código de la actividad *MenuPrincipal* [fuente propia] 50
Figura 3-19. Esquema de cómo se envían y reciben datos entre las actividades y el menú principal [fuente propia] 51
Figura 3-20. Función *Navegar_basico()* de la actividad *MenuPrincipal* [fuente propia] 51

Figura 3-21. Función *getResultDatosBasicos()* [fuente propia] 52
Figura 3-22. Función *Navegar_causa()* [fuente propia] 52
Figura 3-23. Parte del código asociado a la función *getResultCausa()* [fuente propia] 53
Figura 3-24. Código asociado a la función *Navegar_constantes()* [fuente propia] 53
Figura 3-25. Código asociado a la función *getResultConstantes()* [fuente propia] 54
Figura 3-26. Parte del código de *leerNFC()* asociada a la interacción con la tarjeta NFC [fuente propia 54
Figura 3-27. Parte del código de *leerNFC()* asociada a la construcción del *StringBuilder* [fuente propia] 55
Figura 3-28. Parte del código de *leerNFC()* asociada a los registros de datos [fuente propia] 55
Figura 3-29. Parte del código de *escribirNFC()* encargado de comprobar los ID [fuente propia] 56
Figura 3-30. Parte del código de la función *mostrarDialogoConfirmacion()* [fuente propia] 56
Figura 3-31. Parte del código de la función *mostrarDialogoConfirmacion()* encargada de la opción afirmativa [fuente propia] 56
Figura 3-32. Parte del código de la función *mostrarDialogoConfirmacion()* encargada de la opción negativa [fuente propia] 57
Figura 3-33. Parte del código asociado a la función *enviarServidor()* [fuente propia] 57
Figura 3-34. Código de *enviarServidor()* encargado de mostrar una respuesta en función de si la conexión fue exitosa o no [fuente propia] 58
Figura 3-35. Datos básicos añadidos a través de la función *construirMensajeNFC()* [fuente propia]...58
Figura 3-36. Coordenadas de la imagen fractura a través de la función *construirMensajeNFC()* [fuente propia] 59
Figura 3-37. Parte del código del método *onCreate* de la actividad *DatosBasicos* [fuente propia].......59
Figura 3-38. Parte del código de la función *onBackPressed()* [fuente propia].....................................60
Figura 3-39. Primera comprobación realizada por la función *onBackPressed()* [fuente propia] 60
Figura 3-40. Segunda comprobación realizada por la función onBackPressed() [fuente propia] 60
Figura 3-41. Validación de la fecha en la función *onBackPressed()* [fuente propia]............................60
Figura 3-42. Mensaje de error mostrado [fuente propia]..61
Figura 3-43. Obtención del texto ingresado en los campos de los datos básicos [fuente propia]..........61
Figura 3-44. Valores originales que se obtuvieron del intento de la actividad anterior [fuente propia].61
Figura 3-45. Comparación de los datos originales y los datos obtenidos [fuente propia] 62
Figura 3-46. Verificación de que el día, mes y año están en un rango válido [fuente propia] 62
Figura 3-47. Determinación de la cantidad de días que tiene el mes [fuente propia]............................62
Figura 3-48. Comprobación de si el año es bisiesto [fuente propia]..63
Figura 3-49. Esquema de flujo de trabajo seguido en la actividad Causa [fuente propia].....................63
Figura 3-50. Método *onCreate* de la actividad *Causa* [fuente propia] 63
Figura 3-51. Obtención de las coordenadas de la imagen en la función *recuperar_fractura()* [fuente propia] 64

Figura 3-52. Recuperación del contador y obtención del array en la función *recuperar_fractura()* [fuente propia] 64

Figura 3-53. Iteración sobre el array de coordenadas [fuente propia] 65

Figura 3-54. Lógica asociada al botón de eliminación [fuente propia] 65

Figura 3-55. Parte del código asociado a la función *setupImageViewListeners()* [fuente propia] 66

Figura 3-56. Parte responsable del "*Action Down*" del *handleTouchEvent* [fuente propia] 66

Figura 3-57. Parte responsable del "*Action Move*" del *handleTouchEvent* [fuente propia] 67

Figura 3-58. Parte responsable del "*Action Up*" del *handleTouchEvent* [fuente propia] 67

Figura 3-59. Parte inicial del código asociada a la función *createNewImage1()* [fuente propia] 67

Figura 3-60. Código asociado al establecimiento de color de la imagen [fuente propia] 68

Figura 3-61. Asignación de un identificador y una etiqueta [fuente propia] 68

Figura 3-62. Esquema representativo del flujo de trabajo de la actividad *Evaluación* y *Tratamiento* [fuente propia] 68

Figura 3-63. Parte del código encargado de recuperar los valores de diferentes variables [fuente propia] 69

Figura 3-64. Código de la función *obtenerHoraActual()* [fuente propia] 69

Figura 3-65. Código de la función *obtenerFechaActual()* [fuente propia] 69

Figura 3-66. Establecimiento de los valores recuperados en los *EditText* correspondientes [fuente propia] 69

Figura 3-67. Código asociado a la visibilidad de los formularios [fuente propia] 70

Figura 3-68. Método para alternar la visibilidad del formulario [fuente propia] 70

Figura 3-69. Método para verificar si una fecha es posterior a otra [fuente propia] 70

Figura 3-70. Constantes para valores preestablecidos [fuente propia] 71

Figura 3-71. Obtención de los valores predeterminados y establecimiento en los *EditText* correspondientes [fuente propia] 71

Figura 3-72. Obtención de la referencia al *spinner* y creación del adaptador [fuente propia] 71

Figura 3-73. Obtención del valor seleccionado del *Intent* si existe [fuente propia] 72

Figura 3-74. Definición del *listener* [fuente propia] 72

Figura 3-75. Inicialización de la lista de registros y configuración del *RecyclerView* [fuente propia] 72

Figura 3-76. Verificación de si hay registros enviados desde *MenuPrincipal* [fuente propia] 73

Figura 3-77. Obtención de los valores recibidos y establecimiento en los campos de texto asociados [fuente propia] 73

Figura 3-78. Configuración del botón para guardar y eliminar el registro [fuente propia] 74

Figura 3-79. Método para guardar el registro y mostrar los datos ingresados en el *RecyclerView* [fuente propia] 74

Figura 3-80. Diferentes operaciones realizadas con los registros [fuente propia] 75

Figura 3-81. Código empleado para la eliminación de los registros [fuente propia] 75

Figura 4-1. Representación gráfica del escenario de prueba de la aplicación [fuente propia] 77

Figura 4-2. Información básica que contendrían los tags NFC de Pepe y María [fuente propia]...........78

Figura 4-3. Mensaje de error que se mostraría en caso de introducir una fecha de nacimiento incorrecta [fuente propia]............78

Figura 4-4. Pantalla de inicio de sesión a la aplicación en la izquierda y el mensaje de error correspondiente [fuente propia]............79

Figura 4-5. Pantalla del menú principal de la aplicación [fuente propia]............80

Figura 4-6. Mensaje mostrado en el caso de que algún dato del apartado *Identification* sea modificado [fuente propia]............81

Figura 4-7. Pantalla inicial de la pestaña "*Cause*" en la imagen de la izquierda y esa misma pantalla con los datos modificados en la imagen de la derecha [fuente propia]............81

Figura 4-8. Pantalla inicial de la pestaña "*Assessment*" en la izquierda, desplegables abiertos de esa misma pestaña en la imagen central, y mensaje de error en la fecha en la imagen de la derecha [fuente propia]............82

Figura 4-9. Mensaje de error al introducir hora incorrecta en la izquierda y mensaje de error al introducir fecha incorrecta en la derecha [fuente propia]............83

Figura 4-10. Apartado rellenado por Jesús [fuente propia]............84

Figura 4-11. Parte de los datos guardados en el servidor [fuente propia]............84

Figura 4-12. Mensaje mostrado si la información se envía correctamente al servidor en la imagen de la izquierda y mensaje mostrado si existe un error de conexión en la imagen derecha [fuente propia].....85

Figura 4-13. Pantalla inicial de "*Basic vital signs*" en la imagen de la izquierda, datos proporcionados, incluyendo su registro en la imagen central y mensaje de error en la imagen derecha [fuente propia] .86

Figura 4-14. Pantalla inicial de "*Treatment*" en la imagen de la izquierda y datos cubiertos en la imagen de la derecha [fuente propia]............87

Figura 4-15. Pantallas asociadas a "*Movement*" [fuente propia]............88

Figura 4-16. Mensaje de error en la imagen de la izquierda y mensaje que indica que se ha realizado correctamente la escritura en la imagen de la derecha [fuente propia]............88

Figura 4-17. Mensaje de advertencia en caso de que los ID de las tarjetas NFC no coincidan [fuente propia]............89

Figura 4-18. Posibles mensajes que podría ver el usuario durante el proceso de lectura de un tag [fuente propia]............90

Figura 4-19. Capturas de la aplicación *NFC Tools* que muestran el uso real de la capacidad de almacenamiento del tag NFC en diferentes situaciones [fuente propia]............91

ÍNDICE DE TABLAS

Tabla 2-1. Comparación entre Android e iOS [fuente propia] 25
Tabla 2-2. Diferencias entre Java y Kotlin [fuente propia]..30
Tabla 2-3. Especificaciones del dispositivo móvil empleado [28] 31
Tabla 2-4. Especificaciones del tag NFC empleado [29]..32
Tabla 3-1. Ventajas y desventajas de cada opción planteada [fuente propia]..............................37

1 INTRODUCCIÓN Y OBJETIVOS

1.1 Contexto y motivación

Dado el elevado número de conflictos que se han ido sucediendo a lo largo de los años, surge la necesidad de proporcionar una serie de protocolos que regulan la actuación en situaciones de emergencia para la asistencia a víctimas en entornos de alta amenaza, como tiroteos, ataques terroristas o desastres naturales. Hasta 1996, las directrices militares para la gestión de traumas se basaban en los métodos empleados en el sector civil. Sin embargo, gracias a los esfuerzos realizados por el Comando de Guerra Naval Especial de los Estados Unidos, se crean en 1996 una serie de nuevas estrategias denominadas colectivamente como *Tactical Combat Casualty Care* (TCCC) [1].

Actualmente, la asistencia táctica de heridos en combate se divide en tres fases: CUF, TFC y TACEVAC [2], que se especifican a continuación.

La primera fase, CUF (*Care Under Fire*), se refiere a la atención médica que se brinda a los heridos en la zona de combate bajo el fuego enemigo. En esta fase, el personal médico se enfoca en la supresión del fuego hostil y en la evaluación de un plan de rescate. La atención médica en esta fase se limita a la aplicación de torniquetes si es necesario.

La segunda fase, TFC (*Tactical Field Care*), es donde se utilizan las tarjetas TCCC (*Tactical Combat Casualty Care*). En esta fase, los heridos ya no se encuentran bajo fuego enemigo y pueden ser atendidos sobre el terreno en condiciones de mayor seguridad. Durante la misma, el personal médico se enfoca en la evaluación y el tratamiento de las lesiones que ponen en peligro la vida del herido.

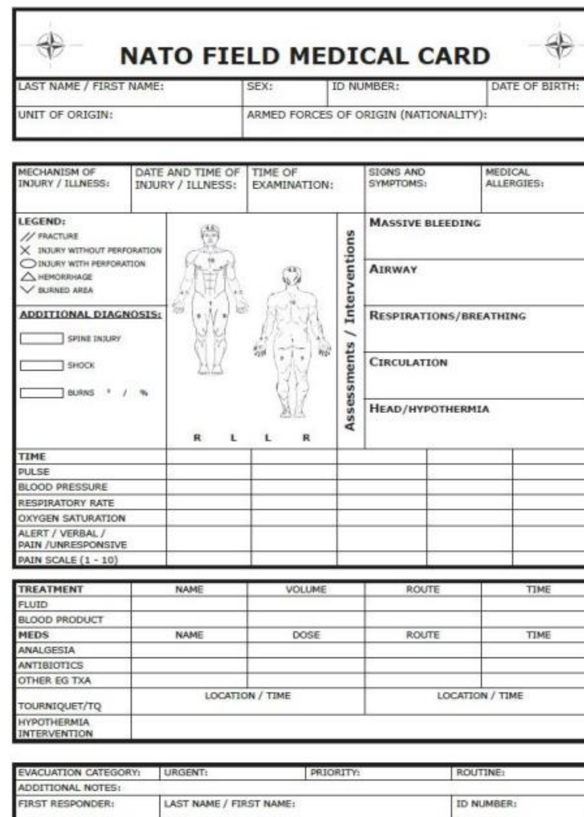
La tercera fase, TACEVAC (*Tactical Evacuation*), se refiere al traslado de los heridos a un lugar seguro para recibir atención médica adicional. En esta fase, el personal médico se enfoca en el transporte seguro y rápido de los heridos a un lugar donde puedan recibir atención médica adecuada.

El protocolo TCCC es un estándar OTAN respaldado por el Colegio Estadounidense de Cirujanos y la Asociación Nacional de EMT (*Emergency Medical Technicians*) para el Manejo de Accidentes en Ambientes Tácticos Militares. En él se definen las pautas de actuación para documentar de forma clara el tratamiento y el estado de los heridos en situaciones de combate.

Dentro de este protocolo se establece el uso de las tarjetas NFMC (*NATO Field Medical Card*) (ver Figura 1-1), definidas estas como las herramientas fundamentales que condensan los procedimientos de atención médica dirigidos a víctimas de combate. Estas tarjetas han sido concebidas para su empleo por parte del personal militar y civil operando en entornos hostiles y, habitualmente, se presentan en forma de tarjetas impermeables. Sin embargo, existen determinados conflictos que se producen en zonas donde existen condiciones climatológicas adversas, véase lluvia intensa, polvo, metralla, terreno embarrado, etc. Todo ello, junto con la posibilidad de la ininteligibilidad de la tarjeta, debido a una mala caligrafía,

suponen importantes inconvenientes a la hora de trabajar. Además, la transformación de dichas tarjetas a un formato electrónico permitiría transferir estos datos a tiempo real a las localizaciones donde posteriormente el herido sería atendido, permitiendo mejorar la gestión de atención al paciente.

En este trabajo se propone el estudio y diseño, y desarrollo de una solución que permita digitalizar la información contenida en las actuales tarjetas NFMC (*NATO Field Medical Card*) y transferirla a algún dispositivo electrónico vinculado físicamente de manera individual a cada paciente.



NATO FIELD MEDICAL CARD

LAST NAME / FIRST NAME:	SEX:	ID NUMBER:	DATE OF BIRTH:
UNIT OF ORIGIN:	ARMED FORCES OF ORIGIN (NATIONALITY):		

MECHANISM OF INJURY / ILLNESS:	DATE AND TIME OF INJURY / ILLNESS:	TIME OF EXAMINATION:	SIGNS AND SYMPTOMS:	MEDICAL ALLERGIES:

LEGEND:
// FRACTURE
X INJURY WITHOUT PERFORATION
O INJURY WITH PERFORATION
△ HEMORRHAGE
V BURNED AREA

ADDITIONAL DIAGNOSIS:
SPINE INJURY
SHOCK
BURNS % / %

R L L R

Assessments / Interventions
MASSIVE BLEEDING
AIRWAY
RESPIRATIONS/BREATHING
CIRCULATION
HEAD/HYPOTHERMIA

TIME					
PULSE					
BLOOD PRESSURE					
RESPIRATORY RATE					
OXYGEN SATURATION					
ALERT / VERBAL / PAIN /UNRESPONSIVE					
PAIN SCALE (1 - 10)					

TREATMENT	NAME	VOLUME	ROUTE	TIME
FLUID				
BLOOD PRODUCT				
MEDS	NAME	DOSE	ROUTE	TIME
ANALGESIA				
ANTIBIOTICS				
OTHER EG TXA				
TOURNIQUET/TQ	LOCATION / TIME		LOCATION / TIME	
HYPOTHERMIA INTERVENTION				

EVACUATION CATEGORY:	URGENT:	PRIORITY:	ROUTINE:
ADDITIONAL NOTES:			
FIRST RESPONDER:	LAST NAME / FIRST NAME:		ID NUMBER:

Figura 1-1. Ejemplo de NFMC [3]

1.2 Objetivos del TFG

El presente trabajo tiene como objetivo el diseño, desarrollo y validación de una aplicación capaz de sustituir a las actuales tarjetas NFMC que portan los combatientes en una situación de conflicto.

Los objetivos principales de esta investigación son:

- Mejorar el proceso de toma de los datos del herido y posterior actualización: la aplicación busca agilizar la toma de datos y su posterior modificación, mejorando la eficiencia en la atención médica en el campo de batalla.
- Eliminar los inconvenientes debidos a la degradación del soporte: la información del herido se almacenará en una tarjeta NFC (*Near Field Communication*), protegida contra condiciones climáticas adversas y otros riesgos físicos asociados al papel.
- Analizar las fortalezas y debilidades de la aplicación: se evaluará la funcionalidad de la aplicación, su impacto económico, social y ético, y su potencial para reemplazar las tarjetas NFMC actuales.

Para alcanzar los objetivos mencionados, se seguirán las siguientes etapas:

1. Revisión del estado del arte.

2. Diseño de una solución: donde se analizarán las diferentes opciones planteadas para explicar el porqué de la que ha sido finalmente seleccionada.
3. Desarrollo de la aplicación: en este apartado se explicará en detalle la codificación de las diferentes actividades de la aplicación, los tipos de datos empleados y cómo se comunican entre sí y con el tag NFC.
4. Validación: donde se expondrán los resultados obtenidos y se le mostrará al lector una visión en detalle del funcionamiento de la aplicación.
5. Análisis de impacto: se estudiarán los posibles impactos económicos, sociales y éticos de la aplicación, teniendo en cuenta las necesidades y preocupaciones de los usuarios finales.

1.3 Organización de la memoria

Tras contextualizar el trabajo y habiendo expuesto los objetivos que se desean cumplir, se pasa a la descripción de la memoria con el fin de mostrar claramente su estructura. El documento quedará dividido en cinco capítulos, la bibliografía y los anexos.

- Capítulo 1: En esta sección se presenta una contextualización general del trabajo, se exponen los motivos subyacentes que lo justifican y se establecen los principales objetivos.

- Capítulo 2: Aquí se desglosará el estado del arte del trabajo. Se aborda la situación actual de la sanidad militar y se profundiza en aspectos cruciales de la evacuación de pacientes. Además, se examinan la tecnología NFC, el sistema operativo Android y el lenguaje de programación Kotlin, empleados en el desarrollo de este TFG.

- Capítulo 3: Este capítulo aborda el desarrollo de la aplicación. Se describen las diferentes alternativas consideradas y se presenta el entorno de trabajo junto con el diseño de la aplicación. Asimismo, se ofrece una explicación exhaustiva del código utilizado en la implementación de cada una de las funcionalidades de la aplicación.

- Capítulo 4: En esta sección se presenta el funcionamiento práctico de la aplicación para un caso de uso que permita evidenciar que se satisfacen los requisitos, incluyendo una medida de la necesidad de almacenamiento en los tags NFC que demanda la aplicación.

- Capítulo 5: Se exponen las conclusiones obtenidas a partir del trabajo realizado y se proponen posibles áreas de investigación futura.

- Por último, se incluyen las referencias bibliográficas y las fuentes consultadas, así como los anexos que contienen información adicional, incluido el código desarrollado.

2 ESTADO DEL ARTE

A lo largo de este capítulo se presenta la evolución y la importancia de la atención médica militar y se analizan los avances realizados en técnicas de evacuación sanitaria y protocolos de tratamiento de heridos en combate. Además, se explica la evolución de protocolos como el *Tactical Combat Casualty Care* (TCCC) al actual estándar OTAN AmedP-8.1 [3] donde se define la *NATO Field Medical Card* (NMFC).

Por otro lado, también se explorará la tecnología NFC (*Near Field Communication*) y se abordará su origen y evolución, así como sus características técnicas y sus aplicaciones en la actualidad. Además, se analizarán las razones que hacen conveniente el desarrollo de una aplicación para dispositivos móviles que aproveche esta capacidad y se examinará el sistema operativo Android.

En la última parte, se explicará el desarrollo de aplicaciones para dispositivos móviles Android utilizando Android Studio y se analizarán sus características. También se explicará la estructura de un proyecto en Android Studio y se compararán los lenguajes de programación Java y Kotlin. Además, se presentarán los recursos hardware que utilizaremos.

Finalmente, se revisará un proyecto previo relacionado con este Trabajo Fin de Grado.

2.1 Contexto de la sanidad militar

2.1.1 Historia de la sanidad militar

En la antigüedad, los ejércitos ya contaban con médicos y cirujanos para tratar heridas y enfermedades y, durante la Edad Media, las órdenes religiosas también desempeñaron un papel relevante en la atención médica militar. Un ejemplo de ello es la Orden de San Juan de Jerusalén (Caballeros Hospitalarios), fundada en el siglo XI y que se dedicaba a la atención médica de los peregrinos y enfermos en Tierra Santa, así como al establecimiento de hospitales en Jerusalén y otras ciudades, proporcionando atención médica a los caballeros cruzados y a la población local.

Por otro lado, durante el siglo XIX y la Primera Guerra Mundial cabe destacar la fundación de La Cruz Roja Internacional en 1863 de la mano de Henry Dunant [4], marcando un hito en la atención médica humanitaria durante el conflicto. También, a nivel nacional, cabe destacar la creación del Cuerpo de Sanidad Militar en 1855. Por otro lado, es reseñable el papel de Florence Nightingale durante la Guerra de Crimea, la cual contribuyó a reducir significativamente la tasa de muertes del bando británico. Tal fue su papel durante el conflicto que la reina Victoria le otorgó la Real Cruz Roja en 1883 y, en 1907, el Rey Eduardo VII le concedió la Orden del Mérito [5], siendo esta la primera vez que se le concedía a una mujer.

Con la llegada de la Segunda Guerra Mundial se realizaron numerosos avances en técnicas de cirugía de emergencia, transfusiones de sangre y tratamientos para enfermedades infecciosas. Además, una vez finalizada esta, se crea la Organización Mundial de la Salud (OMS) el 7 de abril de 1948 para coordinar la atención médica a nivel global. Actualmente la sanidad militar sigue siendo esencial en conflictos, misiones de paz y desastres naturales y contribuye al desarrollo de tratamientos y tecnologías médicas.

2.1.2 Sanidad militar en operaciones

Tal y como establece el EMAD (Estado Mayor de la Defensa) en la Doctrina Sanitaria en operaciones [6], "se entiende por Sanidad Militar el conjunto de organismos, actividades, personal, recursos y procedimientos, que tienen por objetivo principal el apoyo sanitario al personal militar tanto en territorio nacional (TN) como en el ámbito de las operaciones". Esta se rige por unos principios de carácter general y otros de carácter operativo (calidad, oportunidad, flexibilidad, etc). Dentro del primer grupo encontramos:

- Legalidad internacional: asegura el respeto a las leyes que rigen los enfrentamientos bélicos, brindando atención médica a los individuos heridos o enfermos, con la única restricción de no interferir en el cumplimiento de la misión asignada.
- Ética y legislación específica: establece que deben cumplirse las obligaciones legales y éticas propias de cada especialidad fundamental.
- Acreditación: asegura que se ajustará a los principios reconocidos por la comunidad internacional, definiendo parámetros de mejora constante en la prestación del servicio y herramientas que permitan su evaluación objetiva.

Por otro lado, es importante destacar el apoyo sanitario en operaciones militares, el cual abarca un conjunto de actividades, tanto en la fase de planificación como de ejecución, orientadas principalmente a proveer de manera oportuna todo lo necesario para prevenir enfermedades, así como para promover, mantener y restaurar la salud durante el despliegue, sostenimiento y repliegue de una operación. Su objetivo es alcanzar la capacidad operativa indispensable para el cumplimiento de la misión.

En este contexto se entiende por operaciones militares aquellas definidas en el artículo 16 de la Ley Orgánica 5/2005, de 27 de noviembre, de la Defensa Nacional [7]. Estas abarcan la vigilancia de espacios marítimos y aéreos, que garantiza la integridad territorial y la soberanía nacional, asegurando la protección de la vida de los ciudadanos y sus intereses. Además, la colaboración en operaciones internacionales de mantenimiento de la paz y estabilización contribuye a la seguridad global y al cumplimiento de compromisos internacionales. En el ámbito nacional, se presta apoyo en la lucha contra el terrorismo y en operaciones de rescate, asegurando la eficaz respuesta a situaciones de emergencia. Asimismo, se establecen procedimientos operativos para responder a amenazas terroristas que pongan en peligro la vida de la población, coordinándose estrechamente con las autoridades responsables. Esta colaboración se extiende a las administraciones públicas, donde se actúa en casos de emergencia, catástrofe o necesidades públicas, siguiendo las disposiciones legales vigentes. Además, se participa activamente en la preservación de la seguridad y el bienestar de los ciudadanos españoles en el extranjero, cooperando con organismos nacionales e internacionales según los criterios establecidos de coordinación y responsabilidad.

2.1.2.1 Niveles de capacitación (NC)

Tal y como establece el *Real Decreto 230/2017, de 10 de marzo, por el que se regulan las competencias y cometidos de apoyo a la atención sanitaria del personal militar no regulado por la Ley 44/2003, de 21 de noviembre, de ordenación de las profesiones sanitarias, en el ámbito estrictamente militar* [8], y dada la importancia de la reducción en los tiempos de respuesta en la atención a las bajas, el apoyo sanitario es materia tanto del personal militar facultativo como del no facultativo, siendo este último "capacitado mediante los correspondientes cursos de especialización o informativos de la enseñanza de perfeccionamiento, así como mediante las actividades necesarias de instrucción y

adiestramiento, para alcanzar y mantener las competencias requeridas". Dentro de los niveles de capacitación (NC) que se establecen para este personal distinguimos:

- (NC1) Básico: lo posee todo militar que participa en operaciones, y hace referencia a aquel que permite realizar procedimientos de atención inmediata utilizando los recursos del Botiquín Individual del Combatiente y los protocolos de atención establecidos.

- (NC2) Medio: lo componen los miembros pertenecientes a las propias Unidades, que poseen la capacidad de prestar las técnicas de soporte vital con los protocolos y materiales específicos que se determinen. El personal que se encuentre en este nivel recibirá el nombre de Personal de Apoyo a la atención sanitaria en operaciones.

- (NC3) Avanzado: personal específico formado para actuar en el contexto de aquellas unidades/buques que por sus características particulares actúan en entornos de especial aislamiento o en apoyo directo al personal facultativo en materia de asistencia sanitaria de operaciones. El personal que se encuentre en este nivel será denominado Personal de Apoyo Avanzado a la atención sanitaria en operaciones.

En función del nivel de capacitación que se obtenga, es posible tener asignadas determinadas competencias, tal y como se refleja en la Figura 2-1.

Competencias	NC1 Básico	NC2 Medio	NC3 Avanzado
Aspectos generales del apoyo sanitario táctico.	X	X	X
Empleo del torniquete.	X	X	X
Aplicación de presión directa.	X	X	X
Aplicación de vendaje.	X	X	X
Aplicación de vendaje hemostático.	X	X	X
Aplicación de vendaje compresivo.	X	X	X
Aplicación de pinzas y otros de compresión hemostática.			X
Técnicas de movilización de bajas.	X	X	X
Maniobras frente-mentón y elevación mandibular de apertura de la vía aérea.	X	X	X
Cánula nasofaríngea.	X	X	X
Posición de seguridad.	X	X	X
Posiciones de mantenimiento de la permeabilidad de la vía aérea.	X	X	X
Mascarilla laríngea.			X
Tubo laríngeo.			X
Punción cricotiroidea.			X
Aplicación del parche torácico en tórax abierto.	X	X	X
Punción torácica con aguja.		X	X
Administración de oxígeno.		X	X
Valoración del estado de shock.	X	X	X
Canalizar vía venosa periférica para administración de fluidos.			X
Vía intraosea.		X	X
Fluidoterapia.		X	X
Prevención de la hipotermia.	X	X	X
Parche rígido para cobertura ocular.	X	X	X
Analgesia oral de dotación.	X	X	X
Antibióticos orales de dotación.	X	X	X
Aplicador oral de fentanilo.		X	X
Cloruro mórfico subcutáneo.			X
Inmovilización con férula.	X	X	X
Inmovilización con férula de tracción.			X
Aislar de la fuente de quemadura.	X	X	X
Cubrir las áreas quemadas.	X	X	X
Monitorización de signos vitales.		X	X
Desfibrilador externo automático.		X	X

Figura 2-1. Competencias asociadas a cada nivel de capacitación [11]

2.1.2.2 Asistencia a las bajas sanitarias

Es importante comprender que, dentro del apoyo sanitario en operaciones militares, la asistencia a los miembros de una operación con limitaciones físicas o mentales, ya sean temporales o permanentes, que les impiden realizar sus tareas asignadas resulta un pilar fundamental. Es por ello por lo que se distinguen dos tipos de bajas sanitarias diferentes, las Bajas Sanitarias de Combate (BCO), entendidas estas como las que resultan de manera directa del combate, y las Bajas Sanitarias de no Combate (BNCO), que son consecuencia de algún accidente o causa ajena al conflicto.

En lo que al tratamiento de estas bajas se refiere, el papel de las Formaciones Sanitarias de Tratamiento (FST) resulta un punto fundamental. Las FST son unidades móviles de apoyo sanitario en operaciones militares que se clasifican por su capacidad de atención médica (ROLE) y pueden escalarse según las necesidades. Entre las características generales de los diferentes ROLE destacan su movilidad (excepto la FST ROLE 4, que es fija), las capacidades adaptables que poseen según la tecnología, la capacitación del personal y los procedimientos, y su capacidad de convertirse en instalaciones fijas si la operación lo requiere.

Atendiendo al nivel asistencial mínimo que proporcionan, podemos diferenciar entre las siguientes FST [6]:

- Puesto de enfermería: se trata de una FST con menor capacidad que ROLE 1 cuyo uso se contempla si no hay personal médico. Sin embargo, es necesaria la presencia de un oficial enfermero con formación continua específica para realizar las funciones de este tipo de FST y con capacidad de realizar telemedicina en tiempo real.
- ROLE 1: se encuentra físicamente situado en la zona de despliegue y su función principal es proporcionar atención médica en la vanguardia. Cuenta con la capacidad de brindar atención médica completa en una variedad de situaciones, desde atención primaria hasta urgencias y emergencias. Está compuesto por personal del Cuerpo Militar de Sanidad (CMS) médico y enfermero, y personal militar acreditado para el apoyo a la atención sanitaria.
- ROLE 2: se establece en las inmediaciones de la zona de vanguardia y se caracteriza fundamentalmente por aportar capacidad quirúrgica a las capacidades propias de un ROLE 1. Además, podemos diferenciar tres tipos de ROLE 2 en base a sus capacidades sanitarias e independientemente de su movilidad:
    - ROLE 2B (básico): permite realizar técnicas de reanimación, de control de daños y/o cirugía de control de daños.
    - ROLE 2E (reforzado): añade al ROLE 2B las capacidades de cirugía primaria o reparadora, cuidados intensivos y de diagnóstico por imagen.
    - ROLE 2F (avanzado): no dispone de las capacidades vistas en los roles anteriores, sin embargo, permite realizar cirugía de control de daños en escenarios muy exigentes, remotos y en condiciones de escasez.
- ROLE 3: normalmente se materializa en un hospital de campaña (puede estar embarcado o no) y añade la posibilidad de realizar cirugía especializada a las capacidades del ROLE 2E, así como otras especialidades complementarias. Permite, además, dar cobertura a un mayor número de personal y ofrece la posibilidad de hospitalizar hasta cien bajas.
- ROLE 4: es la unidad de apoyo sanitario de más alto nivel. Se encarga de proporcionar cuidados definitivos ya que permite abarcar todo el abanico de especialidades. España cuenta actualmente con una FST ROLE 4, que está ubicada en el Hospital Central de la Defensa "Gómez Ulla" en Madrid [9]. Esta FST tiene capacidad para atender hasta 200 pacientes.

2.1.2.3 Evacuación de las bajas sanitarias

El traslado de bajas desde la zona de combate hasta una FST o entre las propias FST que es llevada a cabo por personal sanitario es, por definición, una evacuación sanitaria. Esta puede ser de dos tipos atendiendo a la preparación y a la rapidez con la que se lleve cabo. De esta manera distinguimos entre

MEDEVAC (*Medical Evacuation*), donde el traslado de bajas se realiza de forma segura, eficiente y con la atención médica adecuada, así como con los medios de transporte especializados y equipados, y CASEVAC (*Casualty Evacuation*), empleado cuando los medios del MEDEVAC no se pueden garantizar debido a factores operativos o por la falta de material.

Por otro lado, es posible clasificar los medios de evacuación en función del área o situación operativa en la que son utilizados. Podemos diferenciar entonces:

- Evacuación avanzada: La evacuación se lleva a cabo desde el punto de origen de la baja o del lugar de recogida hasta la FST más idónea dentro del área de operaciones, la cual no tiene por qué ser la más inmediata. En todo momento, se prioriza el empleo de los medios más eficientes y rápidos acordes a la situación táctica presente.
- Evacuación táctica: Empleada cuando el traslado de bajas es entre FST dentro del área de operaciones.
- Evacuación estratégica: Llevada a cabo cuando es necesario trasladar la baja desde el área de operaciones hasta territorio nacional o a un tercer país amigo o aliado. Dentro de este tipo de evacuaciones, a su vez, es posible clasificarla en función de su urgencia y de su dependencia asistencial.

  Atendiendo a su urgencia:
- Urgente: Requiere la activación y despegue del medio de evacuación aérea con la mayor brevedad posible.
- Prioritaria: El medio de evacuación aérea puede destinarse a otros casos más necesarios; sin embargo, su uso no debería prolongarse sin justificación.
- Rutinaria: El criterio clínico no es el aspecto más determinante para establecer el momento adecuado para llevar a cabo la evacuación aérea.

Atendiendo a su dependencia asistencial distinguimos entre alta dependencia asistencial, media dependencia asistencial, baja dependencia asistencial y ordinaria.

2.2 Estándares que rigen las evacuaciones militares

2.2.1 Protocolo TCCC

En la década de 1980 y debido al reciente conflicto de la Guerra de Vietnam, las Fuerzas Armadas estadounidenses se dieron cuenta de una terrible afirmación extraída a partir de las estadísticas de dicho conflicto: “La mayoría de los heridos en combate morían antes de llegar a ser atendidos por un cirujano” [1].

Como solución a dicho problema surge el TCCC (*Tactical Combat Casualty Care*), definido como un conjunto de protocolos médicos elaborados para la atención de heridos en combate. El origen de dichos protocolos se remonta a 1993, cuando el proyecto del TCCC comenzó a utilizarse por el Comando de Guerra Naval Especial de los Estados Unidos para impulsar un estudio sobre técnicas de atención a víctimas de combate. Sin embargo, no fue hasta 1996 cuando las directrices de dicho protocolo fueron publicadas en Medicina Militar.

Posteriormente, dada la recomendación de mantenerlas actualizadas de manera regular, el Comando de Operaciones Especiales estadounidenses (USSOCOM) financió un estudio de 2 años (2001-2002).

Finalmente, el USSSOCOM creó en 2001 el TCCC, el cual fue revisado y aprobado por el *Committee on Tactical Combat Casualty Care* (CoTCCC), formado inicialmente por oficiales sanitarios con experiencia militar, cirujanos traumatólogos y varios especialistas civiles en atención traumatológica (ver Figura 2-2). Actualmente, el CoTCCC continúa siendo el responsable de las actualizaciones y mejoras de los protocolos del TCCC en base a las lecciones aprendidas en las misiones donde participa el ejército estadounidense [10].

The Committee on Tactical Combat Casualty Care: 2002

Chairman – CAPT Stephen Giebner

COL Robert Allen	LTC Stephen Flaherty	Dr. Norman McSwain
COL Frank Anders	CDR Scott Flinn	SFC Robert Miller
CPT Steve Anderson	MAJ John Gandy	MAJ Kevin O'Connor
COL James Bagian	CAPT Larry Garsha	CAPT Edward Otten
COL Ron Bellamy	COL John Holcomb	LTC Tyler Putnam
1LT Bart Bullock	Dr. David Hoyt	CDR Peter Rhee
CAPT Frank Butler	LTC Donald Jenkins	CAPT Larry Roberts
Dr. Howard Champion	COL Jay Johannigman	CDR Jeff Timby
TSGT George Cum	MSG John Kennedy	HMCM Gary Welt
CAPT Roger Edwards	CPT Robert Mabry	

Executive Assistants: LT David Anderson, Ms. Shannon Addison

Figura 2-2. Composición del CoTCCC en 2002 [1]

2.2.2 Tarjetas TCCC

Las tarjetas TCCC (*Tactical Combat Casualty Care*), ver Figura 2-3, son herramientas esenciales dirigidas a los primeros participantes en situaciones de trauma en entornos tácticos.

Surgieron como una necesidad de recoger la información médica del combatiente con la mayor brevedad posible de tal manera que se minimizase el número de víctimas. Además, suponen una guía rápida y clara para el seguimiento de las lesiones más comunes que se producen en situaciones de combate, como hemorragias, heridas penetrantes, heridas por explosivos, etc.

Las tarjetas TCCC son necesarias por varias razones:

- Proporcionan un resumen claro y conciso de los procedimientos de tratamiento más reseñables. Esto es de vital importancia en entornos tácticos, donde los primeros intervinientes suelen estar bajo presión y pueden no contar con el tiempo necesario para consultar manuales o recursos más extensos.
- Su empleo es relativamente sencillo, incluso para personal con poca formación médica. Las tarjetas han sido diseñadas para ser claras y fáciles de comprender.
- Son portátiles y cómodas a la hora de transportarlas dado su reducido tamaño y peso, lo que las hace fáciles de guardar en el bolsillo o en el equipo de combate.

En la actualidad, las tarjetas TCCC son utilizadas en todo el mundo por fuerzas militares, cuerpos de mantenimiento del orden y otros primeros intervinientes que trabajan en entornos tácticos. Las tarjetas TCCC están disponibles en varios idiomas y se han adaptado para su uso en diferentes contextos, como operaciones militares, operaciones de seguridad interna y respuesta a emergencias.

En España, estas tarjetas se están utilizando cada vez más por fuerzas militares, fuerzas del orden y otros primeros intervinientes. En julio de 2023, el JEMAD publicó una resolución [11] por la que se implanta un estándar OTAN sobre documentación relativa al tratamiento médico inicial y evacuación de personal. Además, el Ministerio de Defensa español, concretamente el Ejército de Tierra, publicó un curso de TCCC [12] que contempla la puesta en práctica de dicha resolución.

Figura 2-3. Tarjeta TCCC [13]

2.2.3 NATO Field Medical Card (NFMC)

La transición a las actuales *NATO Field Medical Card* (NFMC), cuya información mínima requerida se recoge en el estándar OTAN AMedP-8.1 del 27 de junio de 2023 [3], ha sido un proceso gradual que ha ocurrido a lo largo de las últimas décadas. Esta evolución ha surgido como respuesta a la necesidad de mejorar la recopilación y el intercambio de información médica en entornos militares, tanto en situaciones de combate como en escenarios no tácticos.

Inicialmente, las tarjetas médicas de campo, que precedieron a la NFMC, se empleaban principalmente para registrar información médica básica sobre los pacientes heridos en combate y solían ser simples y limitadas en términos de la cantidad de información que podían contener. El progreso tecnológico y la creciente complejidad de las operaciones militares generaron la necesidad de herramientas más sofisticadas y completas para la gestión de la información médica [13]. En respuesta a esta demanda, se desarrolló la TCCC por parte de las Fuerzas de Operaciones Especiales de EEUU a finales de los años noventa. Estas tarjetas ofrecen un formato estandarizado y detallado para recopilar datos médicos, tratamientos, procedimientos y otros aspectos relevantes para la atención médica de los pacientes militares.

La TCCC evoluciona hacia la NFMC impulsada por la creciente importancia de la interoperabilidad entre las fuerzas militares de diferentes países, especialmente en el contexto de operaciones multinacionales. La estandarización de la NFMC facilita el intercambio de información médica entre las fuerzas aliadas y contribuye a una atención médica más efectiva y coordinada en entornos militares conjuntos.

Tal y como se establece en [3], las NFMC (ver Figura 2-4) son unos "documentos estandarizados para informar del tratamiento inicial del paciente. La NFMC proporcionará documentación de la identidad de las víctimas, primeros auxilios, tratamiento médico inicial y atención en tránsito y hasta las instalaciones de tratamiento médico (ROLE 1). La NFMC está destinada para su uso inicial por la

primera persona en prestar ayuda en el lugar de la lesión o lo antes posible y seguirá a la víctima en ruta. La NFMC estará disponible en el lugar de la lesión."

Además, en dicha publicación [3] se establece la información mínima que deben contener dichas tarjetas, siendo esta:

- Identificación:
  a) Apellidos
  b) Nombre
  c) Sexo
  d) Número de ID
  e) Fecha de nacimiento
  f) Lugar de origen
  g) Fuerzas Armadas de origen (Nacionalidad)

- Causa:
  a) Mecanismo de lesión o enfermedad

- Evaluación:
  a) Fecha y hora de las lesiones o de la enfermedad
  b) Fecha del examen
  c) Signos y síntomas
  d) Alergias médicas
  e) Signos vitales básicos (por ejemplo, pulso, presión arterial, frecuencia respiratoria, saturación de oxígeno, estado del paciente, escala de dolor)

- Tratamiento:
  a) Flujo y productos sanguíneos (volumen, vía, hora)
  b) Analgesia (dosis, vía, hora)
  c) Antibióticos (dosis, vía, hora)
  d) Otros, por ejemplo, TXA o "*Tranexamic acid*" (dosis, vía, hora)
  e) Torniquete/TQ (ubicación y hora)
  f) Intervención de hipotermia

- Movimiento:
  a) Categoría de la evacuación (urgente / prioridad / rutina)
  b) Notas adicionales
  c) Primera persona en socorrer (apellidos / nombre / número de ID)

NATO FIELD MEDICAL CARD

LAST NAME / FIRST NAME:	SEX:	ID NUMBER:	DATE OF BIRTH:
UNIT OF ORIGIN:	ARMED FORCES OF ORIGIN (NATIONALITY):		

MECHANISM OF INJURY / ILLNESS:	DATE AND TIME OF INJURY / ILLNESS:	TIME OF EXAMINATION:	SIGNS AND SYMPTOMS:	MEDICAL ALLERGIES:

LEGEND:
// FRACTURE
X INJURY WITHOUT PERFORATION
O INJURY WITH PERFORATION
△ HEMORRHAGE
∨ BURNED AREA

ADDITIONAL DIAGNOSIS:
SPINE INJURY
SHOCK
BURNS ° / %

R L L R

Assessments / Interventions
MASSIVE BLEEDING
AIRWAY
RESPIRATIONS/BREATHING
CIRCULATION
HEAD/HYPOTHERMIA

TIME					
PULSE					
BLOOD PRESSURE					
RESPIRATORY RATE					
OXYGEN SATURATION					
ALERT / VERBAL / PAIN /UNRESPONSIVE					
PAIN SCALE (1 - 10)					

TREATMENT	NAME	VOLUME	ROUTE	TIME
FLUID				
BLOOD PRODUCT				
MEDS	NAME	DOSE	ROUTE	TIME
ANALGESIA				
ANTIBIOTICS				
OTHER EG TXA				
TOURNIQUET/TQ	LOCATION / TIME		LOCATION / TIME	
HYPOTHERMIA INTERVENTION				

EVACUATION CATEGORY:	URGENT:	PRIORITY:	ROUTINE:
ADDITIONAL NOTES:			
FIRST RESPONDER:	LAST NAME / FIRST NAME:		ID NUMBER:

Figura 2-4. ***NATO Field Medical Card*** **[6]**

2.3 Tecnología NFC

2.3.1 Historia

NFC, por sus siglas en inglés, *Near Field Communication* [14], es una plataforma abierta pensada desde su inicio para teléfonos y dispositivos móviles. A pesar del reciente aumento de su popularización, la idea surgió en la década de 1980, como una evolución de la tecnología RFID (*Radio Frecuency Identification*), desarrollada por ingenieros de Sony y Philips [15]. RFID es un término general para la identificación por radiofrecuencia, en donde la comunicación suele darse de manera unidireccional y la distancia a la que puede funcionar varía desde unos pocos centímetros hasta decenas de metros. Esta tecnología es usada en numerosas aplicaciones tales como el seguimiento de vehículos en peajes o estacionamientos, en los dorsales de corredores para cronometraje en carreras y eventos deportivos, etc. Por otro lado, NFC es considerado un subconjunto de RFID que está diseñado para trabajar a distancias muy cortas, generalmente unos pocos de centímetros y que, a diferencia de este, permite el intercambio bidireccional de datos. Inicialmente, la tecnología NFC se empleó para tarjetas de acceso para edificios y sistemas de transporte público.

Sin embargo, no fue hasta 2002 cuando se estableció el estándar ISO/IEC 18092 que permitió a los fabricantes de dispositivos y tarjetas NFC desarrollar productos compatibles entre sí. En 2004 [16] se creó el Foro NFC de la mano de Nokia, Philips y Sony, que definía las especificaciones técnicas. En la década de 2010 dicha tecnología comenzó a expandirse, de tal modo que los teléfonos móviles empezaron a incorporar chips NFC, permitiendo así el desarrollo de nuevas aplicaciones, como el pago móvil sin contacto y la transferencia de datos entre dispositivos. Posteriormente se popularizaron las etiquetas NFC (pequeños chips adhesivos), de manera que los usuarios eran capaces de programar acciones automáticas en su teléfono al aproximarlo a ellas.

En la actualidad se utiliza en una amplia variedad de aplicaciones y, aunque se trata de una tecnología segura y que, conforme avanza en el tiempo, se va reforzando cada día más, es necesario

tomar precauciones para evitar el robo de datos o el uso no autorizado. Cabe señalar que se espera que el NFC siga evolucionando y se integre aún más en nuestras vidas cotidianas dada la creciente adopción de Internet de las cosas (IoT) y los pagos sin contacto.

2.3.2 Características

NFC trabaja en la banda de los 13.56 MHz y su funcionamiento es bastante sencillo ya que únicamente intervienen dos dispositivos: el que inicia la comunicación, denominado "iniciador" y el que responde, denominado "objetivo".

Dentro de las características de NFC destaca su velocidad de transferencia de datos, siendo esta de 424 Kbits por segundo. Esto se traduce en que está diseñada para transmitir pequeñas cantidades de información de manera instantánea a una alta frecuencia. Es, por ello, que su empleo se enfoca fundamentalmente en la identificación y validación de equipos/personas. Como contrapartida, el alcance de la tecnología NFC es muy reducido, pues su rango de alcance máximo son los 20 cm.

2.3.3 Uso de la tecnología NFC

El uso de la tecnología NFC se ha visto incrementado en los últimos años y actualmente se emplea en un amplio rango de aplicaciones (ver Figura 2-5), como:

- Pagos móviles: permite realizar pagos de una forma segura y sin contacto. Muchos teléfonos inteligentes y otros dispositivos móviles incluyen NFC.
- Control de acceso: puede ser utilizada para controlar el acceso a edificios, puertas, máquinas, etc.
- Intercambio de información: permite el intercambio de archivos entre dispositivos, como fotos, música, contactos, etc.
- Identificación: utilizando teléfonos celulares con tecnología NFC en lugar de documentos oficiales de identificación.

La evolución de la tecnología NFC ha continuado en los últimos años. En 2019, se lanzó la versión 2.0 de la especificación NFC, que ofrece mejoras en el rendimiento, la seguridad y la compatibilidad.

Figura 2-5. Usos de NFC [17]

2.3.4 Modos de funcionamiento

La tecnología NFC puede funcionar en dos modos diferentes [17]:

- Modo activo: ambos dispositivos participan de manera activa en la comunicación. Este modo permite la comunicación bidireccional, es decir, ambos dispositivos pueden enviar y recibir datos. Es el modo más utilizado en aplicaciones de pago sin contacto, control de acceso y emparejamiento de dispositivos.

- Modo pasivo: solo un dispositivo participa de manera activa en la comunicación. Este modo solo permite la comunicación unidireccional, es decir, el dispositivo activo puede enviar datos al dispositivo pasivo. Es el modo más utilizado en aplicaciones de lectura de etiquetas NFC.

Además de los dos modos anteriormente mencionados, la tecnología NFC también puede funcionar en otros modos, como:

- Modo lector/escritor: un dispositivo puede leer y escribir datos en otro dispositivo.
- Modo emulación de tarjeta: un dispositivo puede emular una tarjeta inteligente, como una tarjeta de crédito o una tarjeta de transporte público.
- Modo P2P: dos dispositivos pueden comunicarse entre sí directamente, sin necesidad de un dispositivo intermediario.

2.3.5 Tarjetas NFC

Las tarjetas NFC (o etiquetas NFC) son dispositivos electrónicos que emplean unos chips con tecnología NFC en su interior y, de esta manera, permiten el intercambio de datos con otros dispositivos. Este chip puede escribirse y leerse mediante el NFC de nuestro dispositivo móvil para, sobre todo, automatizar acciones de forma rápida y sencilla. Existen varios tipos de chips NFC, pero los más comunes son los NTAG e ICODE, siendo los primeros los más utilizados en dispositivos móviles.

Los chips NFC tienen una memoria interna que puede almacenar datos, como números de identificación, credenciales de autenticación o información de pago. Los chips NTAG210, NTAG213 y NTAG215 son los más comunes en dispositivos móviles [15]. El chip NTAG210 tiene 48 bytes de memoria, el NTAG213 tiene 144 bytes y el NTAG215 tiene 540 bytes. Esto significa que el chip NTAG215 puede almacenar hasta 12,5 veces más datos que el chip NTAG210.

La resistencia a la lectura y escritura de un chip NFC se mide en ciclos. Los chips NTAG tienen una resistencia de 100.000 ciclos, lo que se traduce en que pueden leerse y escribirse hasta 100.000 veces sin que se deterioren.

Los chips NTAG son compatibles con la mayoría de los teléfonos inteligentes que tienen NFC y cabe destacar que no funcionan si se colocan sobre una superficie metálica, ya que las ondas de radio no pueden atravesar el metal.

Durante el presente trabajo, sin embargo, estaremos trabajando con tags NFC NXP MIFARE® DESFire® EV2, los cuales siguen el estándar ISO/IEC 14443 y tienen una capacidad de 4094 *bytes*.

2.4 Selección de tecnologías (hardware y software) a emplear

En este apartado, mencionamos otras tecnologías empleadas en el desarrollo de este TFG al margen de NFC, cuya descripción se ha descrito en el apartado anterior.

2.4.1 Desarrollo de una aplicación móvil

Existen varias razones por las que es conveniente desarrollar una aplicación para dispositivos móviles que permita leer tarjetas NFC en un contexto de conflicto.

- Los móviles son dispositivos portátiles y sencillos de usar, ya que, actualmente, prácticamente todo el mundo está familiarizado con ellos (5.320 millones de personas en todo el mundo disponen de un teléfono móvil, lo que supone un 67% de la población mundial total [18]). Esto los hace ideales para su uso en un entorno bélico, donde los rescatadores pueden estar en movimiento o en situaciones peligrosas, lo que les requiere de decisiones rápidas e instintivas.
- Los móviles, en general, tienen una batería de larga duración. Esto es importante cuando nos encontramos en un conflicto, ya que puede resultar difícil acceder a fuentes de energía.

- Los móviles suelen tener un tamaño reducido, lo que supone una ventaja porque permite a los rescatadores llevarlo con ellos sin ocupar mucho espacio. De esta manera, podrán acceder a la aplicación rápidamente para identificar a los combatientes heridos.
- Existen dispositivos móviles rugerizados adaptados a los entornos de uso de las tarjetas NFMC.

2.4.2 Selección del sistema operativo

Android es un sistema operativo móvil desarrollado por Google y basado en el núcleo Linux. Fue lanzado por primera vez en 2008 y, desde entonces, se ha convertido en el sistema operativo móvil más utilizado del mundo, con más de 2.500 millones de dispositivos activos.

Android está diseñado para dispositivos móviles con pantalla táctil, como teléfonos inteligentes, tabletas, relojes inteligentes y televisores. El sistema operativo proporciona una plataforma flexible y escalable que permite a los desarrolladores crear aplicaciones de todo tipo.

A nivel mundial y con fecha de diciembre de 2023, como puede verse en la Figura 2-6, el sistema operativo Android lidera con un 70.41% de cuota de mercado [18], mientras que iOS, el sistema operativo de Apple, ocupa el segundo lugar con un 28.8% durante el mismo período. Por otro lado, el resto de los sistemas operativos suponen una cuota muy pequeña.

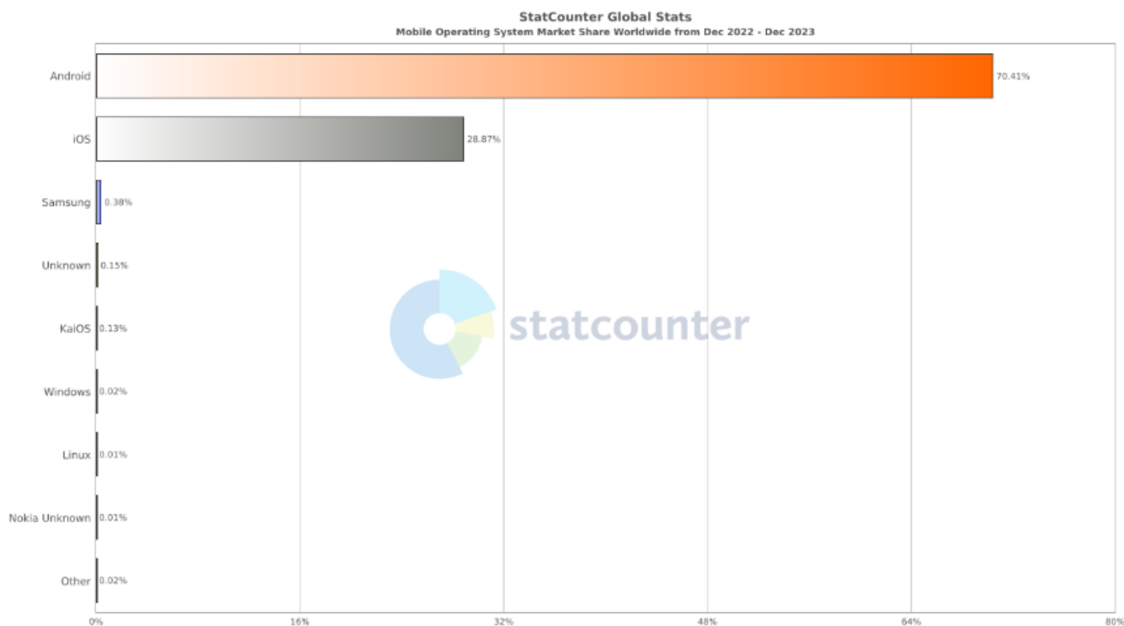


Figura 2-6. Gráfico comparativo del uso de diferentes sistemas operativos para dispositivos móviles [19]

Android e iOS son los sistemas operativos móviles más populares del mundo. Ambos ofrecen una amplia gama de funciones y características, y son compatibles con una gran variedad de dispositivos; sin embargo, existen también una serie de notables diferencias que se resumen en la Tabla 2-1.

	Android	iOS
Origen y desarrollo	Desarrollado por Google. Código fuente abierto.	Desarrollado por Apple. Propietario y cerrado.
Personalización	Altamente personalizable. Permite cambiar la apariencia y usar lanzadores de terceros.	Menos personalizable. Interfaz uniforme y controlada.
Variedad de dispositivos	Amplia variedad de fabricantes y modelos. Desde gama baja hasta alta.	Dispositivos iPhone, iPad y iPod Touch. Gama alta.
Aplicaciones	Google Play Store con muchas aplicaciones gratuitas. Flexibilidad para instalar desde fuentes externas.	Exclusividad de algunas aplicaciones.
Seguridad y privacidad	Mayor riesgo de malware debido a la diversidad de dispositivos. Control de permisos granular.	Mayor seguridad debido al ecosistema cerrado. Control de privacidad más estricto.
Integración con servicios	Integración con servicios de Google como Gmail, Google Drive y Google Maps.	Integración con servicios de Apple como iCloud, iMessage y FaceTime.
Precio y asequibilidad	Amplio rango de precios. Opciones económicas disponibles.	Dispositivos generalmente más costosos. Pocos modelos asequibles.

Tabla 2-1. Comparación entre Android e iOS [fuente propia]

Por tanto, vemos como Android está disponible en un amplio abanico de dispositivos, lo que implica que resulta más sencillo satisfacer las necesidades y el presupuesto de cualquier usuario. Además, al tratarse de un sistema operativo altamente personalizable, permite a los usuarios modificar el aspecto y el comportamiento de su dispositivo según sus preferencias, traduciéndose así en una ventaja para los usuarios que prefieran tener un mayor control sobre su dispositivo. Por último, Google Play Store ofrece una selección más amplia de aplicaciones que la App Store. Esto significa que los usuarios de Android tienen más opciones para encontrar las aplicaciones que necesitan. Teniendo esto en cuenta y dado el gran volumen de usuarios que trabajan con dispositivos Android en la actualidad, se ha decidido implementar la aplicación objeto de este trabajo para que pueda ejecutarse en el sistema operativo Android.

2.4.2.1 Estructura del sistema operativo Android

La estructura del sistema operativo Android está diseñada para ser modular y escalable, lo que permita al sistema operativo adaptarse a una amplia gama de dispositivos. Su estructura (ver Figura 2-7) se compone de las siguientes capas [19]:

- *Linux Kernel*: El núcleo de Linux es el componente esencial del sistema operativo. Su función principal es la de gestionar los recursos del hardware, como la CPU, la memoria y el almacenamiento.

- *Hardware Abstraction Layer* (HAL): Proporciona una capa de abstracción entre el núcleo de Linux y los dispositivos de hardware específicos del dispositivo. De esta manera, el núcleo de Linux es capaz de ser ejecutado en una amplia gama de dispositivos.

- *Android RunTime* (ART): Es el responsable de ejecutar las aplicaciones de Android, así como de transformar el *bytecode* Java o Kotlin en código de máquina nativo, lo que contribuye a mejorar el rendimiento y la eficiencia de las aplicaciones.

- *Native C/C++ Libraries*: Son un conjunto de funciones y servicios esenciales para el correcto funcionamiento de las aplicaciones de Android y están generadas por los desarrolladores de Android.

- *Java API Framework*: El marco de trabajo de la API (*Application Program Interface*) de Java proporciona un entorno para que los desarrolladores creen aplicaciones de Android.

- *System Apps*: Son aplicaciones básicas inherentes al sistema operativo Android. Incluyen, entre otras, aplicaciones como la configuración, el calendario, el reloj y el navegador.

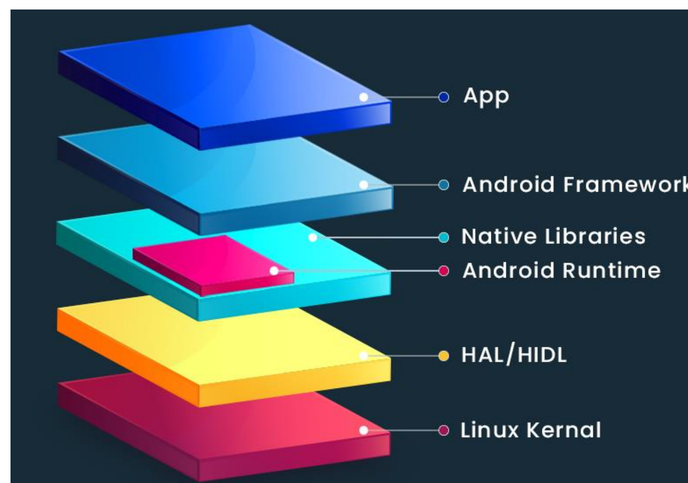


Figura 2-7. Capas de Android [20]

2.4.3 Empleo del IDE de desarrollo y lenguaje de programación

En el presente apartado se expondrá el IDE (*Integrated Development Environment*) de desarrollo empleado y diferentes opciones de lenguajes de programación existentes, así como las ventajas y desventajas que presentan entre ellos. De este modo, se tratará de justificar por qué se han decidido emplear los recursos que se presentarán más adelante.

2.4.3.1 IDE seleccionado

Android Studio [20] es un entorno de desarrollo integrado (IDE) gratuito y de código abierto basado en el editor de código IntelliJ IDEA. Se trata, además, del IDE usado por excelencia en el desarrollo de aplicaciones para Android. Cuenta con numerosas ventajas, entre las que podemos destacar:

- Disponibilidad de un emulador rápido y con multitud de funciones.
- Posee un entorno unificado de tal manera que se pueden desarrollar aplicaciones para todos los dispositivos Android.
- Permite realizar ediciones en tiempo real para actualizar elementos modulables en emuladores y dispositivos físicos.
- Dispone de herramientas para identificar problemas de rendimiento, usabilidad, compatibilidad de versiones, etc.

Una vez dentro de la aplicación es posible diferenciar entre dos modos de visualización de proyecto, siendo estos la vista de Android (establecida por defecto) y la vista Project, que explicaremos en los siguientes apartados.

2.4.3.2 Vista Android

En esta vista se puede observar la organización existente de los módulos y archivos del programa; organizada en una jerarquía lógica. Está diseñada para comprender mejor el proyecto, y tener de manera más sencilla los recursos y elementos más relevantes, tal y como se muestra en la Figura 2-8.

Se organiza en dos partes, una para la *app* y la otra para la compilación del proyecto (*gradle*). Dentro de *app* podemos encontrar [20]:

- *manifests:* contiene el archivo *AndroidManifest.xml*, que es un archivo XML que describe la estructura de la aplicación, sus componentes y sus requisitos. Este archivo es esencial para cualquier aplicación de Android, ya que aporta información relevante acerca de la aplicación al sistema operativo Android, de entre la que podemos destacar:
  - *package*: etiqueta que define el nombre del paquete de nuestra aplicación.
  - *uses-permission*: etiqueta que especifica los permisos que necesita la aplicación para funcionar correctamente.
  - *application*: etiqueta donde están contenidos todos los componentes de la aplicación, como las actividades, los servicios…

- *java:* carpeta donde se almacena el código fuente en Java o Kotlin.

- *res:* contiene los recursos de la aplicación, los cuales son, en su mayoría, ficheros XML. Esta carpeta contiene a su vez cinco subdirectorios:
  - *drawable*: Contiene imágenes de mapa de bits (como PNG y JPEG) o archivos XML que describen formas dibujables.
  - *layout*: Contiene archivos XML que definen la interfaz de usuario de la aplicación.
  - *menu*: Contiene archivos XML que definen menús de opciones.
  - *mipmap*: Contiene iconos de la aplicación en diferentes densidades.
  - *values*: Contiene archivos XML que definen valores simples, como cadenas, colores y dimensiones.

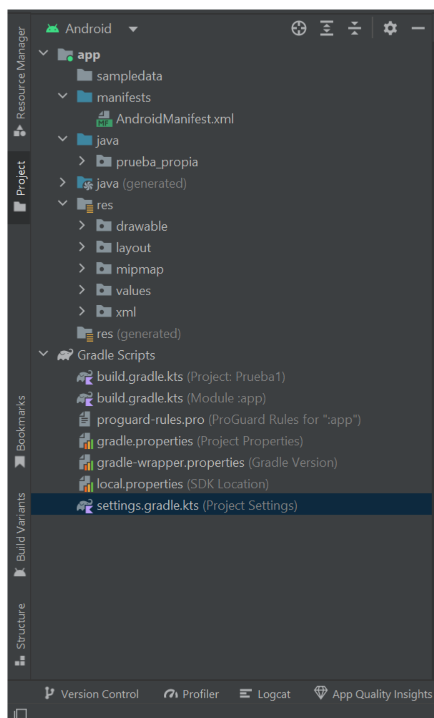


Figura 2-8. Vista de Android en Android Studio [fuente propia]

2.4.3.3 Vista Project

Esta vista nos permite ver la estructura del sistema de ficheros del proyecto como se muestra en la Figura 2-9, lo que incluye también todos los archivos ocultos de la vista de Android. Además, al seleccionarla, también se pueden ver más archivos y directorios, entre los que podemos destacar [21]:

- *app*: Esta carpeta contiene los archivos de código fuente y los recursos para la aplicación principal.
- *build.gradle*: Este archivo contiene la configuración de *Gradle* para el proyecto y los módulos.
- *AndroidManifest.xml*: Este archivo describe la estructura de la aplicación y sus componentes. Además, también nos proporciona información sobre la versión de la aplicación, metadata, etc.
- *res*: Esta carpeta contiene los recursos de la aplicación, como imágenes, diseños y archivos de texto.
- *Gradle Scripts*: Esta carpeta contiene los scripts de *Gradle* para el proyecto y los módulos.

2.4.3.4 Actividades en aplicaciones Android

La clase *Activity* es el componente principal de una aplicación Android que proporciona la interfaz de usuario (IU) para interactuar con este. Cuando una aplicación invoca a otra, la aplicación que realiza la llamada invoca una actividad en la otra, en lugar de a la aplicación en sí. Por lo general, una actividad implementa una pantalla en una aplicación y la mayoría de las aplicaciones contienen varias pantallas, lo que significa que incluyen varias actividades.

Al crearse un nuevo proyecto se genera una actividad principal, que es la primera pantalla que aparece cuando el usuario inicia la aplicación. A partir de ahí, cada actividad puede iniciar otra actividad para realizar diferentes acciones. Además, es necesario que las actividades trabajen en conjunto para crear una experiencia que sea consistente en todos los aspectos de la aplicación, por lo que decimos que existen una serie de dependencias entre las actividades de una aplicación.

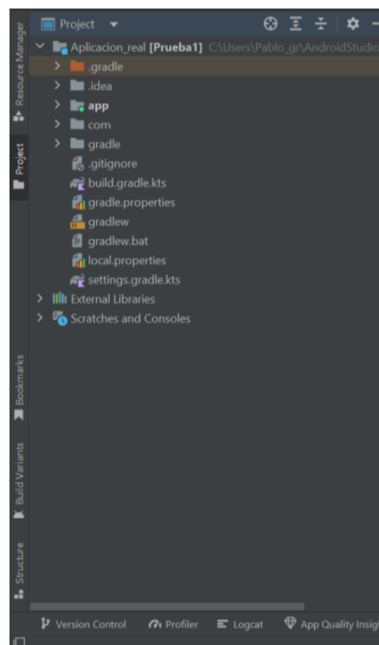


Figura 2-9. Vista Project [fuente propia]

Por lo general, una actividad se compone de los siguientes elementos [22]:

- Métodos: empleados para controlar su comportamiento. Algunos de los métodos más relevantes y que se muestran en la Figura 2-10 son:

- *onCreate*: Este método se llama cuando la actividad se crea por primera vez, permite inicializarla y prepararla para su visualización.
- *onStart*: Este método se llama cuando finaliza *onCreate* y la actividad se vuelve visible para el usuario.
- *onResume*: Este método se llama cuando la actividad se convierte en la actividad activa, es decir, justo antes de que la actividad empiece a interactuar con el usuario.
- *onPause*: Este método se llama cuando la actividad deja de ser la actividad activa; sin embargo, sigue siendo parcialmente visible para el usuario.
- *onStop*: Empleado cuando la actividad ya no es visible para el usuario. A partir de ahí se pueden implementar:
    - *onRestart*: Permite retomar el estado de una actividad, su siguiente método será siempre *onStart*.
    - *onDestroy*: Este método es llamado justo antes de que la actividad sea eliminada.

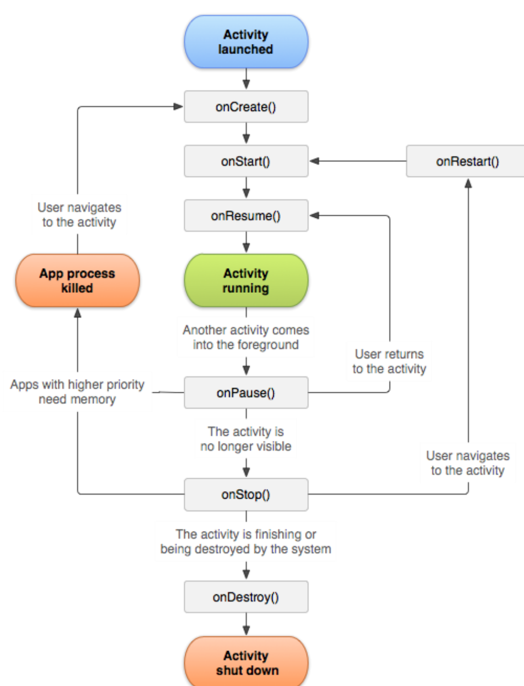


Figura 2-10. Ciclo de vida de una actividad [23]

- Propiedades: Se utilizan para almacenar su estado y acceder a elementos de la interfaz. Algunas de las propiedades más importantes son:
  - *window*: Representa la ventana de la actividad.
  - *contentView*: Representa el contenido de la actividad.
  - *application*: Representa la aplicación a la que pertenece la actividad.

2.4.3.5 Lenguaje de programación empleado

Java [23] es un lenguaje de programación de propósito general, orientado a objetos y multiplataforma. Fue creado en 1995 por Sun Microsystems (ahora Oracle) y se ha convertido en uno de los lenguajes de programación más destacados del mundo. La sintaxis de Java es similar a la de C++, lo que hace que sea más sencilla de aprender para aquellos que ya estén familiarizados con C++.

Kotlin [24], por su parte, fue desarrollado por JetBrains en 2016 para interoperar con Java. Según [25]: "Kotlin es un lenguaje de programación estático de código abierto que admite la programación

funcional y orientada a objetos. Proporciona una sintaxis y conceptos similares a los de otros lenguajes, como C#, Java y Scala, entre muchos otros".

Desde su creación ha ganado popularidad entre los desarrolladores de Android y, en 2017, Google anunció oficialmente el soporte para Kotlin en Android [26]. En los últimos años ha experimentado un crecimiento significativo y se ha posicionado como uno de los lenguajes de programación con mayor expansión en la actualidad. Según las estadísticas proporcionadas por JetBrains [27] "más del 80% de los desarrolladores de Kotlin están satisfechos con el rendimiento de IntelliJ IDEA para Kotlin" y "la cuota de desarrollo móvil multiplataforma ha aumentado en un 50%" entre 2021 y 2022.

Si comparamos Java y Kotlin podemos encontrar diferencias notables como las resumidas en la Tabla 2-2.

	Java	Kotlin
Concisión	Requiere más código para lograr lo mismo.	Permite escribir menos código que Java.
Curva de aprendizaje	Java tiene una curva de aprendizaje más pronunciada que Kotlin al ser más complejo.	Kotlin tiene una curva de aprendizaje más suave que Java. Esto se debe a que se trata de un lenguaje más simple y conciso.
Seguridad	Java tiene una buena corrección de errores, pero no es tan segura como Kotlin. Java carece de soporte nativo para evitar errores comunes como los nulos.	Kotlin tiene una fuerte corrección de errores que ayuda a los desarrolladores a encontrar y corregir errores en su código. Esto es gracias a que utiliza inferencia de tipos y otras características de seguridad.

Tabla 2-2. Diferencias entre Java y Kotlin [fuente propia]

Además, Kotlin cuenta con una sintaxis más moderna y sencilla de leer que Java. Es por todas estas características y ventajas que nos ofrece Kotlin que se ha seleccionado como lenguaje de programación para la realización de este TFG.

2.4.4 Recursos hardware

Los recursos *hardware* con los que contaremos para la realización de este TFG serán:

- Teléfono móvil modelo Xiaomi Redmi 10
- Tarjetas NFC.

En concreto, el teléfono cuenta con las especificaciones técnicas recogidas en la Tabla 2-3.

Dimensiones y peso	161,95 x 75,53 x 8,92 mm y 181 gramos
Pantalla	IPS/LCD de 6,5 pulgadas, resolución FullHD + (2.400 x 1.080 píxeles), 405 ppp, tasa de refresco de 90 HZ, AdaptiveSync y Gorilla Glass 3
Procesador	MediaTek Helio G88 y GPU ARM Mali-G52

Memoria RAM	4 GB LPDDR4x
Almacenamiento interno	64 GB eMMC ampliable con microSD
Cámara trasera	50 MP f/1.8, gran angular 8 MP f/2.2, 120º FOV, macro 2 MP f/2.4 y profundidad 2 MP f/2.4
Cámara delantera	8 MP f/2.0
Batería	5.000 mAh, carga rápida 18W y carga inversa 9W
Sistema operativo	Android 11 con MIUI 12.5
Conectividad	4G, DualSIM, **NFC**, Bluetooth 5.1, Radio FM, GPS, Galileo, GLONASS
Otros	Sensor de infrarrojos, doble altavoz, jack de 3,5 mm lector de huellas lateral y desbloqueo facial

Tabla 2-3. Especificaciones del dispositivo móvil empleado [28]

Además, el dispositivo NFC con el que se trabajará en este proyecto tiene las características recogidas en la tabla 2-4.

Chip NFC	NXP MIFARE® DESFire® EV2
Standard NFC	ISO/IEC 14443A (1-4) / ISO/IEC 7816-4
NFC Forum	Type 4
UID (ID Único)	7 bytes
Memoria disponible	2094 *bytes* – **4094 *bytes*** – 8094 *bytes*
Tasa de comunicación	848 Kbits/s
Resistencia a la lectura/escritura	500.000 ciclos

Conservación de datos	25 años
Funciona en el metal	No
Impermeable	Totalmente
Clase de protección	IP68

Tabla 2-4. Especificaciones del tag NFC empleado [29]

2.5 Trabajos previos

Luego de una exhaustiva revisión de trabajos relacionados con el presente proyecto, se ha logrado identificar únicamente un estudio previo que guarda relación directa con el tema abordado. A continuación, se procederá a resumir la información contenida en dicho estudio.

2.5.1 Proyecto e-SafeTag

El proyecto *e-SafeTag* [30] se llevó a cabo como consecuencia del Acuerdo de Cooperación Industrial entre el Consorcio Iveco Oto Melara –CIO– y el Ministerio de Defensa Español, asociado al suministro de vehículos CENTAURO para el Ejército de Tierra.

Fue concebido con el objetivo principal de desarrollar una aplicación de atención médica que apoye el trabajo de los médicos de las Fuerzas Armadas españolas en diversas circunstancias operativas, tales como misiones de paz, zonas de conflicto o situaciones de emergencia.

Esta aplicación, por lo tanto, permite brindar asistencia al personal sanitario al proporcionar información clínica de los pacientes a través de un chip integrado en la chapa de identificación. Además, al garantizar la recopilación de los datos clínicos del paciente y transmitirlos de forma inmediata a los centros de Control de Misión y de Control de Bajas, facilita la transferencia de información del herido entre profesionales sanitarios.

Dicho proyecto se estructura de la siguiente manera:

- Primera fase de desarrollo: Se inicia con la realización de un triaje clínico inicial y, posteriormente, se lee la información del combatiente que va incluida en la chapa de identificación personal, lo que aporta seguridad a la hora de realizar ciertas maniobras con el paciente. A continuación, ya se comienzan a rellenar diversos apartados de la aplicación de tal modo que será esta la que indique en qué estado se encuentra el paciente (T1-Alfa-Rojo, T-2-Bravo-Amarillo, T3-Charlie-Verde, T4-Delta-Azul [30]). Esta información, junto con la que se vaya añadiendo a la aplicación, le es remitida al centro de control de la misión a través de radio táctica. Durante esta primera fase también se desarrollan una serie de registros temporales que contienen información acerca de la medicación administrada. El sistema tiene además la opción de ofrecer un resumen de la situación del paciente.
- Segunda fase de desarrollo: Se introduce el uso de dispositivos ANNOTO (bolígrafos) que permiten seguir trabajando de forma clásica, es decir, sobre una plantilla de papel, pero que envían toda la información a los centros de control en formato similar al que lo hacen los dispositivos informáticos.

Es importante resaltar que, hasta donde llega nuestro conocimiento, no se ha encontrado evidencia alguna de la implementación de la solución propuesta en dicho proyecto, ni de la existencia de actualizaciones o investigaciones posteriores sobre la misma.

3 DESARROLLO DEL TFG

En este apartado, se presentarán inicialmente los criterios requeridos en la fase de diseño de la aplicación, junto con las distintas alternativas consideradas. A partir de un análisis de las ventajas y desventajas de cada una, se procederá a la selección de una opción definitiva. Asimismo, se abordará la estructura de diseño de la aplicación y los elementos integrantes en un proyecto de Android. Por último, se describirá el progreso en la implementación y codificación de cada una de las actividades que conforman la aplicación.

3.1 Requisitos de la aplicación

El propósito de este Trabajo de Fin de Grado (TFG) es diseñar, desarrollar y validar una aplicación para dispositivos Android que, aprovechando la tecnología NFC, permite acceder y modificar, de manera rápida y eficiente, los datos médicos básicos de cualquier persona herida en un entorno de combate. Para abordar inicialmente este proyecto, consideraremos que todos los rescatadores llevarán consigo un dispositivo móvil con sistema operativo Android con conexión a red en zonas de cobertura móvil y con capacidad NFC, y que cada combatiente portará una tarjeta NFC que contendrá su información personal y médica básica (nombre, apellidos, alergia a medicamentos, etc.), siguiendo directrices OTAN.

En la primera versión de la aplicación a desarrollar, no considerarán los aspectos de seguridad de los datos y de las comunicaciones dadas las restricciones temporales para la realización de este TFG, pero son cuestiones fundamentales para el despliegue de esta aplicación y deberían realizarse en el futuro como se expondrá en el apartado de líneas futuras.

3.1.1 Análisis de soluciones planteadas

En esta sección se presentarán las soluciones investigadas en la fase inicial de este TFG. Asimismo, se llevará a cabo un análisis de las ventajas y desventajas asociadas a cada opción, con el objetivo de llegar a una conclusión fundamentada sobre la elección final del diseño de la aplicación tal como se expone en este trabajo.

3.1.1.1 Almacenamiento mínimo de información del paciente en la tarjeta NFC

En esta configuración, se propone únicamente almacenar en la tarjeta NFC el número de identificación del paciente, es decir, su ID. Todos los datos restantes se envían a un servidor (ver Figura 3-1).

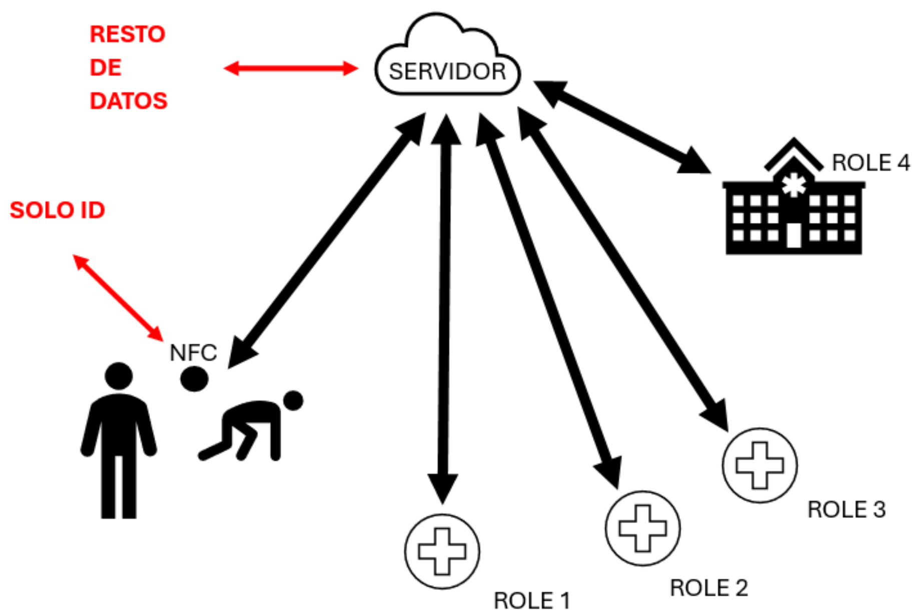


Figura 3-1. Esquema de la primera configuración propuesta [fuente propia]

3.1.1.2 Almacenamiento de todos los datos en la tarjeta NFC

Esta alternativa implica el almacenamiento de todos los datos del paciente en el tag NFC, excluyendo el envío de información al servidor (ver Figura 3-2).

Figura 3-2. Esquema de la segunda configuración propuesta [fuente propia]

3.1.1.3 Combinación de los enfoques anteriores

Esta opción propone una combinación de las estrategias previas. Se almacena toda la información tanto en el tag NFC como en el servidor web (ver Figura 3-3).

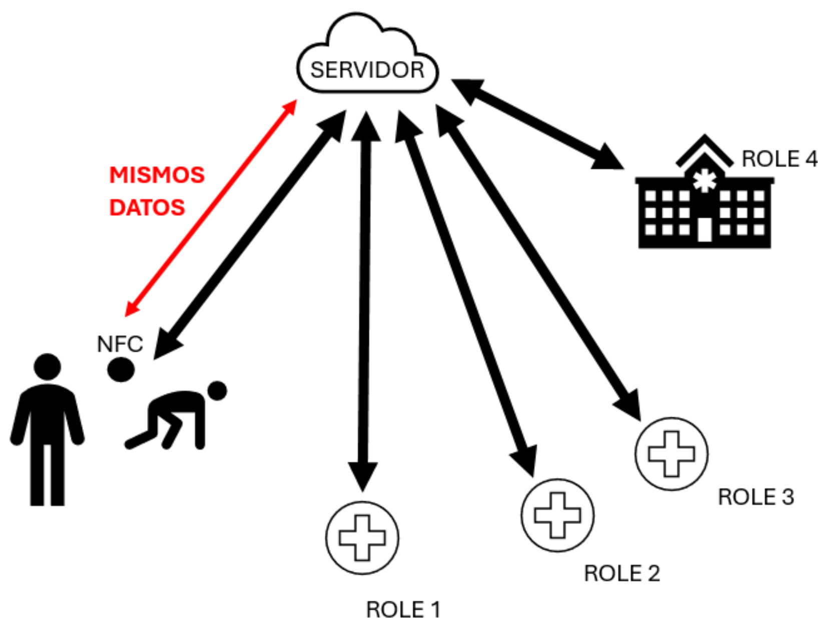


Figura 3-3. Esquema de la tercera configuración propuesta [fuente propia]

3.1.1.4 Elección de la solución

La Tabla 3-1 se muestra las ventajas y desventajas, representadas en color verde y rojo respectivamente, de las diferentes opciones contempladas en los tres subapartados anteriores.

Característica	Almacenar solo ID en el tag	Almacenar todo en el tag	Combinación de ambas
Riesgo de pérdida del historial del paciente	No	Sí	No
Dependencia de la conectividad de red para acceder a la información completa	Sí	No	No
Vulnerabilidad de seguridad durante la transmisión de datos entre la aplicación móvil y el servidor web	Sí	No	Sí
Requiere de una infraestructura más compleja	Sí	No	Sí
Existe la posibilidad de acceder simultáneamente a la información del herido desde otras FST	Sí	No	Sí
Ofrece redundancia y respaldo de datos	Sí	No	Sí

Tabla 3-1. Ventajas y desventajas de cada opción planteada [fuente propia]

Adicionalmente, tal y como se indica en la PDC-4.10 Doctrina sanitaria en operaciones [6], "toda actividad asistencial realizada sobre una baja debe quedar registrada para garantizar su trazabilidad (*patient traceability*). Este registro acompañará al paciente a lo largo de la cadena asistencial. El uso de sistemas de información y telecomunicaciones en el apoyo sanitario a las operaciones debe facilitar la transmisión anticipada de dicha información a los siguientes elementos y formaciones de tratamiento, lo que redundará en una mejor disposición para la recepción de la baja y posterior tratamiento".

También se establece en este mismo documento que "los CIS implicados en este proceso deben ser capaces de facilitar, como se ha señalado en el apartado anterior, el seguimiento de la baja. Además, obtendrán en tiempo real información sobre los niveles de ocupación y capacidades de las FST, así como la situación y disponibilidad de los medios de evacuación. Todo ello se integrará en la Imagen Sanitaria dentro de la COP (*Common Operational Picture* en la OTAN) y tendrán las funcionalidades requeridas para apoyar la decisión en incidentes de bajas masivas (MASCAL) o incidentes graves".

Por tanto, se ha seleccionado la última configuración como la más adecuada ya que combina las ventajas de los dos primeros esquemas ideas mientras minimiza las desventajas. Proporciona redundancia y respaldo de datos al almacenarlos tanto en la tarjeta NFC como en el servidor web, lo que reduce el riesgo de pérdida de datos. Además, ofrece flexibilidad y accesibilidad al permitir la consulta de los datos de cualquier paciente desde cualquier ubicación con conexión a Internet, de tal modo que posibilita la transmisión anticipada de datos críticos a los elementos de tratamiento siguientes y facilita la toma de decisiones informadas en tiempo real.

3.2 Diseño de la aplicación

Dado que estamos trabajando con un estándar OTAN como son las NFMC, se ha tomado la decisión de implementar la aplicación en inglés.

La aplicación está estructurada como una actividad principal que permite el inicio de sesión introduciendo el DNI y una contraseña. Una vez concedido el acceso, el usuario accede al menú principal en el cual se encuentran dispuestos diversos elementos de la interfaz de usuario para facilitar la navegación a través del resto de actividades (tal y como se muestra en la Figura 3-4) y acceso a la información del paciente. Cabe destacar que la autenticación que se está llevando a cabo es a nivel local; sin embargo, sería necesaria la implementación de métodos de seguridad más complejos en posibles versiones futuras de la aplicación.

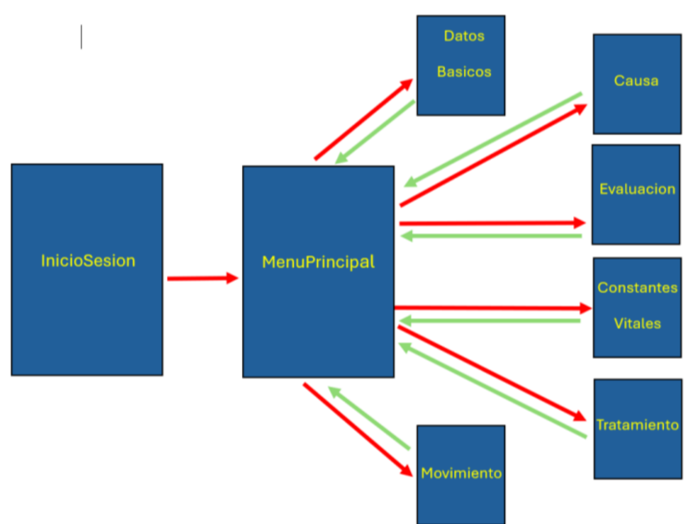


Figura 3-4. Esquema que muestra el flujo de información entre actividades [fuente propia]

La pantalla principal está diseñada con una serie de botones (ver Figura 3-5) que actúan como un menú y permiten al usuario acceder a diferentes secciones de la aplicación, cada una enfocada en aspectos específicos relacionados con la evacuación de la baja. Además, también existen dos botones

que permiten realizar las operaciones de lectura y escritura en la tarjeta NFC. Se incluyen también una serie de instrucciones en la parte inferior de la pantalla con el objetivo de facilitar la comprensión del funcionamiento de la aplicación.

Las diferentes pantallas en las que está organizada la aplicación incluyen campos de texto que permiten al usuario ingresar información que se desea escribir en la tarjeta NFC. Estos campos son elementos interactivos que facilitan la entrada de datos y están diseñados para adaptarse a diferentes tamaños de pantalla y orientaciones del dispositivo.

Para facilitar el uso de la aplicación, el usuario recibe constantemente mensajes de lo que está sucediendo, como cambios de modo (escritura o lectura), resultados de las operaciones NFC (éxito o error), así como una notificación en caso de haber introducido un fecha u hora incongruente.

El diseño de la aplicación busca ofrecer una experiencia fluida y eficiente al usuario, centrándose en la claridad de la interfaz, la facilidad de navegación y la retroalimentación informativa en todo momento.

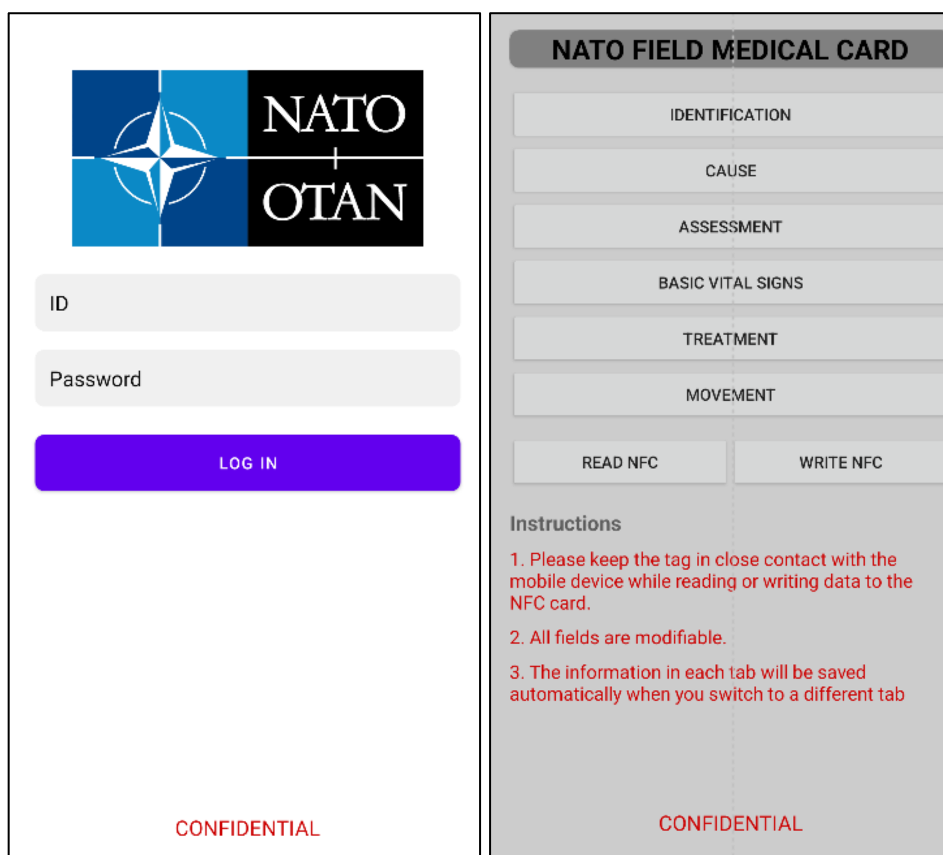


Figura 3-5. Pantallas de Inicio de sesión y menú principal [fuente propia]

3.3 Desarrollo de la aplicación

A la hora de desarrollar una aplicación para Android, una de las características más importantes para tener en cuenta es la elección de la versión del sistema operativo, ya que, en función de ello, podrá ser ejecutada o no en determinados dispositivos móviles. Es por ello por lo que, en este caso, se ha seleccionado una *minSdkVersion* 24, lo que indica que la aplicación es compatible con dispositivos que ejecutan Android 7.0 (Nougat) o superior, como se muestra en la Figura 3-7.

De este modo, como muestra la Figura 3-6, únicamente un porcentaje inferior al 2,11% de todos los dispositivos móviles que ejecutan actualmente Android no serían compatibles con la aplicación desarrollada.

CODE NAME	VERSION	API LEVEL
Pie	**9**	**API level 28**
Oreo	8.1.0	API level 27
Oreo	8.0.0	API level 26
Nougat	7.1	API level 25
Nougat	7.0	API level 24
Marshmallow	6.0	API level 23
Lollipop	5.1	API level 22
Lollipop	**5.0**	**API level 21**
KitKat	4.4 – 4.4.4	API level 19
Jelly Bean	4.3.x	API level 18
Jelly Bean	4.2.x	API level 17
Jelly Bean	4.1.x	API level 16
Ice Cream Sandwich	4.0.3 – 4.0.4	API level 15, NDK 8
Ice Cream Sandwich	4.0.1 – 4.0.2	API level 14, NDK 7
Honeycomb	3.2.x	API level 13
Honeycomb	3.1	API level 12, NDK 6
Honeycomb	3.0	API level 11
Gingerbread	2.3.3 – 2.3.7	API level 10
Gingerbread	2.3 – 2.3.2	API level 9, NDK 5
Froyo	2.2.x	API level 8, NDK 4
Eclair	2.1	API level 7, NDK 3
Eclair	2.0.1	API level 6
Eclair	2.0	API level 5
Donut	1.6	API level 4, NDK 2
Cupcake	1.5	API level 3, NDK 1
(no code name)	1.1	API level 2
(no code name)	1.0	API level 1

Figura 3-6. Asociación de las versiones de Android con sus niveles API [31]

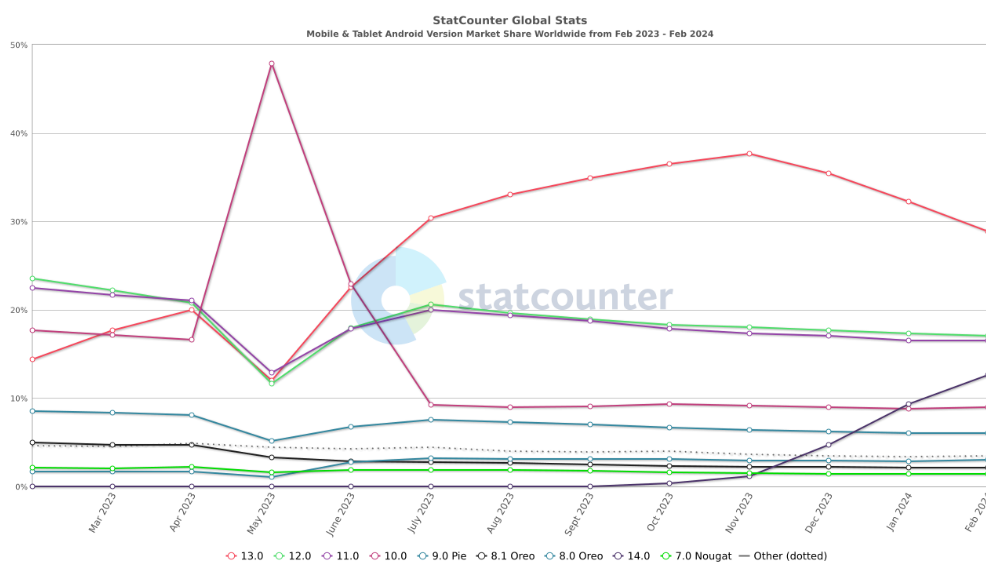


Figura 3-7. Gráfico que representa el porcentaje de uso de las diferentes versiones del sistema operativo Android [32]

3.3.1 Estructura del proyecto Android

En esta sección, se explicará la estructura fundamental de un proyecto Android, destacando los componentes esenciales que la forman. Desde el archivo *Manifest.xml*, que define la configuración general de la aplicación, hasta las actividades que constituyen las distintas pantallas y funcionalidades. También abordaremos el uso de *data classes* para representar y manipular datos, así como la importancia de los *layouts* para el diseño de la interfaz de usuario. Además, examinaremos las dependencias, que son bibliotecas y recursos externos utilizados para ampliar la funcionalidad del proyecto.

3.3.1.1 Android Manifest

Como se explica en el apartado 2.4.3.2, *AndroidManifest.xml* es un archivo fundamental en *Android* sin el cual la aplicación no puede funcionar. Se localiza en el directorio raíz del proyecto y, cuando se construye una aplicación, este elemento se empaqueta dentro del archivo APK (*Android Package*). El archivo APK es el paquete que contiene todos los recursos y archivos necesarios para instalar y ejecutar la aplicación en dispositivos Android. El archivo *AndroidManifest.xml* contiene parámetros como:

- Nombre de la aplicación.
- Componentes de la aplicación (actividades, servicios, proveedores de contenido, etc.).
- Permisos necesarios para que la aplicación funcione correctamente.
- Requisitos del sistema (versión mínima de Android, características del dispositivo, etc.).
- Interacciones con otras aplicaciones.

Una de las características fundamentales del *AndroidManifest,* al igual que sucede con todos los archivos XML, es su estructura jerárquica bien definida. El elemento raíz del archivo *AndroidManifest* es <*manifest*>, que contiene:

- <*application*>: Define la configuración general de la aplicación.
- <*activity*>: Describe cada una de las pantallas o actividades de la aplicación. Todas las actividades que se vayan a ejecutar deben estar definidas bajo este elemento.
- <*service*>: Define un servicio en segundo plano que se ejecuta independientemente de la actividad actual. A diferencia de las actividades, los servicios no poseen una interfaz de usuario visible.
- <*provider*>: Define un proveedor de contenido que permite compartir datos con otras aplicaciones.
- <*uses-permission*>: Define un permiso que la aplicación necesita para acceder a recursos del sistema o de otras aplicaciones.

En el desarrollo de nuestra aplicación, tal y como se observa en la Figura 3-8, se requieren dos permisos: el acceso a la funcionalidad NFC y a Internet. Además, dentro de su configuración general se especifican varios atributos como el nombre de la aplicación, la compatibilidad con la dirección de escritura de derecha a izquierda (*android:supportsRtl*) [33], el tema de la aplicación, entre otros.

Por otro lado, se definen varias actividades entre las que destaca *InicioSesion,* definida como la actividad principal *(android.intent.action.MAIN*), lo que significa que será la primera actividad que se inicie cuando se ejecute la aplicación. Otras actividades tienen filtros de *intents* que les permiten responder a acciones específicas, como la detección de etiquetas NFC (*android.nfc.action.NDEF_DISCOVERED*), es decir, pueden manejar las acciones asociadas con estos *intents*.

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <uses-permission android:name="android.permission.NFC" />
    <uses-permission android:name="android.permission.INTERNET" />

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.Prueba1"
        android:networkSecurityConfig="@xml/network_security_config"
        tools:targetApi="31">

        <activity
            android:name="NFMC.InicioSesion"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name="NFMC.MenuPrincipal"
            android:exported="true">
            <intent-filter>
                <action android:name="android.nfc.action.NDEF_DISCOVERED" />

                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
        </activity>
        <activity
            android:name="NFMC.DatosBasicos"
            android:exported="false" />
        <activity
            android:name="NFMC.RegistroAdapter"
            android:exported="false"
            tools:ignore="Instantiatable" />
        <activity
            android:name="NFMC.ConstantesVitales"
            android:exported="false" />
        <activity
            android:name="NFMC.Movimiento"
            android:exported="false" />
        <activity
            android:name="NFMC.Tratamiento"
            android:exported="false" />
        <activity
            android:name="NFMC.Evaluacion"
            android:exported="false" />
        <activity
            android:name="NFMC.Causa"
            android:exported="false" />

    </application>
</manifest>
```

Figura 3-8. Contenido del fichero *AndroidManifest.xml* de la aplicación [fuente propia]

3.3.1.2 Actividades

Como se explica en el apartado 2.4.3.2, cada actividad representa una pantalla visible para el usuario y constituye la interfaz gráfica con la que interactúa. En este sentido, una aplicación puede estar compuesta por múltiples actividades, cada una dedicada a una funcionalidad específica. Las actividades se componen de dos elementos fundamentales:

- Parte lógica (archivo Kotlin):
    - La parte lógica de una actividad está definida por su archivo .Java o Kotlin. Aquí se implementa la funcionalidad específica de la *Activity*, como la lógica, la gestión de datos y las interacciones con otros componentes.

  - Por ejemplo, en nuestra aplicación de "Menú Principal", existe una lógica que permite la escritura/lectura de tarjetas NFC, así como la visualización de su resultado en diferentes campos de texto.
- Parte gráfica (archivo XML):
  - La parte gráfica de una actividad se describe mediante un archivo XML (*Extensible Markup Language*). Este archivo contiene los elementos visuales que componen la pantalla, como botones, campos de texto, imágenes y *layouts*.
  - Mediante etiquetas XML, definimos la disposición y apariencia de los elementos en la pantalla. Por ejemplo, un archivo XML puede especificar la posición relativa de los botones y el estilo de fuente de los textos.

Desde una actividad es posible iniciar otra actividad. Esto es especialmente relevante cuando trabajamos con funcionalidades como la lectura y escritura de tarjetas NFC. Por ejemplo, al tocar un botón en la actividad principal, podemos cubrir los cuadros de texto de otra actividad a partir de los datos existentes en una tarjeta NFC.

Las actividades dentro de la aplicación desarrollada en este TFG, tal y como se muestra en la Figura 3-8 son las siguientes:

- *InicioSesion:* donde el usuario deberá ingresar su ID y una clave.
- *MenuPrincipal:* permite gestionar datos relacionados con identificación, evaluación, tratamiento y movimientos de los heridos. Utilizando la tecnología *Near Field Communication*, los usuarios pueden leer y escribir información en etiquetas NFC. Además, también es posible modificar datos relacionados con la información básica del paciente**,** evaluaciones médicas, causa de lesión, detalles de tratamiento y datos de los movimientos llevados a cabo. Además, al escribir los datos en la tarjeta NFC también se enviarán al servidor online.
- *DatosBasicos:* permite capturar, mostrar y editar los datos básicos de identificación del herido, así como la nacionalidad de las fuerzas armadas en las que está integrado.
- *Causa:* permite definir de las zonas en donde existe una lesión, así como el tipo de lesión, empleando para ello símbolos estandarizados y recogidos en el *AMedP-8.1 NATO Standard* [3].
- *Evaluación:* permite capturar, mostrar y editar los datos relacionados con la evaluación realizada al herido.
- *ConstantesVitales*: permite llevar a cabo un registro en el tiempo de las diferentes constantes vitales del paciente, así como de su estado y la escala de dolor que siente.
- *Tratamiento:* permite gestionar los datos relacionados con el tratamiento (analgésicos, antibióticos, realización de algún torniquete, etc) llevado a cabo por la primera persona en asistir al combatiente.
- *Movimiento:* permite gestionar datos relacionados con la categoría de la evacuación, información básica acerca de la primera persona en asistir al herido y posibles notas adicionales que se necesiten.
- *RegistroAdapter*: organiza y muestra datos de registros médicos en una lista. Utiliza un *RecyclerView* para mostrar cada registro de forma eficiente. Cada elemento de la lista se visualiza mediante un diseño personalizado llamado *activity_registro_constantes.xml*. La actividad *RegistroAdapter* administra la asignación de datos a las vistas de cada registro para su visualización en la interfaz de usuario.

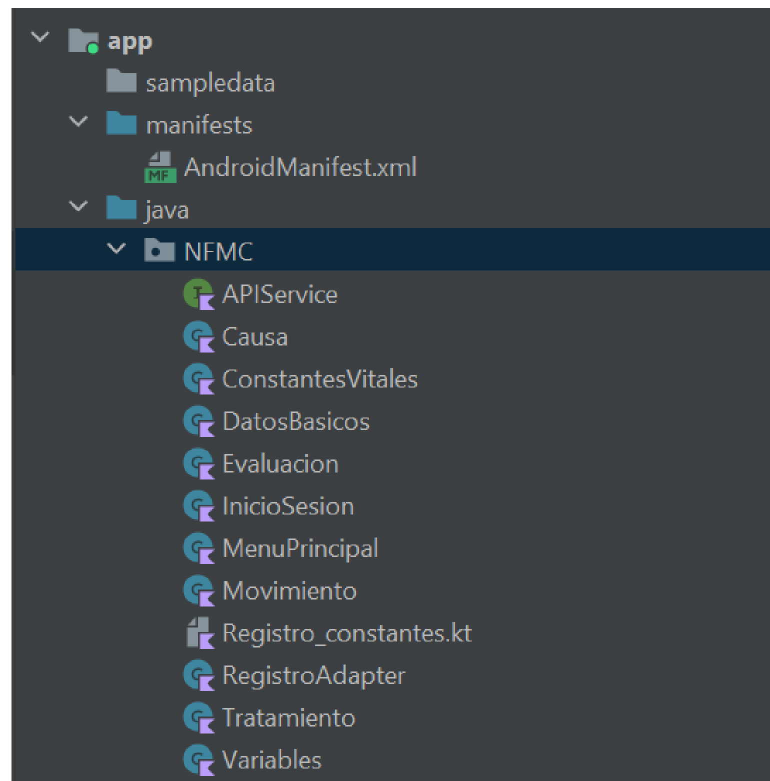


Figura 3-9. Actividades de la aplicación [fuente propia]

Todas las actividades llamadas desde *MenuPrincipal* reciben los campos que gestionan con los datos leídos de la tarjeta NFC (en el caso de lectura previa). Además, existen ciertos apartados relacionados con la fecha y la hora que se completan de manera automática con los datos del dispositivo móvil para asistir al usuario.

Los usuarios cuentan con la capacidad de editar esta información y, al seleccionar el botón de retroceso, la información de los datos modificados es devuelta a la actividad principal (*MenuPrincipal*). Cabe destacar que se implementa una comprobación en caso de la introducción errónea de fechas y horas, de manera que, si esto sucediese, el usuario no podría retroceder y le aparecería un mensaje en pantalla informándole de lo que está pasando.

3.3.1.3 Data Class

La estructura de *Data Class* en Android es una estructura especial de clase que se utiliza principalmente para encapsular datos, de manera que este tipo de clases contiene solo propiedades (campos) y métodos básicos para acceder a ellas (como *getters* y *setters*). Estas clases son simplemente contenedores de datos utilizados por otras clases cuyo objetivo principal es almacenar información de manera eficiente y ordenada.

Además de diferenciarse de las actividades por utilizarse estas para representar una pantalla con una interfaz de usuario, existen también otras diferencias:

- Propósito: Las *Data Classes* almacenan datos, mientras que las actividades son responsables de la lógica de la interfaz de usuario y la navegación.
- Estructura: Las *Data Classes* son simples y contienen solo propiedades, mientras que las actividades son más complejas y pueden incluir múltiples métodos y eventos.
- Uso: Las *Data Classes* se utilizan para representar objetos de datos (por ejemplo, usuarios, productos), mientras que las actividades son componentes de la aplicación que interactúan con el usuario.

- Ciclo de Vida: Las *Data Classes* no tienen un ciclo de vida específico, mientras que las actividades pasan por estados como *onCreate*, *onResume*, etc.

En el desarrollo específico de nuestra aplicación se han creado las *Data Classes* que se detallan a continuación y que pueden verse en la Figura 3-9:

- *Registro_constantes*: es un modelo de datos que representa un registro médico con atributos como hora, pulso, presión sanguínea, entre otros. Implementa la interfaz *Parcelable* para permitir que los objetos de esta clase puedan ser transferidos entre componentes de la aplicación a través de *Parcel*. Además, define un constructor secundario que recibe un *Parcel* para la lectura de los datos y métodos para escribir los contenidos del *Parcel*. También, proporciona un objeto *Creator* que implementa las funciones para crear un array de objetos y para crear un objeto *Registro* a partir de un *Parcel*. Esto facilita la transferencia de datos entre componentes de la aplicación en Android.
- *Variables*: sirve como un contenedor de múltiples variables que representan los campos de la tarjeta NFC.

3.3.1.4 Layouts

Los *layouts* son archivos XML que definen la estructura visual de una pantalla en una aplicación Android. Son una parte fundamental del desarrollo de interfaces de usuario en Android. Entre sus funciones principales destacan:

- Organización de elementos: Permiten organizar los elementos visuales de la pantalla, como botones, imágenes, textos y otros *widgets*, en una estructura jerárquica.
- Definición de propiedades: Establecen las propiedades de cada elemento visual, como su tamaño, posición, color, estilo y otras características.
- Reutilización de diseños: Facilitan la reutilización de diseños comunes en diferentes pantallas de la aplicación.
- Compatibilidad con diferentes dispositivos: Permiten crear diseños adaptables que se ajustan a diferentes tamaños de pantalla y orientaciones del dispositivo.

En nuestro caso se han empleado los siguientes recursos para el desarrollo de la aplicación:

- *ImageView*: Se emplea para mostrar imágenes, como un logotipo o una foto. Puede configurarse para mostrar imágenes desde recursos locales o desde URL externas.
- *EditText*: Este elemento permite al usuario introducir texto o números. Normalmente es usado para ingresar información como nombres de usuario, contraseñas, u otros datos.
- *TextView*: Se usa para mostrar texto en la interfaz de usuario. Puede contener texto estático o ser actualizado dinámicamente en respuesta a eventos o cambios en la aplicación.
- *Button*: Representa un botón que el usuario puede pulsar para realizar una acción específica. En la aplicación, se utiliza para activar eventos como el inicio de sesión o el envío de datos.

Por otro lado, existen diferentes contenedores que permiten organizar los elementos gráficos de diferentes maneras [34]:

- *LinearLayout*: Organiza los elementos en una sola fila o columna.
- *RelativeLayout*: Posiciona los elementos de forma relativa a otros elementos o a la pantalla.
- *FrameLayout*: Posiciona los elementos uno encima del otro.
- *TableLayout*: Organiza los elementos en una tabla con filas y columnas.
- *ConstraintLayout*: Posiciona los elementos mediante restricciones, lo que facilita la creación de diseños complejos.
- *ScrollView*: Es un contenedor que permite desplazamiento vertical dentro de una vista.

En la Figura 3-10 y en la Figura 3-11 se muestran la vista diseño y código, respectivamente, de la pantalla de inicio. En ambas figuras se observan algunos de los elementos previamente mencionados,

como un *ImageView* (representando el logo de la OTAN), un *Button* (de color azul, el cual, si los datos introducidos son correctos, nos dirige al *MenuPrincipal*), y dos *EditText*. Estos últimos muestran inicialmente un texto ("*ID*" y "*Password*") para facilitar la comprensión de su propósito al usuario y permiten la introducción de datos correspondientes.

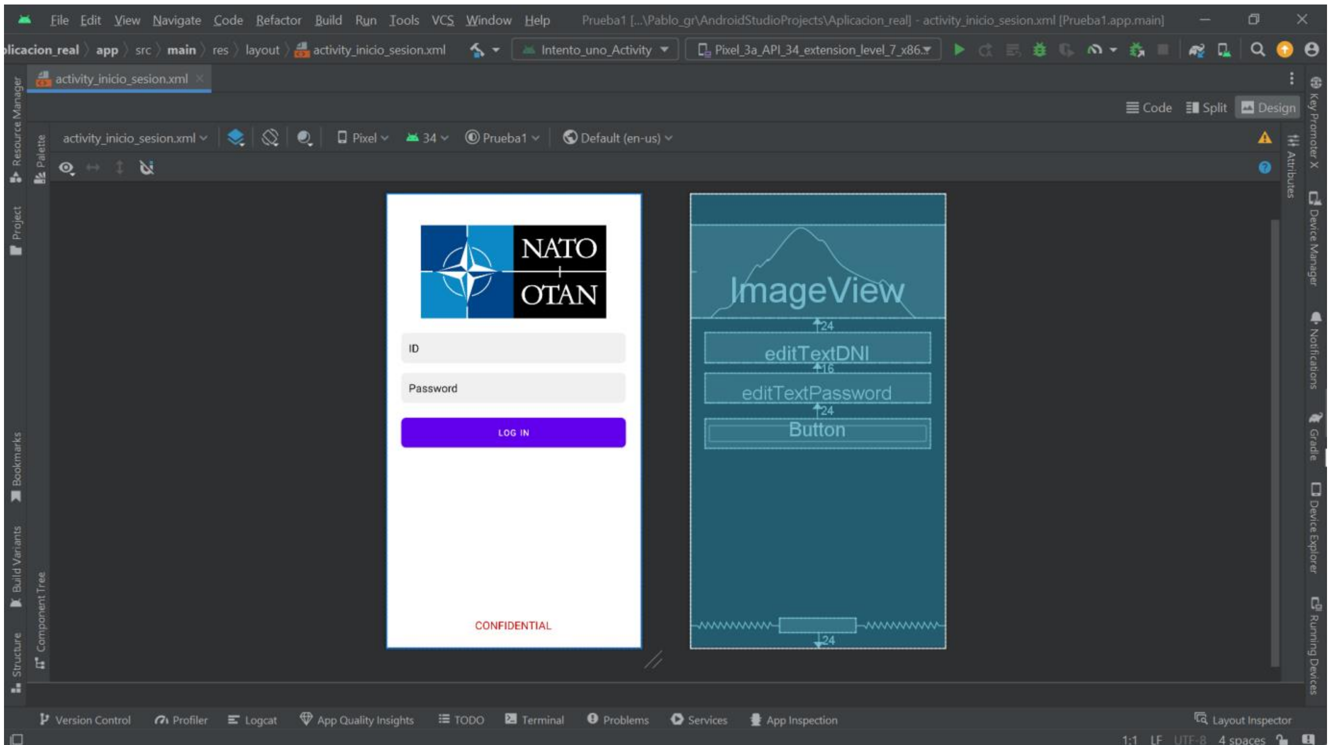


Figura 3-10. Vista diseño del *layout* de *InicioSesion* [fuente propia]

```
<EditText
    android:id="@+id/editTextPassword"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_below="@id/editTextDNI"
    android:layout_marginStart="24dp"
    android:layout_marginTop="16dp"
    android:layout_marginEnd="24dp"
    android:background="@drawable/edit_text_rounded_corners"
    android:hint="Password"
    android:inputType="textPassword"
    android:padding="12dp"
    android:textColorHint="@color/black" />

<Button
    android:id="@+id/buttonLogin"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_below="@id/editTextPassword"
    android:layout_marginStart="24dp"
    android:layout_marginTop="24dp"
    android:layout_marginEnd="24dp"
    android:background="@drawable/button_rounded_corners"
    android:text="Log in"
    android:textColor="@color/white" />
```

Figura 3-11. Parte del código del *layout* de *InicioSesion* [fuente propia]

3.3.1.5 Dependencias

Las dependencias en Android son archivos o módulos externos que un proyecto necesita para funcionar correctamente. Estas dependencias pueden ser:

- Bibliotecas: Proporcionan funcionalidades predefinidas que se pueden utilizar en el proyecto, como acceso a redes, procesamiento de imágenes, o gestión de bases de datos.
- Módulos: Son partes independientes del proyecto que se pueden reutilizar en otros proyectos.

- Archivos: Pueden ser archivos de código fuente, recursos (imágenes, sonidos, etc.) o archivos de configuración.

Existen dos tipos principales de dependencias en Android:

- Dependencias locales: Localizadas en el mismo proyecto que la aplicación.
- Dependencias remotas: Localizadas en un repositorio remoto, como *Maven Central* o *Google Maven Repository*.

Para el desarrollo de esta aplicación se han empleado las dependencias listadas en la Figura 3-12, que se explican a continuación:

- *androidx.core:core-ktx*: Proporciona extensiones de Kotlin para APIs de Android, lo que facilita la escritura de código más conciso y legible.
- *androidx.appcompat:appcompat*: Proporciona compatibilidad con versiones anteriores de Android para las características de la biblioteca de soporte de Android.
- *com.google.android.material:material*: Implementa los componentes de *Material Design* en la aplicación.
- *androidx.constraintlayout:constraintlayout*: Proporciona un diseño flexible para la interfaz de usuario.
- *io.github.vicmikhailau:MaskedEditText*: Ofrece un *EditText* con formato de máscara para la entrada de datos a partir del repositorio.
- *com.squareup.picasso:picasso:2.71828*: Picasso es una biblioteca de carga y visualización de imágenes para Android. Facilita la carga de imágenes desde recursos locales, recursos en línea o incluso en caché, y las muestra de manera eficiente en las vistas de la interfaz de usuario.
- *com.squareup.retrofit2:retrofit:2.9.0*: *Retrofit* es una biblioteca de cliente HTTP para Android y Java que simplifica la realización de solicitudes a servicios web *RESTful* (*Representational State Transfer*). Facilita la creación de solicitudes HTTP, el manejo de respuestas y la conversión de los datos recibidos en objetos Java.
- *com.squareup.retrofit2:converter-gson:2.9.0*: Este es un complemento para *Retrofit* que permite la conversión automática de los datos JSON (*JavaScript Object Notation*) recibidos en objetos Java utilizando la biblioteca *Gson*. *Gson* es una biblioteca de serialización/deserialización de JSON muy utilizada en el ecosistema de desarrollo de Android.
- *org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.6*: Las *coroutines* son una característica de Kotlin que permite escribir código asíncrono (realizar múltiples tareas simultáneamente sin bloquear el hilo de ejecución principal) de forma más sencilla y concisa. Esta biblioteca proporciona soporte específico para Android, permitiendo la ejecución de operaciones asíncronas de manera segura en el hilo principal de la interfaz de usuario.
- *junit:junit*: Utilizado para escribir y ejecutar pruebas unitarias en la aplicación.
- *androidx.test.ext:junit*: Proporciona soporte para la ejecución de pruebas de unidad en Android.
- *androidx.test.espresso:espresso-core*: *Framework* para escribir pruebas de interfaz de usuario (UI) en Android.

```
dependencies {  this: DependencyHandlerScope

    implementation("androidx.core:core-ktx:1.9.0")
    implementation("androidx.appcompat:appcompat:1.6.1")
    implementation("com.google.android.material:material:1.8.0")
    implementation("androidx.constraintlayout:constraintlayout:2.1.4")
    implementation("io.github.vicmikhailau:MaskedEditText:5.0.1")
    implementation("com.squareup.picasso:picasso:2.71828")
    implementation("com.squareup.retrofit2:retrofit:2.9.0")
    implementation("com.squareup.retrofit2:converter-gson:2.9.0")
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.6")
    testImplementation("junit:junit:4.13.2")
    androidTestImplementation("androidx.test.ext:junit:1.1.5")
    androidTestImplementation("androidx.test.espresso:espresso-core:3.5.1")

}
```

Figura 3-12. Dependencias de la aplicación [fuente propia]

3.3.2 Conexión con el servidor

En este apartado se explicarán las configuraciones y bibliotecas (o dependencias) realizadas para permitir una conexión con el servidor siempre que exista conectividad a la red, de tal modo que los datos escritos en la tarjeta NFC se actualicen también en el servidor.

Cabe señalar que la implementación del servidor no es objeto de este TFG y ha sido suministrada por los tutores de este, por lo que no se describe en este documento el detalle de su implementación. Sin embargo, si es objeto del TFG y de la aplicación desarrollada la conexión con el servidor y el envío de información al mismo.

Inicialmente fue necesario incluir las dependencias y bibliotecas que se muestran en la Figura 3-13 y que se han explicado en el apartado 3.3.1.5.

```
implementation("com.squareup.picasso:picasso:2.71828")
implementation("com.squareup.retrofit2:retrofit:2.9.0")
implementation("com.squareup.retrofit2:converter-gson:2.9.0")
implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.6")
```

Figura 3-13. Dependencias añadidas para garantizar la conexión con el servidor [fuente propia]

En el archivo *Manifest* (ver Figura 3-8) se tienen que otorgar permisos a la aplicación para acceso a Internet. También se modificó la configuración de seguridad de red (ver Figura 3-14).

```
<uses-permission android:name="android.permission.INTERNET" />
```

Figura 3-14. Código añadido para conceder acceso a Internet a la aplicación [fuente propia]

El archivo XML que define la seguridad de red (ver Figura 3-15) está configurado para permitir tráfico sin cifrar hacia el servidor con la IP 193.146.212.177. Esto es debido a que el servidor al que nos conectaremos para remitir la información no soporta HTTPS, ya que es un servidor de pruebas. En un entorno de producción, el servidor soportará HTTPS y la comunicación vía cifrada.

```
<network-security-config>
    <domain-config cleartextTrafficPermitted = "true">

        <domain includeSubdomains = "true" > 193.146.212.177</domain>
    </domain-config>

</network-security-config>
```

Figura 3-15. Configuración de la seguridad de la red [fuente propia]

Además, fue necesario crear una interfaz *APIService* (ver Figura 3-16) encargada de declarar el método para crear un registro en el servidor remoto mediante una solicitud *HTTP POST*, en nuestro caso enviando los datos contenidos en un objeto de tipo *Variables*.

```
package NFMC

import ...

interface APIService {
    @POST("/")
    fun crearRegistro(@Body nfcregister: Variables): retrofit2.Call<Void>

}
```

Figura 3-16. Código relacionado con la interfaz API [fuente propia]

3.3.3 Implementación de las diferentes actividades

A lo largo de este apartado, se expondrán y explicarán detalladamente las diversas etapas que se seguido durante el proceso de creación de la aplicación en Android Studio. Cabe destacar que, dado que existen funciones que se repiten en las diferentes actividades, solo se explicarán la primera vez, aunque aparezcan de nuevo en líneas de código posteriores.

3.3.3.1 Inicio de sesión

La primera parte del desarrollo de la aplicación consistió en la creación de una pantalla para iniciar sesión, de modo que los usuarios ingresen su nombre de usuario y contraseña en campos de texto dedicados. Cuando el cliente hace clic en el botón de inicio de sesión, el sistema verifica si los datos ingresados coinciden con ciertas credenciales predefinidas. Si los datos son correctos (en este caso el usuario debe ser “71903419Z” y la contraseña “CUD”) pasa a la actividad principal. En caso contrario, se muestra un mensaje de "Acceso denegado". Este proceso de autenticación y respuesta se realiza mediante el uso de *listeners* de eventos y mensajes *toast* para notificar al usuario sobre el resultado de su intento de inicio de sesión, como se ve en la Figura 3-17.

```kotlin
class InicioSesion : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_inicio_sesion)

        val editTextDNI = findViewById<EditText>(R.id.editTextDNI)
        val editTextPassword = findViewById<EditText>(R.id.editTextPassword)
        val buttonLogin = findViewById<Button>(R.id.buttonLogin)

        buttonLogin.setOnClickListener { it: View!
            val nombre = editTextDNI.text.toString()
            val contraseña = editTextPassword.text.toString()

            if (nombre == "71903419Z" && contraseña == "CUD") {
                val intent = Intent( packageContext: this@InicioSesion, MenuPrincipal::class.java)
                startActivity(intent)
            } else {
                Toast.makeText( context: this@InicioSesion,  text: "Acceso denegado", Toast.LENGTH_SHORT).show()
            }
        }
    }
}
```

Figura 3-17. Código asociado a la actividad de *InicioSesion* [fuente propia]

3.3.3.2 Creación del menú principal

La segunda fase del desarrollo de la aplicación consistió en el diseño de un menú principal (ver Figura 3-5) que permita al usuario acceder a cada una de las actividades que conforman la aplicación como se muestra en la Figura 3-4. Para ello, se establecieron distintos botones que permiten la navegación entre las actividades de este proyecto Android gracias a una función específica que se activa al ser seleccionado dicho botón. Estas funciones, identificadas como "*Navegar_basico()*", "*Navegar_causa()*", entre otras, inician una nueva actividad de acuerdo con el botón seleccionado, tal y como se muestra en la Figura 3-18.

```kotlin
//Botones de movimiento entre aplicaciones
val boton_datos_basicos = findViewById<AppCompatButton>(R.id.boton_identificacion)
val boton_causa = findViewById<AppCompatButton>(R.id.boton_causa)
val boton_evaluacion = findViewById<AppCompatButton>(R.id.boton_evaluacion)
val boton_constantes = findViewById<AppCompatButton>(R.id.boton_constantes)
val boton_tratamiento = findViewById<AppCompatButton>(R.id.boton_tratamiento)
val boton_movimiento = findViewById<AppCompatButton>(R.id.boton_movimiento)

boton_datos_basicos.setOnClickListener { Navegar_basico() }
boton_causa.setOnClickListener { Navegar_causa() }
boton_evaluacion.setOnClickListener { Navegar_evaluacion() }
boton_constantes.setOnClickListener { Navegar_constantes() }
boton_tratamiento.setOnClickListener { Navegar_tratamiento() }
boton_movimiento.setOnClickListener { Navegar_movimiento() }
[...]
```

Figura 3-18. Parte del código de la actividad *MenuPrincipal* [fuente propia]

En el caso concreto de la interacción entre las actividades *MenuPrincipal* y *DatosBasicos* se sigue el esquema que vemos en la Figura 3-19. La comunicación entre las dos actividades se realiza en la función *Navegar_basico()* (ver Figura 3-20), que crea el *Intent* donde van los datos de entrada que se le envían a la actividad *DatosBasicos*, y que lanza con el método *launch()*. Cuando *DatosBasicos* retorne resultados, estos serán recogidos por la función *getResultDatosBasicos*.

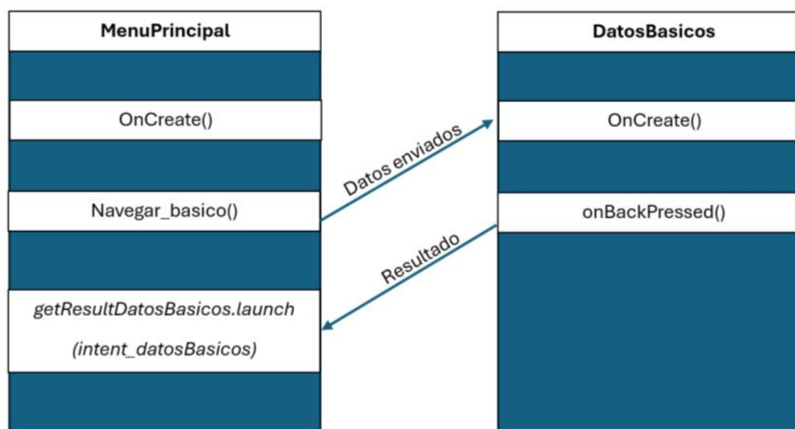


Figura 3-19. Esquema de cómo se envían y reciben datos entre las actividades y el menú principal [fuente propia]

```
fun Navegar_basico() {
    //pasamos datos al edit text de la pestaña de datos basicos
    val intent_datosBasicos = Intent( packageContext: this, DatosBasicos::class.java)
    intent_datosBasicos.putExtra( name: "nombre_vuelta", variables.nombre)
    intent_datosBasicos.putExtra( name: "apellidos_vuelta", variables.apellidos)
    intent_datosBasicos.putExtra( name: "sexo_vuelta", variables.sexo)
    intent_datosBasicos.putExtra( name: "Id_vuelta", variables.numeroId)
    intent_datosBasicos.putExtra( name: "nacimiento_vuelta", variables.fechaNacimiento)
    intent_datosBasicos.putExtra( name: "origen_vuelta", variables.lugarOrigen)
    intent_datosBasicos.putExtra( name: "fuerzasArmadas_vuelta", variables.fuerzasArmadasOrigen)
    getResultDatosBasicos.launch(intent_datosBasicos)
}
```

Figura 3-20. Función *Navegar_basico()* de la actividad *MenuPrincipal* [fuente propia]

La función *getResultDatosBasicos.launch(intent_datosBasicos)*, cuyo código vemos en la Figura 3-21, utiliza el método *registerForActivityResult()* para obtener resultados de inicio de actividad. Cuando se completa la actividad invocada y se devuelve un resultado, este bloque de código se ejecuta. Inicialmente verifica si la obtención fue exitosa y, si es así, actualiza las variables asociadas (nombre, apellidos, sexo, etc.); sin embargo, si alguno de estos datos no está presente, se asigna una cadena vacía como valor predeterminado.

```kotlin
private val getResultDatosBasicos =
    registerForActivityResult(
        ActivityResultContracts.StartActivityForResult()
    ) { result: ActivityResult ->
        if (result.resultCode == RESULT_OK) {
            val data = result.data
            variables.nombre = data?.getStringExtra( name: "nombre") ?: ""
            variables.apellidos = data?.getStringExtra( name: "apellidos") ?: ""
            variables.sexo = data?.getStringExtra( name: "sexo") ?: ""
            variables.numeroId = data?.getStringExtra( name: "numeroID") ?: ""
            variables.fechaNacimiento = data?.getStringExtra( name: "fechaNacimiento") ?: ""
            variables.lugarOrigen = data?.getStringExtra( name: "lugarOrigen") ?: ""
            variables.fuerzasArmadasOrigen = data?.getStringExtra( name: "fuerzasArmadasOrigen") ?: ""
        }
    }
```

Figura 3-21. Función *getResultDatosBasicos()* [fuente propia]

La comunicación entre la actividad MenuPrincipal y el resto de las actividades se realiza de forma análoga, exceptuando las actividades *Causa* y *ConstantesVitales*.

En la Figura 3-22 se presenta el código correspondiente a la función *Navegar_causa()*, la cual se encarga de enviar datos a la actividad *Causa*, así como de llevar al usuario a la actividad "*Causa*" en caso de que pulse sobre el botón correspondiente. En este contexto, los datos enviados consisten en listas de coordenadas y valores de contadores. Estas listas, que contienen coordenadas (pares de valores *Float* para las coordenadas X e Y), están asociadas a las posiciones de imágenes relacionadas con diversas lesiones, como fracturas, hemorragias, perforaciones, entre otras, presentes en la actividad denominada "*Causa*". Dichas listas son recibidas a través de la función *getResultCausa()*, cuya implementación parcial se muestra en la Figura 3-23. Además, *getResultCausa()* también se encarga de obtener y actualizar los contadores de imágenes asociados a cada lesión, los cuales registran la cantidad de imágenes relacionadas con dichas condiciones médicas.

```kotlin
fun Navegar_causa() {
    val intent_causa = Intent( packageContext: this, Causa::class.java)
    intent_causa.putExtra( name: "positionsList_fractura_vuelta", variables.positionsList_fractura)
    intent_causa.putExtra( name: "positionsList_hemorragia_vuelta", variables.positionsList_hemorragia)
    intent_causa.putExtra( name: "positionsList_perforacion_vuelta", variables.positionsList_perforacion)
    intent_causa.putExtra( name: "positionsList_no_perforacion_vuelta", variables.positionsList_no_perforacion)
    intent_causa.putExtra( name: "positionsList_quemadura_vuelta", variables.positionsList_quemadura)
    intent_causa.putExtra( name: "imageCounter_fractura_vuelta", this.imageCounter_fractura)
    intent_causa.putExtra( name: "imageCounter_hemorragia_vuelta", this.imageCounter_hemorragia)
    intent_causa.putExtra( name: "imageCounter_perforacion_vuelta", this.imageCounter_perforacion)
    intent_causa.putExtra( name: "imageCounter_no_perforacion_vuelta", this.imageCounter_no_perforacion)
    intent_causa.putExtra( name: "imageCounter_quemadura_vuelta", this.imageCounter_quemadura)
    getResultCausa.launch(intent_causa)
}
```

Figura 3-22. Función *Navegar_causa()* [fuente propia]

```
private val getResultCausa =
    registerForActivityResult(
        ActivityResultContracts.StartActivityForResult()
    ) { result: ActivityResult ->
        if (result.resultCode == RESULT_OK) {
            val data = result.data
            // Extraer los valores de solOriginal.x y solOriginal.y como Float
            variables.positionsList_fractura =
                (data?.getSerializableExtra( name: "positionsList_fractura") as? ArrayList<Pair<Float, Float>>)!!
            if (variables.positionsList_fractura != null) {
                // Recorrer la lista de coordenadas y mostrarlas
                for ((index, pair) in variables.positionsList_fractura.withIndex()) {...}
            }

            this.imageCounter_fractura = data?.getIntExtra( name: "imageCounter_fractura",  defaultValue: 0) ?: 0
            this.imageCounter_hemorragia = data?.getIntExtra( name: "imageCounter_hemorragia",  defaultValue: 0) ?: 0
            this.imageCounter_perforacion =
                data?.getIntExtra( name: "imageCounter_perforacion",  defaultValue: 0) ?: 0
            this.imageCounter_no_perforacion =
                data?.getIntExtra( name: "imageCounter_no_perforacion",  defaultValue: 0) ?: 0
            this.imageCounter_quemadura = data?.getIntExtra( name: "imageCounter_quemadura",  defaultValue: 0) ?: 0
        }
```

Figura 3-23. Parte del código asociado a la función *getResultCausa()* [fuente propia]

Por otro lado, en la Figura 3-24, se muestra parte del código asociado a la función *Navegar_constantes(),* que tiene una función similar a *Navegar_causa(),* solo que en este caso los datos que envía incluyen información sobre constantes vitales de un paciente: el pulso, la presión sanguínea, la tasa de respiración, etc. Además, si hay una lista de registros disponible, se adjunta al *Intent* para ser procesada por la actividad de destino. Estos valores son obtenidos de la actividad en cuestión gracias a la función *getResultConstantes.launch(intent_constantes)*, cuyo código se puede ver en la Figura 3-25. Esto garantiza que la información sobre las constantes vitales del paciente y sus registros médicos se maneje correctamente después de que el usuario interactúe con la actividad *ConstantesVitales*.

```
fun Navegar_constantes() {
    val intent_constantes = Intent( packageContext: this, ConstantesVitales::class.java)
    intent_constantes.putExtra( name: "hora_constantes_vuelta", variables.hora_constantes)
    intent_constantes.putExtra( name: "pulso_vuelta", variables.pulso)
    intent_constantes.putExtra( name: "presion_sanguinea_vuelta", variables.presion_sanguinea)
    intent_constantes.putExtra( name: "tasa_respiracion_vuelta", variables.tasa_respiracion)
    intent_constantes.putExtra( name: "saturacion_oxigeno_vuelta", variables.saturacion_oxigeno)
    intent_constantes.putExtra( name: "estado_herido_vuelta", variables.estado_herido)
    intent_constantes.putExtra( name: "escala_dolor_vuelta", variables.escala_dolor)

    // Verificar si la lista de registros no es nula y luego agregarla al intent
    if (variables.registros != null) {
        intent_constantes.putParcelableArrayListExtra(
            name: "registros_vuelta",
            ArrayList(variables.registros)
        )
    }
    getResultConstantes.launch(intent_constantes)
}
```

Figura 3-24. Código asociado a la función *Navegar_constantes()* [fuente propia]

```
private val getResultConstantes = registerForActivityResult(
    ActivityResultContracts.StartActivityForResult()
) { result: ActivityResult ->
    if (result.resultCode == RESULT_OK) {
        val data = result.data
        variables.hora_constantes = data?.getStringExtra( name: "hora_constantes") ?: ""
        variables.pulso = data?.getStringExtra( name: "pulso") ?: ""
        variables.presion_sanguinea = data?.getStringExtra( name: "presion_sanguinea") ?: ""
        variables.tasa_respiracion = data?.getStringExtra( name: "tasa_respiracion") ?: ""
        variables.saturacion_oxigeno = data?.getStringExtra( name: "saturacion_oxigeno") ?: ""
        variables.estado_herido = data?.getStringExtra( name: "estado_herido") ?: ""
        variables.escala_dolor = data?.getStringExtra( name: "escala_dolor") ?: ""


        val registrosArray = data?.getParcelableArrayExtra( name: "registros")

        // Convertir el array de registros en una lista mutable
        val registrosList = mutableListOf<Registro>()
        registrosArray?.forEach { registro ->
            if (registro is Registro) {
                registrosList.add(registro)
            }
        }
        // Asignar la lista mutable de registros
        variables.registros = registrosList
    }
}
```

Figura 3-25. Código asociado a la función *getResultConstantes()* [fuente propia]

MenuPrincipal también se encarga de la lectura y escritura de los datos recogidos en los tags NFC. Para la acción de lectura se emplea la función *leerNFC()*, la cual toma como entrada un objeto Tag que representa la etiqueta NFC. Primero, intenta obtener una instancia de *Ndef*, que proporciona los métodos para interactuar con una etiqueta NFC que contiene registros NDEF, a partir del objeto Tag y se conecta a la etiqueta NFC como vemos en la Figura 3-26.

```
private fun leerNFC(tag: Tag?): String {

    val ndef = Ndef.get(tag)
    ndef?.connect()
    val message = ndef?.ndefMessage
    val stringBuilder = StringBuilder()
    [...]
```

Figura 3-26. Parte del código de *leerNFC()* asociada a la interacción con la tarjeta NFC [fuente propia]

Posteriormente, lee el mensaje de la etiqueta y lo procesa recorriendo los registros del mensaje y verificando si son registros de texto bien conocidos. A continuación, extrae el texto de cada registro y lo agrega al *StringBuilder*, el cual se convierte en una cadena y luego se divide en mensajes separados como se refleja en la Figura 3-27.

```
    val mensajeCompleto = stringBuilder.toString()
    val mensajesSeparados = mensajeCompleto.split( ...delimiters: "\n")
    [...]
```

Figura 3-27. Parte del código de *leerNFC()* asociada a la construcción del *StringBuilder* [fuente propia]

Por otro lado, en la Figura 3-28 se muestra parte del código encargado de analizar y verificar si el mensaje de la etiqueta contiene registros de datos específicos y los asigna a las variables correspondientes. Además, la función limpia las listas existentes y crea nuevos registros con los datos extraídos de la etiqueta NFC. Finalmente, la función devuelve una cadena con el contenido de la etiqueta NFC o un mensaje de error si no se encuentra contenido en la etiqueta para notificar al usuario si ha sucedido algún problema durante el proceso.

```
if (mensajesSeparados.size >= 2) {
    //Datos Básicos
    variables.nombre = mensajesSeparados[0]
    variables.apellidos = mensajesSeparados[1]
    variables.sexo = mensajesSeparados[2]
    variables.numeroId = mensajesSeparados[3]
    variables.fechaNacimiento = mensajesSeparados[4]
    variables.lugarOrigen = mensajesSeparados[5]
    variables.fuerzasArmadasOrigen = mensajesSeparados[6]

    // Evaluación
    variables.fecha = mensajesSeparados[7]
    [...]
    // Si hay datos para fractura
    if (mensajesSeparados.size >= 54) {
        val coordenadasFractura = mensajesSeparados[53].split( ...delimiters: ";")
        for (coordenada in coordenadasFractura) {
            val partes = coordenada.split( ...delimiters: ",")
            if (partes.size == 2) {
                val x = partes[0].toFloat()
                val y = partes[1].toFloat()
                variables.positionsList_fractura.add(x to y)
            }
        }
    [...]
    ndef?.close()
    return if (stringBuilder.isNotEmpty()) stringBuilder.toString() else "No content found on the NFC tag."
}
```

Figura 3-28. Parte del código de *leerNFC()* asociada a los registros de datos [fuente propia]

En lo que a la escritura se refiere, primero se realiza un proceso de lectura como el que acabamos de describir para comprobar que el tag donde queremos escribir tenga el mismo ID de tal modo que se evite sobrescribir de manera accidental los datos del tag de otro paciente. Para ello empleamos el código reflejado en la Figura 3-29.

Si el número ID es efectivamente el mismo, el proceso de escritura se realiza y, además, se muestra un mensaje al usuario indicando que el proceso se ha realizado con éxito. Sin embargo, en caso de que sean ID diferentes, se ejecuta la función *mostrarDialogoConfirmacion(ndef, message)*, que vemos más en detalle en la Figura 3-30.

```
val mensajeCompleto = stringBuilder.toString()
val mensajesSeparados = mensajeCompleto.split( ...delimiters: "\n")
if (mensajesSeparados.size >= 4) { // Suponiendo que el numeroId está en la cuarta posición
    val numeroIdTag = mensajesSeparados[3] // Obtener el numeroId del tag NFC

    if (numeroIdTag == numeroIdDeseado) { // Comparar con el numeroId deseado
        val ndefRecord = NdefRecord.createTextRecord( languageCode: null, message)
        val newNdefMessage = NdefMessage(arrayOf(ndefRecord))
        ndef?.writeNdefMessage(newNdefMessage)
        mostrarMensaje( mensaje: "Message written on the NFC tag.")
    } else {
        mostrarDialogoConfirmacion(ndef, message)
    }
} else {
    mostrarMensaje( mensaje: "Failed to extract the number ID from the NFC tag message.")
}
```

Figura 3-29. Parte del código de *escribirNFC()* encargado de comprobar los ID [fuente propia]

El código *mostrarDialogoConfirmacion()* crea un diálogo de confirmación utilizando un *AlertDialog*. Este diálogo presenta un mensaje de advertencia al usuario indicando que el número de identificación no coincide, como vemos en la Figura 3-30. El usuario tiene la opción de confirmar o cancelar la operación.

```
private fun mostrarDialogoConfirmacion(ndef: Ndef?, message: String) {
    val builder = AlertDialog.Builder( context: this)
    builder.setTitle("Confirmation")
    builder.setMessage("The number ID does not match. Are you sure you want to write the data anyway?")
    [...]
```

Figura 3-30. Parte del código de la función *mostrarDialogoConfirmacion()* [fuente propia]

Si el usuario elige confirmar ("*Yes*"), se intenta conectar a la etiqueta NFC y escribir un nuevo mensaje NDEF en la etiqueta con el texto proporcionado en el parámetro *message* (ver Figura 3-31). Si la escritura es exitosa, se muestra un mensaje indicando que se ha escrito el mensaje en la etiqueta NFC mientras que, si ocurre un error durante la escritura, se muestra un mensaje de error.

```
builder.setPositiveButton( text: "Yes (Keep the TAG close to the device)") { dialog, which ->
    try {
        ndef?.connect() // Conectar antes de escribir el mensaje
        val ndefRecord = NdefRecord.createTextRecord( languageCode: null, message)
        val newNdefMessage = NdefMessage(arrayOf(ndefRecord))
        ndef?.writeNdefMessage(newNdefMessage)
        mostrarMensaje( mensaje: "Message written on the NFC tag.")
    } catch (e: IOException) {
        mostrarMensaje( mensaje: "Error writing to the NFC tag: ${e.message}")
    } finally {
        ndef?.close()
    }
}
[...]
```

Figura 3-31. Parte del código de la función *mostrarDialogoConfirmacion()* encargada de la opción afirmativa [fuente propia]

Si el usuario elige cancelar ("*No*"), se muestra un mensaje indicando que se debe verificar la información como se ve en la Figura 3-32.

```
    builder.setNegativeButton( text: "No") { dialog, which ->
        mostrarMensaje( mensaje: "Verify data")
    }

    val dialog: AlertDialog = builder.create()
    dialog.show()
}
```

Figura 3-32. Parte del código de la función *mostrarDialogoConfirmacion()* encargada de la opción negativa [fuente propia]

Es importante destacar que, durante el proceso de escritura en la tarjeta NFC, también se ejecuta *enviarServidor*() (ver Figura 3-33), encargada de enviar los mismos datos que se copian en la tarjeta NFC a un servidor utilizando *Retrofit*, una biblioteca de cliente HTTP para Android. Primero, se configura *Retrofit* con la URL del servidor y se crea una instancia de la interfaz de servicio API utilizando la interfaz definida anteriormente. Luego, se realiza una llamada asíncrona (de manera que se permite que la aplicación siga siendo receptiva y usable mientras se realizan estas operaciones en segundo plano) para enviar los datos al servidor utilizando el método *crearRegistro()*.

```
private fun enviarServidor() {
    //PARTE DEL SERVIDOR
    val retrofit = this.getRetrofit()
    val servicioAPI = retrofit.create(APIService::class.java)
    //Toast.makeText(this@MenuPrincipal,this.variables.IDsocorrista, Toast.LENGTH_SHORT).show()
    //return
    servicioAPI.crearRegistro(this.variables).enqueue(object : Callback<Void> {
        [...]
```

Figura 3-33. Parte del código asociado a la función *enviarServidor()* [fuente propia]

Dependiendo de la respuesta del servidor, se muestra un mensaje indicando si la conexión fue exitosa o si hubo algún error, como se ve en el código de la Figura 3-34. Si la conexión es exitosa (*response.isSuccessful*), se muestra un mensaje indicando que se ha establecido una conexión exitosa con el servidor. En caso de error, se muestra un mensaje con información sobre el error, ya sea un problema de red u otro tipo de error.

```
override fun onResponse(call: Call<Void>, response: Response<Void>) {

    if (response.isSuccessful) {
        // La llamada fue exitosa, se ha creado el usuario
        Toast.makeText(
            context: this@MenuPrincipal,
            text: "Successful connection to the server.",
            Toast.LENGTH_SHORT
        )
            .show()
    } else {
        // La llamada falló por algún motivo
        Toast.makeText( context: this@MenuPrincipal, response.toString(), Toast.LENGTH_SHORT)
            .show()
    }
}

override fun onFailure(call: Call<Void>, t: Throwable) {
    // La llamada falló debido a un error de red u otro tipo de error
    Toast.makeText( context: this@MenuPrincipal, t.toString(), Toast.LENGTH_SHORT).show()
}
```

Figura 3-34. Código de *enviarServidor()* encargado de mostrar una respuesta en función de si la conexión fue exitosa o no [fuente propia]

Por otro lado, para poder escribir un mensaje en un tag NFC, es necesario primero construir dicho mensaje para que pueda ser almacenado en una etiqueta NFC. Este proceso se realiza a través de la función *construirMensajeNFC()*, la cual comienza incluyendo los datos básicos (ver Figura 3-35) y luego agrega las coordenadas asociadas a los distintos tipos de lesiones (fractura, perforación, hemorragia, etc.) como se muestra en la Figura 3-36. Posteriormente, agrega datos de registro de las constantes vitales si están disponibles. En este caso, los campos se separan por puntos y coma.

```
private fun construirMensajeNFC(): String {
    val stringBuilder = StringBuilder()

    // Incluir datos básicos
    stringBuilder.append(
        "${variables.nombre}\n${variables.apellidos}\n${variables.sexo}\n${variables.numeroId}\n${variables.fechaNacimiento}" +
                "\n${variables.lugarOrigen}\n${variables.fuerzasArmadasOrigen}\n" +
                "${variables.fecha}\n${variables.hora}\n${variables.fechaEvaluacion}\n${variables.horaEvaluacion}" +
                "\n${variables.senales}\n${variables.alergias}\n${variables.sangrado}\n${variables.airway}\n${variables.respiracion}" +
                "\n${variables.circulacion}\n${variables.head}\n" +
                "${variables.hora_constantes}\n${variables.pulso}\n${variables.presion_sanguinea}\n${variables.tasa_respiracion}" +
                "\n${variables.saturacion_oxigeno}\n${variables.escala_dolor}\n${variables.estado_herido}\n" +
                "${variables.nombre_t}\n${variables.volumen}\n${variables.via}\n${variables.hora_t}\n${variables.nombre_productos_sanguineos}
                "\n${variables.volumen_productos_sanguineos}\n" +
                "${variables.via_productos_sanguineos}\n${variables.hora_productos_sanguineos}\n${variables.nombreAnalgesia}" +
                "\n${variables.dosisAnalgesia}\n" +
                "${variables.viaAnalgesia}\n${variables.horaAnalgesia}\n${variables.nombreAntibioticos}" +
                "\n${variables.dosisAntibioticos}\n${variables.viaAntibioticos}\n" +
                "${variables.horaAntibioticos}\n${variables.nombreOtros}\n${variables.dosisOtros}\n${variables.viaOtros}" +
                "\n${variables.horaOtros}\n${variables.ubicacionTorniquete}\n" +
                "${variables.horaTorniquete}\n${variables.hipotermia}\n" +
                "${variables.categoria}\n${variables.nombreSocorrista}\n${variables.apellidosSocorrista}" +
                "\n${variables.IDsocorrista}\n${variables.datosAdicionales}"
    )
    [...]
```

Figura 3-35. Datos básicos añadidos a través de la función *construirMensajeNFC()* [fuente propia]

```
    stringBuilder.append("\n") // Agrega un salto de línea antes de comenzar las coordenadas de fractura
    // Agregar las coordenadas de positionsList_fractura
    for ((index, pair) in variables.positionsList_fractura.withIndex()) {
        val (x, y) = pair
        stringBuilder.append("$x,$y") // Agrega las coordenadas x y y separadas por una coma
        if (index < variables.positionsList_fractura.size - 1) {
            stringBuilder.append(";") // Agrega un punto y coma si no es el último par de coordenadas
        }
    }
    [...]
```

Figura 3-36. Coordenadas de la imagen fractura a través de la función *construirMensajeNFC()* [fuente propia]

3.3.3.3 DatosBasicos

Esta actividad recupera los datos básicos del usuario que fueron pasados desde la actividad anterior (ver esquema de la Figura 3-4) y asociarlos a los campos de texto correspondientes en la interfaz gráfica de la aplicación (ver Figura 3-37). Cada campo de texto representa un dato básico, como nombre, apellidos, sexo, etc. De esta manera, cuando el usuario llega a esta pantalla, puede ver los datos que ya ha ingresado anteriormente y, si es necesario, puede modificarlos.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_datos_basicos)

    val bundle = intent.extras
    // nombre
    val nombre = bundle?.getString( key: "nombre_vuelta")
    nombreText = findViewById(R.id.escribir_nombre)
    nombreText.setText(nombre)

    // apellidos
    val apellidos = bundle?.getString( key: "apellidos_vuelta")
    apellidosText = findViewById(R.id.escribir_apellidos)
    apellidosText.setText(apellidos)
    [...]
```

Figura 3-37. Parte del código del método *onCreate* de la actividad *DatosBasicos* [fuente propia]

Los datos que son modificados se devuelven a la actividad llamante (*MenuPrincipal*) cuando se invoca la función *onBackPressed()*. Esta se encarga de manejar el evento de retroceso (cuando se presiona el botón de "volver atrás"). Inicialmente, tal y como se muestra en la Figura 3-38, se recupera el texto ingresado en cada uno de los campos de datos básicos (como nombre, apellidos, sexo, etc.). Luego realiza varias comprobaciones para determinar si se puede permitir que el usuario retroceda a la pantalla anterior.

```
override fun onBackPressed() {
    val nombre = nombreText.text.toString().trim()
    val apellidos = apellidosText.text.toString().trim()
    val sexo = sexoText.text.toString().trim()
    val numeroID = numeroIDText.text.toString().trim()
    val fechaNacimiento = fechaNacimientoText.text.toString().trim()
    val lugarOrigen = lugarOrigenText.text.toString().trim()
    val fuerzasArmadasOrigen = fuerzasArmadasOrigenText.text.toString().trim()
    [...]
```

Figura 3-38. Parte del código de la función *onBackPressed()* [fuente propia]

La primera comprobación, representada en el código de la Figura 3-39, verifica si todos los campos están en blanco. Si es así, significa que el usuario no ha ingresado ningún dato nuevo y se permite volver atrás sin mostrar ningún mensaje de advertencia.

```
// Verificar si los campos están en blanco
if (nombre.isBlank() && apellidos.isBlank() && sexo.isBlank() &&
    numeroID.isBlank() && fechaNacimiento.isBlank() &&
    lugarOrigen.isBlank() && fuerzasArmadasOrigen.isBlank()
) {
    super.onBackPressed() // Si todos los campos están en blanco, se permite volver atrás
    return
}
```

Figura 3-39. Primera comprobación realizada por la función *onBackPressed()* [fuente propia]

Luego, se verifica si los campos han sido modificados desde la última vez que se cargaron los datos a través de la función *camposModificados(),* la cual se explicará más adelante. Si no se han modificado, también se permite volver atrás sin mostrar mensajes adicionales (ver Figura 3-40).

```
if (!camposModificados()) {
    super.onBackPressed()
    return
}
[...]
```

Figura 3-40. Segunda comprobación realizada por la función *onBackPressed()* [fuente propia]

Además, si se han ingresado datos en el campo de fecha de nacimiento, se realiza una validación adicional para asegurarse de que la fecha sea válida (ver Figura 3-41) y esté en el formato correcto. Si la fecha no es válida o es posterior a la fecha actual (ver Figura 3-42), se muestra el mensaje de error correspondiente.

```
// Verificar si el campo de fecha de nacimiento no está en blanco antes de validar la fecha
if (fechaNacimiento.isNotBlank()) {
    if (!esFechaValida(fechaNacimiento)) {
        fechaNacimientoText.error = "The date of birth provided is not valid"
        return
    }
    [...]
```

Figura 3-41. Validación de la fecha en la función *onBackPressed()* [fuente propia]

```
val sdf = SimpleDateFormat( pattern: "dd/MM/yyyy", Locale.getDefault())
val fechaNacimientoDate: java.util.Date? = sdf.parse(fechaNacimiento)
val fechaActual = Calendar.getInstance().time

if (fechaNacimientoDate != null) {
    if (fechaNacimientoDate.after(fechaActual)) {
        fechaNacimientoText.error =
            "The date of birth entered must be before or equal to the current date"
        return
    }
}
[...]
```

Figura 3-42. Mensaje de error mostrado [fuente propia]

Como ya se mencionó anteriormente, para comprobar si alguno de los campos de los datos básicos del paciente ha sido modificado se emplea la función *camposModificados*(). De esta manera se evita mostrar al usuario una notificación cuando no se ha realizado ningún cambio en la actividad. Primero, obtiene el texto actualmente ingresado en cada uno de los campos de datos básicos (ver Figura 3-43) y lo compara con los valores originales que se obtuvieron del intento de la actividad anterior (ver Figura 3-44).

```
private fun camposModificados(): Boolean {
    val nombre = nombreText.text.toString().trim()
    val apellidos = apellidosText.text.toString().trim()
    val sexo = sexoText.text.toString().trim()
    val numeroID = numeroIDText.text.toString().trim()
    val fechaNacimiento = fechaNacimientoText.text.toString().trim()
    val lugarOrigen = lugarOrigenText.text.toString().trim()
    val fuerzasArmadasOrigen = fuerzasArmadasOrigenText.text.toString().trim()
    [...]
```

Figura 3-43. Obtención del texto ingresado en los campos de los datos básicos [fuente propia]

```
val bundle = intent.extras

val nombreOriginal = bundle?.getString( key: "nombre_vuelta").toString().trim()
val apellidosOriginal = bundle?.getString( key: "apellidos_vuelta").toString().trim()
val sexoOriginal = bundle?.getString( key: "sexo_vuelta").toString().trim()
val numeroIDOriginal = bundle?.getString( key: "Id_vuelta").toString().trim()
val fechaNacimientoOriginal = bundle?.getString( key: "nacimiento_vuelta").toString().trim()
val lugarOrigenOriginal = bundle?.getString( key: "origen_vuelta").toString().trim()
val fuerzasArmadasOrigenOriginal =
    bundle?.getString( key: "fuerzasArmadas_vuelta").toString().trim()
[...]
```

Figura 3-44. Valores originales que se obtuvieron del intento de la actividad anterior [fuente propia]

Luego, compara cada campo individualmente para determinar si hay alguna diferencia entre los valores actuales y los valores originales (ver Figura 3-45). Si encuentra al menos una diferencia en algún campo, devuelve *true*, lo que indica que los campos han sido modificados. En caso contrario, si no se encuentra ninguna diferencia, devuelve *false*, indicando que no ha habido modificaciones en los campos.

```
    [...]
    return (nombre != nombreOriginal || apellidos != apellidosOriginal || sexo != sexoOriginal ||
            numeroID != numeroIDOriginal || fechaNacimiento != fechaNacimientoOriginal ||
            lugarOrigen != lugarOrigenOriginal || fuerzasArmadasOrigen != fuerzasArmadasOrigenOriginal)
}
```

Figura 3-45. Comparación de los datos originales y los datos obtenidos [fuente propia]

Por otro lado, cabe destacar también el uso de *esFechaValida()* dentro de la función *onBackPressed()* para la comprobación de la fecha. Este código primero divide la fecha en partes para obtener el día, el mes y el año. Luego, verifica si el año está dentro de un rango válido (entre 1925 y 9999), si el mes está entre 1 y 12 y si el día es mayor que 1 (ver Figura 3-46). Si alguna de estas condiciones no se cumple, la función devuelve *false*, indicando que la fecha no es válida.

```
private fun esFechaValida(fecha: String): Boolean {
    val partesFecha = fecha.split( ...delimiters: "/")
    val dia = partesFecha[0].toInt()
    val mes = partesFecha[1].toInt()
    val año = partesFecha[2].toInt()

    // Validar el rango del año, el mes y el día
    if (año < 1925 || año > 9999 || mes < 1 || mes > 12 || dia < 1) {
        return false
    }
    [...]
```

Figura 3-46. Verificación de que el día, mes y año están en un rango válido [fuente propia]

Luego, determina la cantidad de días que tiene el mes específico teniendo en cuenta si el año es bisiesto o no (ver Figura 3-47). Si el día ingresado es mayor que la cantidad de días en el mes correspondiente, la función devuelve *false*, indicando que la fecha no es válida. En caso contrario, si todas las condiciones se cumplen, la función devuelve *true*, indicando que la fecha es válida.

```
    [...]
    val diasPorMes = when (mes) {
        1, 3, 5, 7, 8, 10, 12 -> 31
        4, 6, 9, 11 -> 30
        2 -> if (esAnoBisiesto(año)) 29 else 28
        else -> return false // Mes inválido
    }

    return dia <= diasPorMes
}
```

Figura 3-47. Determinación de la cantidad de días que tiene el mes [fuente propia]

Cabe resaltar que, para comprobar si el año es bisiesto o no, se emplea el código reflejado en la Figura 3-48, el cual establece que un año es bisiesto si es divisible por 4 pero no divisible por 100, a menos que también sea divisible por 400. Entonces, si el año es divisible por 4 y (no es divisible por 100 o es divisible por 400), la función devuelve *true*, indicando que el año es bisiesto. En caso contrario, devuelve *false*, indicando que el año no es bisiesto.

```
private fun esAnoBisiesto(año: Int): Boolean {
    return año % 4 == 0 && (año % 100 != 0 || año % 400 == 0)
}
```

Figura 3-48. Comprobación de si el año es bisiesto [fuente propia]

3.3.3.4 Causa

Durante esta actividad se seguirá un flujo de trabajo como el que se representa en el esquema de la Figura 3-49.

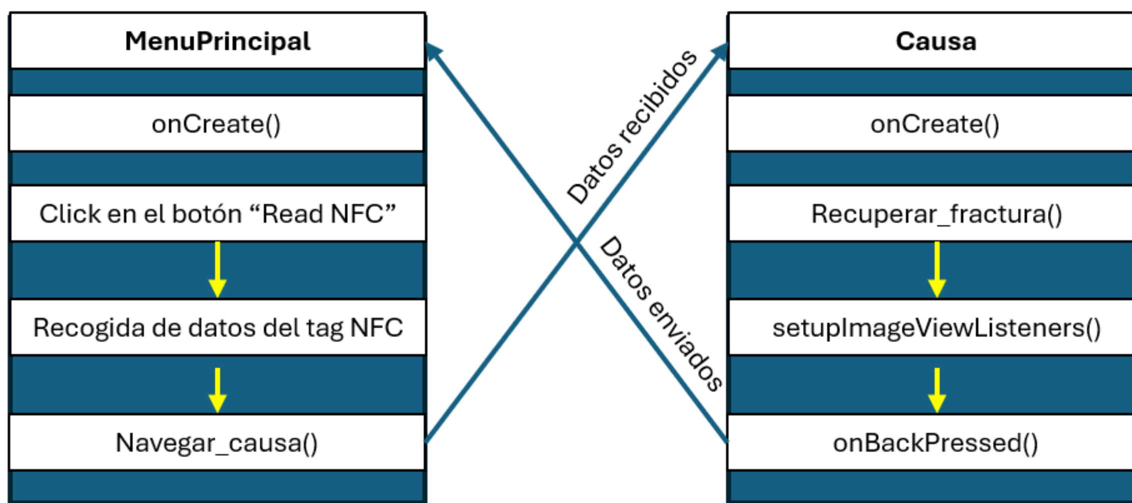


Figura 3-49. Esquema de flujo de trabajo seguido en la actividad *Causa* [fuente propia]

Inicialmente se recuperan los datos de las posiciones de las imágenes que existían en *MenuPrincipal* y la visibilidad de las imágenes a través de la función *recuperar_fractura()*, *recuperar_hemorragia()*, etc. tal y como se muestra en la Figura 3-50.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_causa)

    // Inicializar la lista de posiciones
    recuperar_fractura()
    recuperar_hemorragia()
    recuperar_perforacion()
    recuperar_no_perforacion()
    recuperar_quemadura()

    // Configurar el onTouchListener para cada ImageView
    setupImageViewListeners()
}
```

Figura 3-50. Método *onCreate* de la actividad *Causa* [fuente propia]

Explicaremos únicamente la función *recuperar_fractura()* ya que el resto emplea el mismo código, pero con imágenes diferentes. Primero, como se ve en la Figura 3-51, localiza la *ImageView* asociada a

la imagen de fractura mediante su identificador único en el diseño de la actividad. Luego, guarda las coordenadas iniciales de esta imagen original.

```
private fun recuperar_fractura() {

    val simbolo_fractura = findViewById<ImageView>(R.id.simbolo_fractura)

    // Guardar las coordenadas iniciales de la imagen original
    initialXFractura = simbolo_fractura.x
    initialYFractura = simbolo_fractura.y
    [...]
```

Figura 3-51. Obtención de las coordenadas de la imagen en la función *recuperar_fractura()* [fuente propia]

A continuación, como se ve en la Figura 3-52, la función recupera el valor de un contador llamado *imageCounter_fractura*, que se utiliza para mantener un seguimiento del número de imágenes de fractura presentes en la pantalla y, de esta manera, poder eliminar las imágenes de manera eficiente como se explicará más adelante. Además, obtiene un array de coordenadas de las imágenes de fractura enviadas desde otra actividad.

```
//Recibir valor imageCounter
this.imageCounter_fractura = intent.getIntExtra( name: "imageCounter_fractura_vuelta",  defaultValue: 0)

// Recibir el array de posiciones enviado por la actividad NFC
positionsList_fractura =
    intent.getSerializableExtra( name: "positionsList_fractura_vuelta") as? MutableList<Pair<Float, Float>>
        ?: mutableListOf()

[...]
```

Figura 3-52. Recuperación del contador y obtención del array en la función *recuperar_fractura()* [fuente propia]

Después de recuperar estos datos, la función itera sobre el array de coordenadas de las imágenes de fractura como se ve en el código de la Figura 3-53. Para cada par de coordenadas, crea una nueva *ImageView* que representa la imagen de fractura. Configura su imagen de origen, tamaño y color. Luego, establece su posición en la pantalla según las coordenadas proporcionadas.

```
// Mostrar las imágenes y sus copias
positionsList_fractura.forEachIndexed { index, pair ->
    val (x, y) = pair
    val newImage = ImageView( context: this)// Obtener la imagen original

    // Establecer la imagen de origen y las dimensiones
    newImage.setImageResource(R.drawable.simbolo_fractura_bueno)
    newImage.layoutParams = ConstraintLayout.LayoutParams(80.dpToPx(), 80.dpToPx())

    // Establecer el color de la imagen
    newImage.setColorFilter(
        ContextCompat.getColor( context: this, R.color.amarillo),
        PorterDuff.Mode.SRC_ATOP
    )

    // Establecer la posición de la nueva imagen
    newImage.x = x
    newImage.y = y

    // Agregar la nueva imagen al diseño
    val layout = findViewById<ConstraintLayout>(R.id.causa)
    layout.addView(newImage)
```

Figura 3-53. Iteración sobre el array de coordenadas [fuente propia]

La función también crea un botón asociado a cada imagen de fractura para permitir su eliminación, cuya lógica se muestra en la Figura 3-54. Este botón se posiciona, en este caso, en la esquina inferior derecha de la imagen y se configura para que, al hacer clic en él, elimine la imagen correspondiente de la pantalla junto con el botón. El diseño, al superponer los botones del mismo tipo de imágenes, proporciona la sensación al usuario de existir solo un botón responsable de eliminar las imágenes. Con este mecanismo se consigue ir eliminando las imágenes creadas más recientemente hasta llegar a la primera.

```
    // Configurar el OnClickListener para el botón de eliminar
    deleteButton.setOnClickListener { it: View!
        val idx = deleteButtonMap_fractura[it] ?: -1
        if (idx != -1) {
            layout.removeView(newImage)
            positionsList_fractura.removeAt(idx)
            layout.removeView(deleteButton)
            deleteButtonMap_fractura.remove(deleteButton)
            if (imageCounter_fractura > 0) {
                imageCounter_fractura--
            }
        }
    }
    [...]
```

Figura 3-54. Lógica asociada al botón de eliminación [fuente propia]

Por otro lado, como habíamos mencionado anteriormente, dentro de *onCreate()* también se hace una llamada a la función *setupImageViewListeners()* (ver Figura 3-55). Esta se encarga de asignar a cada

imagen (representada por un *ImageView*) un "*listener*" para que pueda responder cuando el usuario toca o interactúa con ella.

```
private fun setupImageViewListeners() {
    val fracturaImageView = findViewById<ImageView>(R.id.simbolo_fractura)
    val hemorragiaImageView = findViewById<ImageView>(R.id.simbolo_hemorragia)
    val perforacionImageView = findViewById<ImageView>(R.id.simbolo_herida_perforacion)
    val sinPerforacionImageView = findViewById<ImageView>(R.id.simbolo_herida_no_perforacion)
    val quemadoImageView = findViewById<ImageView>(R.id.simbolo_quemado)

    initialXFractura = fracturaImageView.x
    initialYFractura = fracturaImageView.y
    initialXHemorragia = hemorragiaImageView.x
    initialYHemorragia = hemorragiaImageView.y
    initialXPerforacion = perforacionImageView.x
    initialYPerforacion = perforacionImageView.y
    initialXsinPerforacion = sinPerforacionImageView.x
    initialYsinPerforacion = sinPerforacionImageView.x
    initialXquemado = quemadoImageView.x
    initialYquemado = quemadoImageView.y


    fracturaImageView.setOnTouchListener { _, event ->
        handleTouchEvent(fracturaImageView, event)
    }
    [...]
```

Figura 3-55. Parte del código asociado a la función *setupImageViewListeners()* [fuente propia]

Dentro de *setupImageViewListeners()* se hace uso de la función *handleTouchEvent()* encargada de gestionar las interacciones táctiles con una imagen en la interfaz de usuario. Podemos dividir la función en tres partes fundamentales, la primera, *MotionEvent.ACTION_DOWN* (ver Figura 3-56), tiene lugar cuando el usuario toca la imagen y la función asociada consiste en guardar las coordenadas iniciales del evento táctil.

```
private fun handleTouchEvent(imageView: ImageView, event: MotionEvent): Boolean {
    when (event.action) {
        MotionEvent.ACTION_DOWN -> {
            // Guarda las coordenadas iniciales del evento
            initialX = event.rawX - imageView.x
            initialY = event.rawY - imageView.y
        }
        [...]
```

Figura 3-56. Parte responsable del "*Action Down*" del *handleTouchEvent* [fuente propia]

La segunda es *MotionEvent.ACTION_MOVE* (ver Figura 3-57), que tiene lugar si el usuario mueve el dedo sobre la pantalla mientras mantiene presionada la imagen y cuya función es actualizar la posición de la imagen en función del movimiento del dedo.

```
MotionEvent.ACTION_MOVE -> {
    imageView.x = event.rawX - initialX
    imageView.y = event.rawY - initialY
}
[...]
```

Figura 3-57. Parte responsable del "*Action Move*" del *handleTouchEvent* [fuente propia]

Por último, está *MotionEvent.ACTION_UP* (ver Figura 3-58), que se desencadena cuando el usuario levanta el dedo de la pantalla después de tocar y mover la imagen. Esta parte es la responsable de crear una copia de la imagen en su nueva posición y de restaurar la posición original de la imagen que se movió.

```
        [...]
        MotionEvent.ACTION_UP -> {
            // Siempre que se levante el dedo, crea una copia de la imagen movida
            createNewImage1(imageView.x,  y: imageView.y + 248, this.imageCounter_fractura)
            // Restaurar la posición original de la imagen original
            imageView.x = initialXFractura + 44
            imageView.y = initialYFractura + 88
            this.imageCounter_fractura++


        }
    }
    return true
}
```

Figura 3-58. Parte responsable del "*Action Up*" del *handleTouchEvent* [fuente propia]

Como acabamos de ver, al levantar el usuario el dedo de la pantalla se ejecuta *createNewImage()*, cuyo código se muestra en la Figura 3-59 y que inicialmente obtiene la imagen original que servirá como base para la nueva imagen a crear. Luego, configura una nueva instancia de *ImageView*, asignándole la misma imagen que la original y ajustando sus dimensiones para que coincidan.

```
private fun createNewImage1(x: Float, y: Float, index: Int) {
    val originalImage = findViewById<ImageView>(R.id.simbolo_fractura)
    val newImage = ImageView( context: this)// Obtener la imagen original

    // Establecer la imagen de origen y las dimensiones
    newImage.setImageResource(R.drawable.simbolo_fractura_bueno)
    newImage.layoutParams =
        ConstraintLayout.LayoutParams(originalImage.width, originalImage.height)
    [...]
```

Figura 3-59. Parte inicial del código asociada a la función *createNewImage1()* [fuente propia]

Posteriormente, aplica un filtro de color a la nueva imagen para modificar su apariencia visual y establece sus coordenadas x e y para posicionarla en la pantalla (ver Figura 3-60)

```
// Establecer el color de la imagen
newImage.setColorFilter(
    ContextCompat.getColor( context: this, R.color.amarillo),
    PorterDuff.Mode.SRC_ATOP
)
[...]
```

Figura 3-60. Código asociado al establecimiento de color de la imagen [fuente propia]

Una vez configurada la nueva imagen, se le asigna un identificador único y una etiqueta para su identificación y gestión posterior como se muestra en la Figura 3-61. Luego se agrega al diseño de la interfaz de usuario para que sea visible para el usuario. Además, se guarda la posición de la nueva imagen en una lista para llevar un registro de su ubicación en pantalla.

```
// Asignar un ID y una etiqueta a la nueva imagen si es necesario
newImage.id = View.generateViewId()
newImage.tag = "Image $index"

// Agregar la nueva imagen al diseño
val layout = findViewById<ConstraintLayout>(R.id.causa)
layout.addView(newImage)
[...]
```

Figura 3-61. Asignación de un identificador y una etiqueta [fuente propia]

Por último, se ejecutaría la función *onBackPressed()*, cuyo funcionamiento es similar al que ya se explicó en *MenuPrincipal,* y que asegura que los datos relevantes se pasen correctamente de la actividad actual a la actividad principal (*MenuPrincipal*) antes de cerrarla.

3.3.3.5 Evaluación y Tratamiento

Durante esta actividad se seguirá un flujo de trabajo como el que se representa en el esquema de la Figura 3-62.

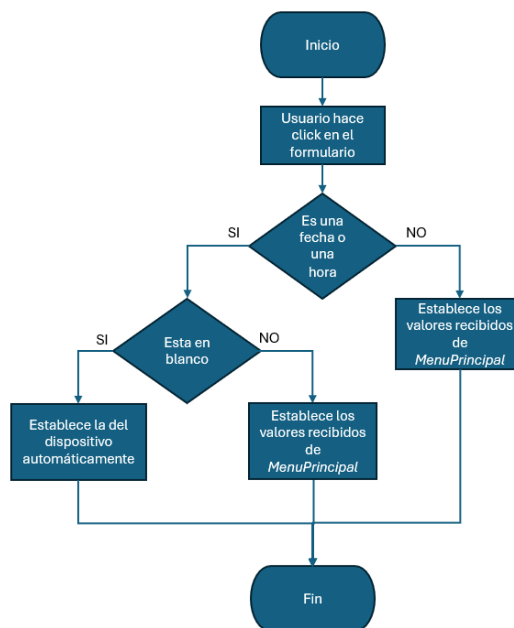


Figura 3-62. Esquema representativo del flujo de trabajo de la actividad *Evaluación* y *Tratamiento* [fuente propia]

Inicialmente se ejecuta el método *onCreate*, que recupera los datos pasados desde la actividad anterior. Estos datos incluyen valores como la fecha, hora, señales vitales, alergias, etc. como se muestra en la Figura 3-63.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_tratamiento)
    [...]
    val nombre = bundle?.getString( key: "nombre_vuelta")
    val volumen = bundle?.getString( key: "volumen_vuelta")
    val via = bundle?.getString( key: "via_vuelta")
    val hora = bundle?.getString( key: "hora_vuelta")
    [...]
```

Figura 3-63. Parte del código encargado de recuperar los valores de diferentes variables [fuente propia]

Cabe destacar que, en el caso de la fecha (ver Figura 3-64) y la hora (ver Figura 3-65), se ha implementado un código que establece automáticamente los valores que tiene el dispositivo en ese momento si no se han leído datos de una tarjeta NFC previamente.

```
private fun obtenerHoraActual(): String {
    val formato = SimpleDateFormat( pattern: "HH:mm", Locale.getDefault())
    val calendario = Calendar.getInstance()
    return formato.format(calendario.time)
}
```

Figura 3-64. Código de la función *obtenerHoraActual()* [fuente propia]

```
private fun obtenerFechaActual(): String {
    val calendar = Calendar.getInstance()
    val dateFormat = SimpleDateFormat( pattern: "dd/MM/yyyy", Locale.getDefault())
    return dateFormat.format(calendar.time)
}
```

Figura 3-65. Código de la función *obtenerFechaActual()* [fuente propia]

Una vez recuperados los datos, se asignan a los elementos correspondientes de la interfaz de usuario utilizando los métodos *setText* de los *EditText* como se ve en la Figura 3-66. Esto asegura que los datos recibidos se muestren correctamente en la pantalla para que el usuario los pueda ver y modificar si es necesario.

```
[...]
// Asignar los valores a los EditText correspondientes
findViewById<EditText>(R.id.escribir_fecha).setText(fechaMostrar)
findViewById<EditText>(R.id.escribir_hora).setText(horaMostar)
findViewById<EditText>(R.id.escribir_fecha_evaluacion).setText(fecha_evaluacion_Mostrar)
findViewById<EditText>(R.id.escribir_hora_evaluacion).setText(horaEvaluacionMostrar)
[...]
```

Figura 3-66. Establecimiento de los valores recuperados en los *EditText* correspondientes [fuente propia]

Por otro lado, todos los formularios permanecen inicialmente ocultos hasta que el usuario pulsa en ellos de cara a mantener una presentación más limpia de la aplicación. Esto se consigue gracias al código que se muestra en la Figura 3-67.

```
fun mostrarFormularioFechaHora(view: View) {
    toggleVisibility(formularioFechaHoraLesion)
}
```

Figura 3-67. Código asociado a la visibilidad de los formularios [fuente propia]

En este caso, la función llama a *toggleVisibility* (ver Figura 3-68), la cual hace que el formulario de fecha y hora de la lesión se muestre si estaba oculto, y se oculte si ya estaba visible. Cabe destacar que para el resto de los formularios se realiza exactamente de la misma manera, por lo que solo se explicará con este ejemplo.

```
private fun toggleVisibility(layout: LinearLayout) {
    if (layout.visibility == View.VISIBLE) {
        layout.visibility = View.GONE
    } else {
        layout.visibility = View.VISIBLE
    }
}
```

Figura 3-68. Método para alternar la visibilidad del formulario [fuente propia]

Finalmente, al igual que se hizo con los datos de la actividad *DatosBasicos*, se ejecuta la función *onBackPressed()* cuando se presiona el botón de retroceso en el dispositivo Android. Esta, además de recopilar los datos de texto y de validar la fecha y la hora ingresadas por el usuario para asegurarse de que estén en un formato correcto, comprueba que la fecha no sea posterior a la actual (ver Figura 3-69). Si algún dato es inválido, se muestra un mensaje de error en el campo correspondiente y evita que el usuario retroceda mandando los datos a *MenuPrincipal*.

```
private fun esFechaPosterior(fechaIngresada: String, fechaActual: String): Boolean {
    // Convertir las fechas a objetos Calendar para compararlas
    val formato = SimpleDateFormat( pattern: "dd/MM/yyyy", Locale.getDefault())
    val calFechaIngresada = Calendar.getInstance()
    val calFechaActual = Calendar.getInstance()

    calFechaIngresada.time = formato.parse(fechaIngresada)
    calFechaActual.time = formato.parse(fechaActual)

    return calFechaIngresada.after(calFechaActual)
}
```

Figura 3-69. Método para verificar si una fecha es posterior a otra [fuente propia]

3.3.3.6 Movimiento

De manera similar a cómo se diseñaron las actividades de *Evaluacion* y *Tratamiento*, se ha desarrollado *Movimiento*. Esta también cuenta con formularios desplegables y campos de texto editables, sin embargo, a diferencia de las otras dos, existen unos datos preestablecidos (ver Figura 3-70) para los campos asociados a la información del primer asistente del herido, ya que, en un principio, estarían asociados al dispositivo móvil personal de cada uno.

```
companion object {
    const val KEY_NOMBRE_SOCORRISTA = "nombre_socorrista"
    const val KEY_APELLIDOS_SOCORRISTA = "apellidos_socorrista"
    const val KEY_ID_SOCORRISTA = "sexo_socorrista"

    const val DEFAULT_NOMBRE = "Pablo"
    const val DEFAULT_APELLIDOS = "García Rodríguez"
    const val DEFAULT_ID = "71903419Z"
}
```

Figura 3-70. Constantes para valores preestablecidos [fuente propia]

Sin embargo, si alguno de estos datos es modificado y enviado a la actividad principal a través del método *onBackPressed()* como ya vimos en explicaciones previas, esta nueva información quedaría almacenada (ver Figura 3-71).

```
    val nombreSocorristaVuelta = bundle?.getString( key: "nombreSocorrista_vuelta")
    val apellidosSocorristaVuelta = bundle?.getString( key: "apellidosSocorrista_vuelta")
    val idSocorristaVuelta = bundle?.getString( key: "IDsocorrista_vuelta")

    val nombreSocorrista =
        if (nombreSocorristaVuelta.isNullOrEmpty()) {
            sharedPreferences.getString(KEY_NOMBRE_SOCORRISTA, DEFAULT_NOMBRE)
        } else {
            nombreSocorristaVuelta
        }
    [...]
```

Figura 3-71. Obtención de los valores predeterminados y establecimiento en los *EditText* correspondientes [fuente propia]

Además, en esta actividad se ha implementado el uso de *spinners* (menús desplegables) para facilitar el uso de la aplicación al usuario. Para ello, se llama a la función *spinnerCategoriaEvacuacion()* dentro del método *onCreate().* Esta función, como se ve en la Figura 3-72, primero obtiene una referencia al *spinner* de la vista mediante su identificador y luego crea un adaptador para vincular los datos del *spinner* con un conjunto predefinido de opciones, que en este caso son obtenidas de un recurso de cadenas definido en el archivo de recursos de la aplicación.

```
private fun spinnerCategoriaEvacuacion() {
    val spinnerCategoriaEvacuacion = findViewById<Spinner>(R.id.spinnerCategoriaEvacuacion)
    val adapter = ArrayAdapter.createFromResource(
        context: this,
        R.array.categoria_evacuacion,
        android.R.layout.simple_spinner_item
    )
    adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item)
    spinnerCategoriaEvacuacion.adapter = adapter
    [...]
```

Figura 3-72. Obtención de la referencia al *spinner* y creación del adaptador [fuente propia]

Posteriormente, como se refleja en la Figura 3-73, se asigna este adaptador al *spinner* para que se muestren las opciones. Además, se verifica también si hay algún valor seleccionado previamente y, en caso de que exista, se establece esa opción como seleccionada en el *spinner*.

```
if (intent.hasExtra( name: "categoria_vuelta")) {
    categoria_evacuacion = intent.getStringExtra( name: "categoria_vuelta")
    val position = adapter.getPosition(categoria_evacuacion)
    if (position != -1) {
        spinnerCategoriaEvacuacion.setSelection(position)
    }
}
[...]
```

Figura 3-73. Obtención del valor seleccionado del *Intent* si existe [fuente propia]

Además de la configuración inicial del *spinner*, se define un *listener* (*onItemSelectedListener*) para manejar los eventos de selección de elementos en el *spinner* (ver Figura 3-74). Cuando se selecciona un elemento de este, se actualiza la variable *categoria_evacuacion* con el valor del elemento seleccionado. Si no se selecciona ningún elemento, el *listener* no realiza ninguna acción.

```
    [...]
    spinnerCategoriaEvacuacion.onItemSelectedListener = object : AdapterView.OnItemSelectedListener {
        override fun onItemSelected(parent: AdapterView<*>, view: View, position: Int, id: Long) {
            categoria_evacuacion = parent.getItemAtPosition(position).toString()
        }

        override fun onNothingSelected(parent: AdapterView<*>?) {
        }
    }
}
```

Figura 3-74. Definición del *listener* [fuente propia]

Finalmente, al igual que se hizo en el resto de las actividades, se ejecuta el método *onBackPressed()* para enviar la información correspondiente a *MenuPrincipal* a través de los *intents* respectivos.

3.3.3.7 ConstantesVitales

En esta actividad, tal y como se muestra en la Figura 3-75, primero se inicializa una lista vacía para almacenar los registros de las constantes vitales del paciente y se configura un *RecyclerView* y un adaptador para mostrar los registros de constantes vitales en forma de lista desplazable. Esto permite al usuario ver los registros de forma ordenada y fácil de leer. Los registros se mostrarán utilizando el diseño definido en *RegistroAdapter*.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_constantes_vitales)

    // Inicializar la lista de registros
    registros = mutableListOf()

    // Configurar el RecyclerView y el adaptador
    val recyclerView = findViewById<RecyclerView>(R.id.registro_constantes)
    adapter = RegistroAdapter(registros)
    recyclerView.adapter = adapter
    recyclerView.layoutManager = LinearLayoutManager( context: this)
    [...]
```

Figura 3-75. Inicialización de la lista de registros y configuración del *RecyclerView* [fuente propia]

Además, de la misma manera que se explicó en el apartado anterior, se configuran dos *spinners* para permitir al usuario seleccionar la escala de dolor y el estado del herido. Estos elementos proporcionan información adicional sobre la condición del paciente.

Dentro del método *onCreate()* también se verifica si hay registros de constantes vitales enviados desde la actividad *MenuPrincipal* (ver Figura 3-76). Si se reciben registros, se agregan a la lista de registros y se notifica al adaptador que los datos han cambiado para que se actualice la vista.

```
val registrosRecibidos = intent.getParcelableArrayListExtra<Registro>( name: "registros_vuelta")
if (registrosRecibidos != null && registrosRecibidos.isNotEmpty()) {
    // Agregar los registros recibidos a la lista
    registros.addAll(registrosRecibidos)
    // Notificar al adaptador que los datos han cambiado
    adapter.notifyDataSetChanged()
}
[...]
```

Figura 3-76. Verificación de si hay registros enviados desde *MenuPrincipal* [fuente propia]

A continuación, tal y como se refleja en la Figura 3-77, se obtienen los valores de las constantes vitales recibidos desde la actividad *MenuPrincipal* y se muestran en los campos de texto correspondientes de manera que el usuario puede ver las constantes vitales previamente ingresadas.

```
val bundle = intent.extras
val hora_constantes = obtenerHoraActual()
val pulso = bundle?.getString( key: "pulso_vuelta")
val presion_sanguinea = bundle?.getString( key: "presion_sanguinea_vuelta")
val tasa_respiracion = bundle?.getString( key: "tasa_respiracion_vuelta")
val saturacion_oxigeno = bundle?.getString( key: "saturacion_oxigeno_vuelta")
findViewById<EditText>(R.id.hora_constantes).setText(hora_constantes)
findViewById<EditText>(R.id.pulso).setText(pulso)
findViewById<EditText>(R.id.presion_sanguinea).setText(presion_sanguinea)
findViewById<EditText>(R.id.tasa_respiracion).setText(tasa_respiracion)
findViewById<EditText>(R.id.saturacion_oxigeno).setText(saturacion_oxigeno)
[...]
```

Figura 3-77. Obtención de los valores recibidos y establecimiento en los campos de texto asociados [fuente propia]

Finalmente, como muestra el código de la Figura 3-78, se configuran los eventos de los botones "*Guardar Registro*" y "*Eliminar Registro*". El primero de ellos permite al usuario guardar un nuevo registro de constantes vitales, verificando primero si la hora ingresada es válida, mientras que el otro permite al usuario eliminar el último registro de constantes vitales añadido.

```
    [...]
    // Configurar el clic del botón "Guardar Registro"
    val btnGuardarRegistro = findViewById<Button>(R.id.btnGuardarRegistro)
    btnGuardarRegistro.setOnClickListener { it: View!
        val hora_constantesText: EditText = findViewById<EditText>(R.id.hora_constantes)
        val hora_constantes = hora_constantesText.text.toString()
        val isValidHora = esHoraValida(hora_constantes) || hora_constantes.isBlank()
        if (isValidHora) {
            guardarRegistro()
        } else {
            hora_constantesText.error = "Invalid time"
        }
    }
    val btnEliminarRegistro = findViewById<Button>(R.id.btnEliminarRegistro)
    btnEliminarRegistro.setOnClickListener { it: View!
        eliminarRegistro()
    }
}
```

Figura 3-78. Configuración del botón para guardar y eliminar el registro [fuente propia]

Concretamente, el código que se acaba de explicar emplea las funciones *guardarRegistro()* y *eliminarRegistro()*. La primera de ellas, inicialmente, obtiene los valores ingresados por el usuario desde los campos de texto y los *spinners* de la interfaz de usuario como se ve en la Figura 3-79.

```
fun guardarRegistro() {
    val hora = findViewById<EditText>(R.id.hora_constantes).text.toString()
    val pulso = findViewById<EditText>(R.id.pulso).text.toString()
    val presionSanguinea = findViewById<EditText>(R.id.presion_sanguinea).text.toString()
    val tasaRespiracion = findViewById<EditText>(R.id.tasa_respiracion).text.toString()
    val saturacionOxigeno = findViewById<EditText>(R.id.saturacion_oxigeno).text.toString()
    val estadoHerido = obtenerValorSpinner(R.id.spinnerEstadoHerido)
    val escalaDolor = obtenerValorSpinner(R.id.spinnerEscalaDolor)
    [...]
```

Figura 3-79. Método para guardar el registro y mostrar los datos ingresados en el *RecyclerView* [fuente propia]

Luego, crea un nuevo objeto de tipo *Registro* (ver Figura 3-80) utilizando los valores obtenidos de tal manera que representa una única entrada de las constantes vitales del paciente. Una vez creado, se agrega a la lista de registros existente, que almacena todas las entradas de constantes vitales del paciente. Posteriormente, se actualiza la información de la interfaz del usuario.

```
    val registro = Registro(
        hora,
        pulso,
        presionSanguinea,
        tasaRespiracion,
        saturacionOxigeno,
        estadoHerido,
        escalaDolor
    )

    // Agregar el registro a la lista
    registros.add(registro)

    // Notificar al adaptador que se agregó un nuevo registro
    adapter.notifyDataSetChanged()
}
```

Figura 3-80. Diferentes operaciones realizadas con los registros [fuente propia]

También se considera la opción de eliminar los registros de constantes vitales como se muestra en la Figura 3-81.

```
fun eliminarRegistro() {
    if (registros.isNotEmpty()) {
        registros.removeAt( index: registros.size - 1) // Eliminar el último registro agregado
        adapter.notifyDataSetChanged() // Notificar al adaptador que los datos han cambiado
    }
}
```

Figura 3-81. Código empleado para la eliminación de los registros [fuente propia]

Finalmente, al igual que se hizo en el resto de las actividades, al volver a la actividad *MenuPrincipal* se invoca el método *onBackPressed()*, que retorna la información correspondiente, comprobando que la hora definida tiene un formato adecuado.

4 RESULTADOS Y VALIDACIÓN DE LA APLICACIÓN

Como resultado del desarrollo detallado en el capítulo 3, se ha generado una aplicación denominada Digital NFMC. Se dedica este apartado a describir su funcionamiento de forma práctica para servir a dos propósitos, ilustrar que cumple los requisitos iniciales y servir como guía al posible usuario de dicha aplicación.

4.1 Descripción del escenario de prueba

Inicialmente se partirá de un supuesto donde existen dos bajas en la zona de combate, se identificarán a los pacientes como Pepe y María. Además, Jesús será el primer asistente con la aplicación instalada en su dispositivo móvil, tal y como se refleja en el esquema de la Figura 4-1.

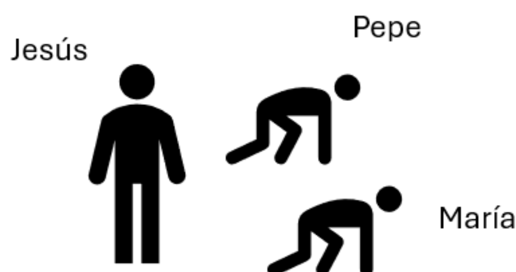


Figura 4-1. Representación gráfica del escenario de prueba de la aplicación [fuente propia]

Los pacientes disponen con un tag NFC, equipado a todos los combatientes al inicio de la operación, que contiene la información personal básica estandarizada por la OTAN que se muestra en la Figura 4-2. Es relevante resaltar que estos campos deben ser completados por una entidad competente antes de ser entregados a sus respectivos propietarios. En caso de que dicha autoridad introduzca una fecha de nacimiento incorrecta, ya sea por un error en el día, mes o año, o una fecha futura, se generará un mensaje en pantalla para notificar el error, como se muestra en la Figura 4-3. Además, se impedirá que retroceda a la pestaña del menú principal. Asimismo, la aplicación está configurada de manera que en el campo correspondiente al ID, los primeros ocho dígitos deben ser números y el último una letra mayúscula, lo que simplifica la tarea de ingresar los datos.

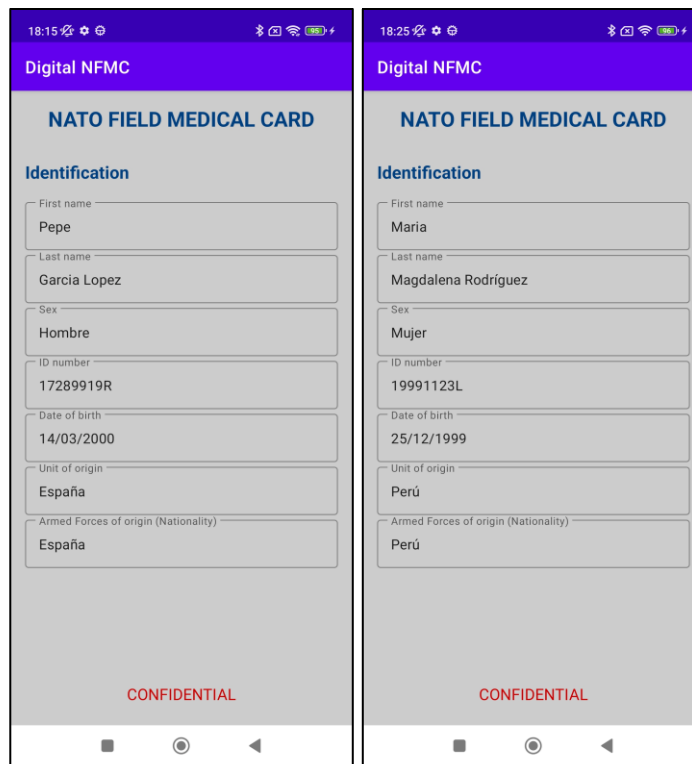


Figura 4-2. Información básica que contendrían los tags NFC de Pepe y María [fuente propia]

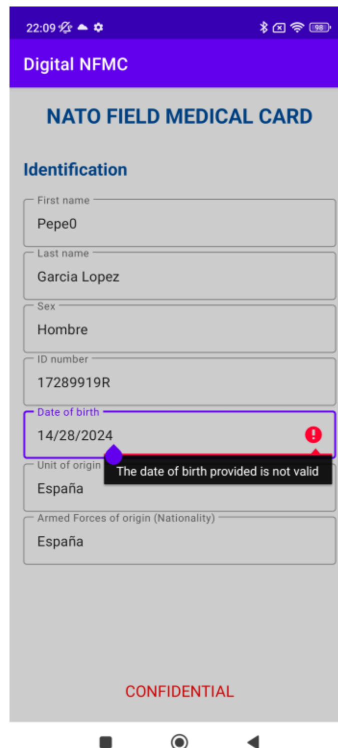


Figura 4-3. Mensaje de error que se mostraría en caso de introducir una fecha de nacimiento incorrecta [fuente propia]

4.1.1 Conjunto de pruebas realizadas

Se va a suponer que Pepe, en una situación más crítica que María debido a una fractura en la rodilla izquierda y una herida perforante en la parte posterior del hombro izquierdo causada por un disparo, requiere atención médica urgente. Para garantizar un seguimiento efectivo de su estado, se registrarán

estos datos en el tag que Pepe lleva consigo, lo que permitirá informar a todas las FST responsables de su evacuación.

Por consiguiente, Jesús procederá a escanear el tag de Pepe con su dispositivo móvil. Posteriormente introduce en la aplicación su ID y una clave previamente proporcionada antes de su despliegue en la zona de conflicto, lo que garantiza un nivel mínimo de seguridad local en caso de pérdida del dispositivo móvil (ver Figura 4-4).

En caso de que Jesús, debido a la situación de tensión en la que se encuentra, ingrese de manera incorrecta los datos, una alerta aparecerá en la pantalla, denegando el acceso al sistema, tal como se ilustra en la Figura 4-4.

Una vez identificado correctamente, Jesús accede a la pantalla principal de la aplicación, como se muestra en la Figura 4-5. Esta pantalla proporciona acceso directo a las diversas secciones que abarcan desde la información personal a la médica del paciente, facilitando así su gestión de manera eficiente. Además, también se le muestran una serie de instrucciones en color llamativo para garantizar que se hace un uso correcto de la aplicación.

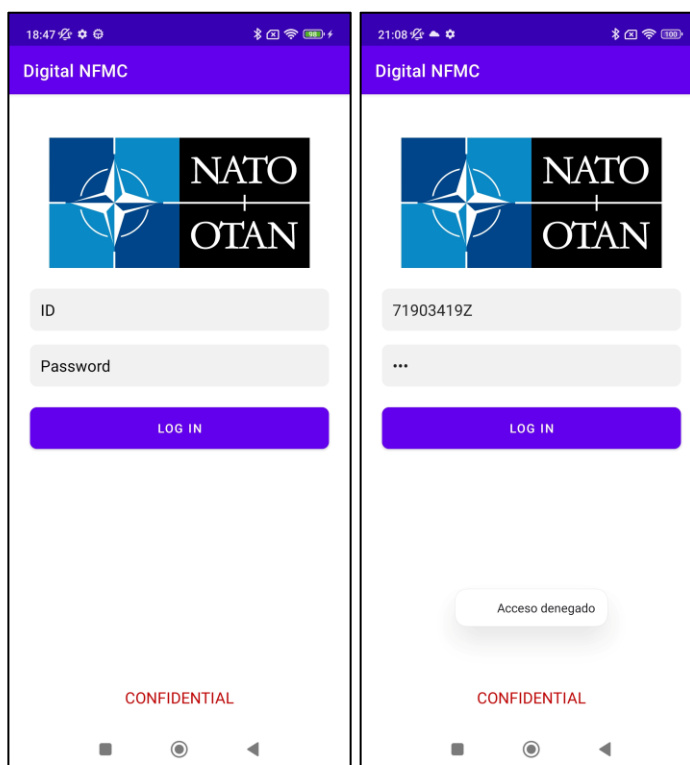


Figura 4-4. Pantalla de inicio de sesión a la aplicación en la izquierda y el mensaje de error correspondiente [fuente propia]

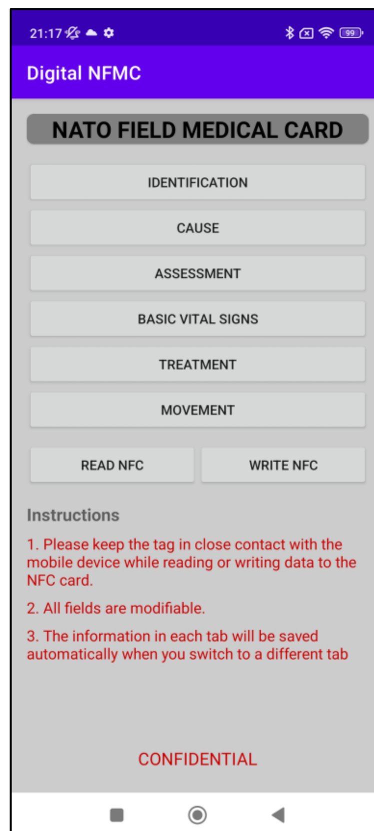


Figura 4-5. Pantalla del menú principal de la aplicación [fuente propia]

Jesús, en su búsqueda por verificar los datos del herido, accede al apartado de "*Identification*". Estos datos, originalmente registrados en el tag de Pepe, se presentan inicialmente tal como se mostraron en la Figura 4-3 y deben permanecer inalterados. Si, por alguna razón, Jesús inadvertidamente modifica un dato y retrocede utilizando la flecha del dispositivo móvil, se activará un aviso, como se ilustra en la Figura 4-6. Este mensaje alerta a Jesús sobre cualquier modificación realizada, asegurando así que su acción sea intencionada. Si Jesús confirma los cambios seleccionando "*Yes*", retornará a la pantalla principal con las modificaciones efectuadas correctamente. Sin embargo, si opta por "*No*", se redirigirá nuevamente a la pantalla de "*Identification*" para corregir cualquier campo modificado.

Por otra parte, Jesús debe introducir ahora la causa de la lesión de Pepe, accediendo a la pestaña "*Cause*". Al principio, se muestra una pantalla como la que se ve en la imagen izquierda de la Figura 4-7, con un breve mensaje explicativo para guiar al usuario sobre cómo representar cada tipo de herida. En el caso de Jesús, debe arrastrar los iconos "*Fracture*" y "*Injury with perforation*" a la zona del cuerpo afectada, como se muestra en la imagen derecha de la Figura 4-7. Es relevante destacar que, junto con la selección de la imagen, se genera un botón de eliminación asociado, permitiendo corregir cualquier error de ubicación. Cada tipo de lesión se representa con un color distinto, coincidiendo con los colores del texto dentro los botones generados. Esto se hace para agilizar la acción del usuario, en este caso Jesús, en situaciones críticas que puedan requerir una respuesta rápida, como en un escenario de combate.

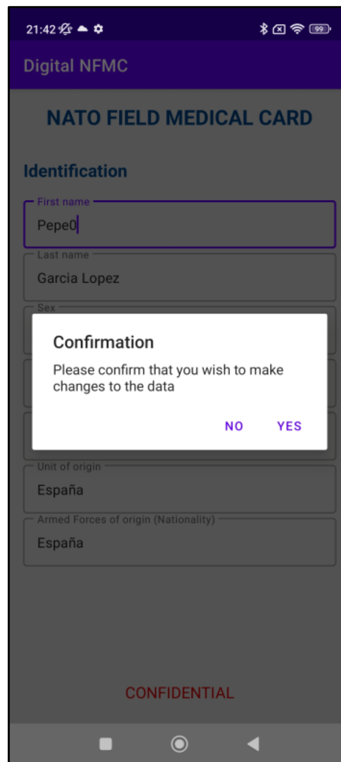


Figura 4-6. Mensaje mostrado en el caso de que algún dato del apartado *Identification* sea modificado [fuente propia]

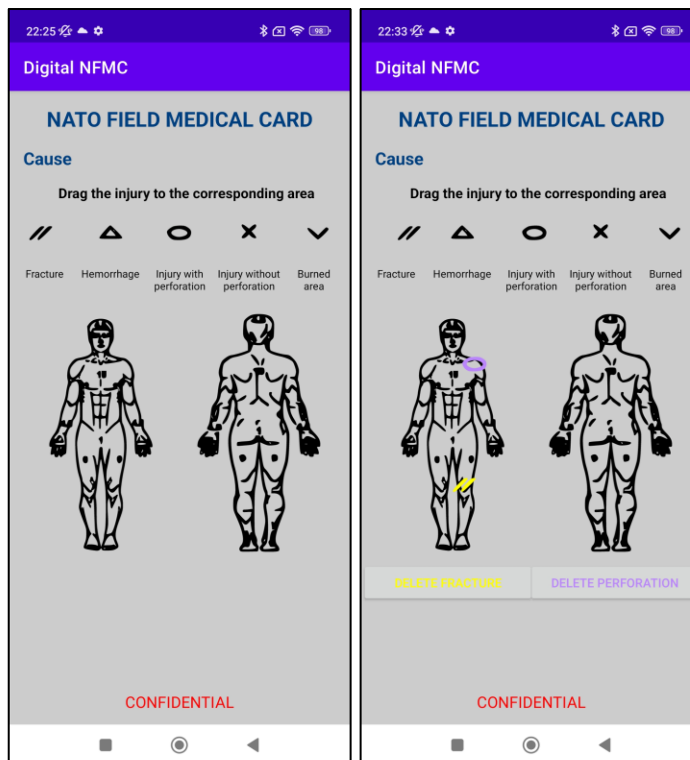


Figura 4-7. Pantalla inicial de la pestaña "*Cause*" en la imagen de la izquierda y esa misma pantalla con los datos modificados en la imagen de la derecha [fuente propia]

Una vez se han rellenado los motivos de la lesión, Jesús podría entrar al apartado de "*Assessment*" en donde, inicialmente, se le mostraría una pantalla como la que se ve en la imagen de la izquierda de la Figura 4-8. Si Jesús pulsa sobre cualquiera de los diferentes desplegables, se le mostrarán diferentes apartados donde puede escribir, con una breve explicación inicial del contenido que debe proporcionar, como se ve en la imagen central de la Figura 4-8. Además, los apartados "*Date and time of injury or illness*" y "*Time of examination*" se rellenarán inicialmente con la fecha y hora del dispositivo móvil de Jesús, para intentar agilizar el proceso. Ambos campos son editables y están configurados de tal modo que, si en alguno de ellos se escribe una fecha u hora incorrecta, a Jesús le saldrá un aviso (Figura 4-9) y no le dejará retroceder a la pantalla del menú principal. Asimismo, en "*Date and time of injury or illness*" no se puede introducir una hora o fecha posterior a la actual (ver imagen de la derecha en la Figura 4-8) ya que se asume que Jesús abrirá la aplicación y completará los campos poco después de la lesión de Pepe.

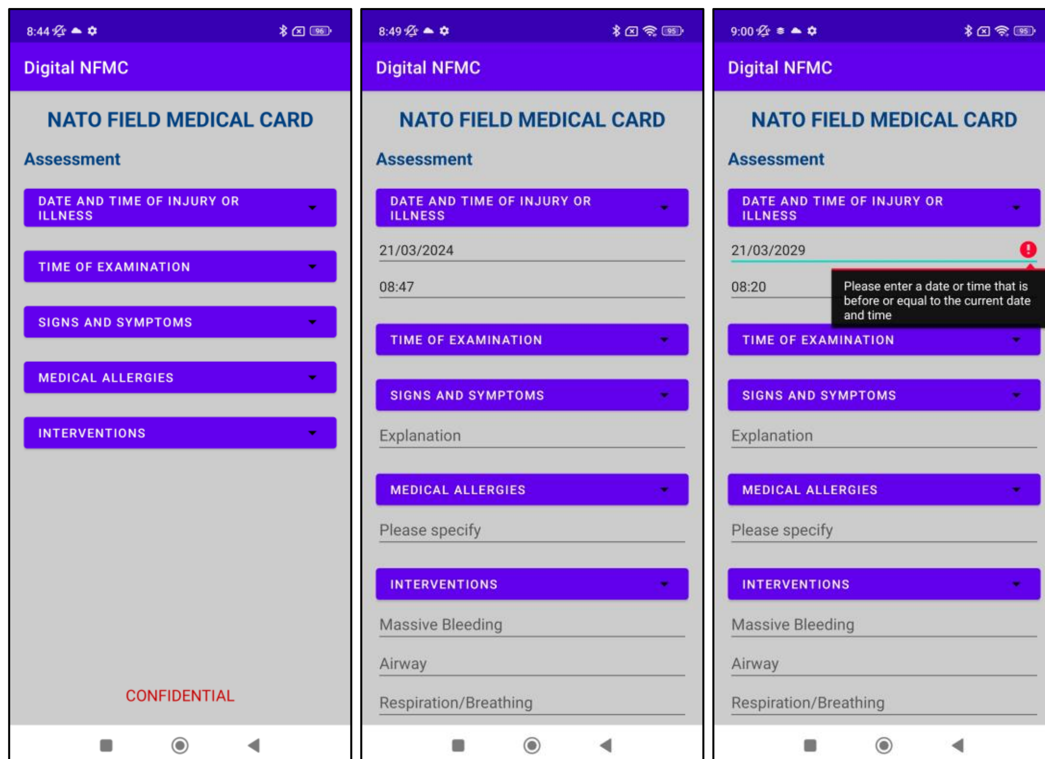


Figura 4-8. Pantalla inicial de la pestaña "*Assessment*" en la izquierda, desplegables abiertos de esa misma pestaña en la imagen central, y mensaje de error en la fecha en la imagen de la derecha [fuente propia]

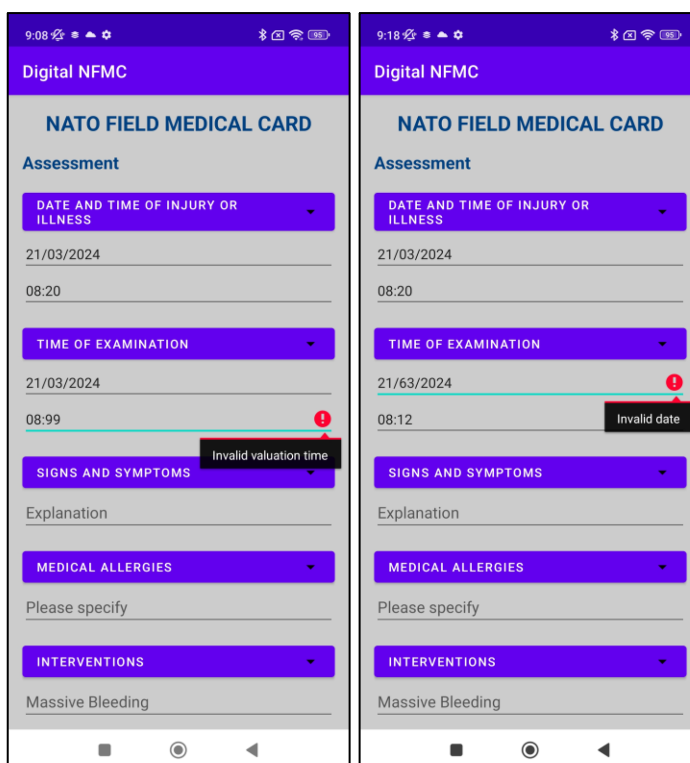


Figura 4-9. Mensaje de error al introducir hora incorrecta en la izquierda y mensaje de error al introducir fecha incorrecta en la derecha [fuente propia]

A continuación, Jesús procederá a completar el campo *"Signs and symptoms"* según se ilustra en la Figura 4-10. Las intervenciones posteriores se llevarán a cabo en el próximo ROLE, basándose en la información proporcionada por Jesús. Este proceso se facilita gracias a la actualización automática de los datos en el tag de Pepe, visible en la Figura 4-10, que a su vez se transmite a un servidor en línea, del cual se muestra la captura de una parte de sus datos en la Figura 4-11. Este servidor permite que el personal de diferentes ROLES acceda a una base centralizada de información en tiempo real sobre todas las bajas y sus características individuales.

Es importante destacar que, si la conexión con el servidor se establece correctamente, se mostrará un mensaje en la pantalla, tal como se representa en la imagen de la izquierda en la Figura 4-12. Por otro lado, en caso de que surja algún problema de conexión, el usuario también recibirá una notificación, como se observa en la imagen de la derecha en la Figura 4-12.

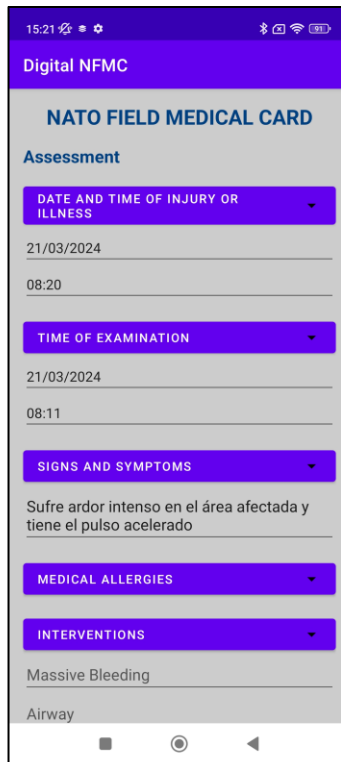


Figura 4-10. Apartado rellenado por Jesús [fuente propia]

id	nombre	apellidos	sexo	numeroId	fechaNacimiento	lugarOrigen	fuerzasArmadasOrigen	fecha	hora	fechaEvaluacion	horaEvaluacion	senales	alergias	sangrado	airway	respiracion	circulacic
												y tiene el...					
14	Pepe	Garcia Lopez	Hombre	17289919R	14/03/2000	España	España	21/03/2024	08:20	21/03/2024	08:11	Sufre ardor intenso en el área afectada y tiene el...	NULL	NULL	NULL	NULL	NULL
15	Pepe	Garcia Lopez	Hombre	17289919R	14/03/2000	España	España	21/03/2024	08:20	21/03/2024	08:11	Sufre ardor intenso en el área afectada y tiene el...	NULL	NULL	NULL	NULL	NULL
16	Pepe	Garcia Lopez	Hombre	17289919R	14/03/2000	España	España	21/03/2024	08:20	21/03/2024	08:11	Sufre ardor intenso en el área afectada y tiene el...	NULL	NULL	NULL	NULL	NULL

Figura 4-11. Parte de los datos guardados en el servidor [fuente propia]

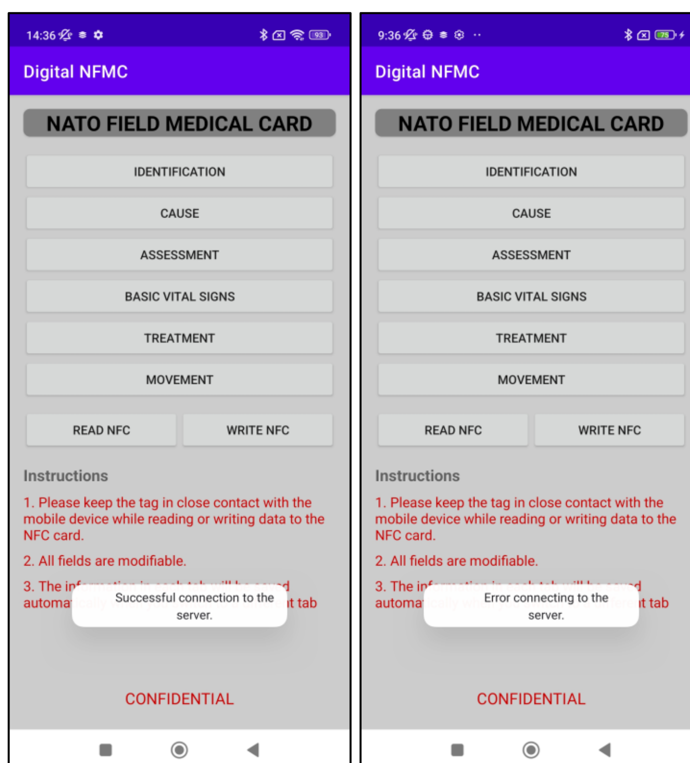


Figura 4-12. Mensaje mostrado si la información se envía correctamente al servidor en la imagen de la izquierda y mensaje mostrado si existe un error de conexión en la imagen derecha [fuente propia]

Manteniendo el orden seguido hasta ahora en lo que al acceso de las pestañas se refiere, Jesús accederá ahora a “*Basic vital signs*”, donde se encontrará con una pantalla similar a la que se muestra en la imagen de la izquierda en la Figura 4-12. Al igual que en casos anteriores, la hora se registra automáticamente, aunque puede ser modificada según sea necesario y, en caso de introducir una hora con formato incorrecto, se mostrará un aviso en pantalla (ver imagen de la derecha en la Figura 4-12) y no le dejará retroceder a la pantalla del menú principal.

En esta sección, Jesús completará únicamente los campos relacionados con el pulso, la tasa respiratoria, el estado del paciente y su nivel de dolor, ya que estos no requieren ningún instrumento adicional. Además, se conservará un único registro de estos datos, como se muestra en la imagen central en la Figura 4-13 para que el siguiente nivel de evacuación esté informado sobre la evolución de Pepe hasta ese momento.

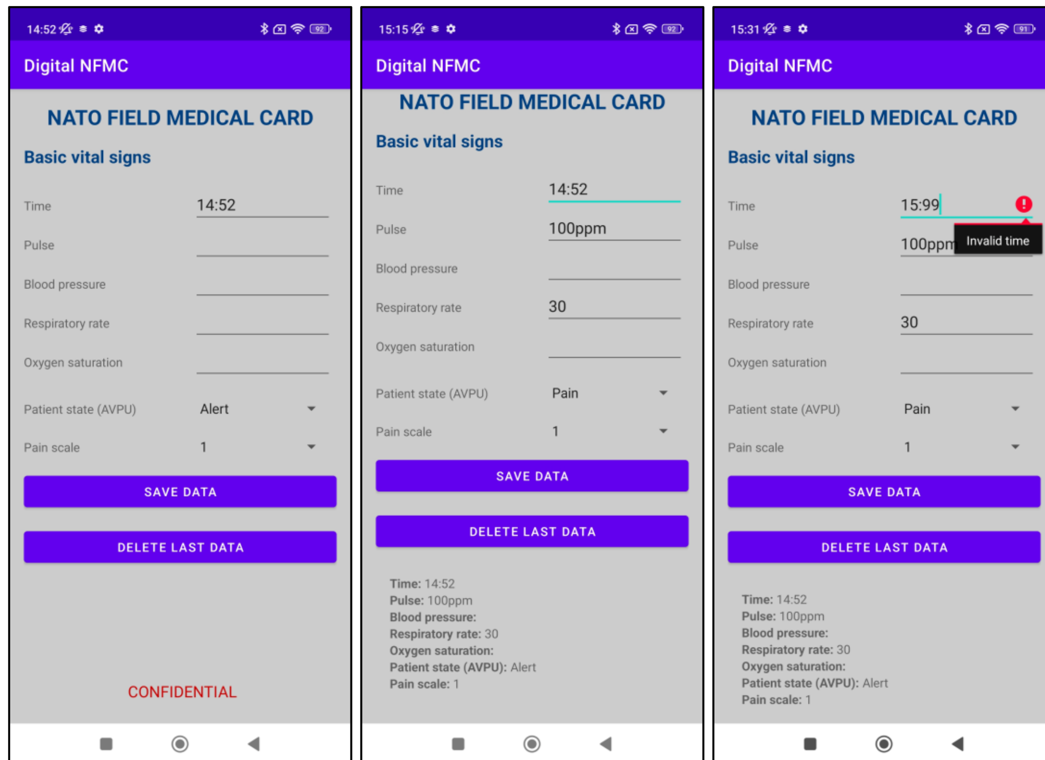


Figura 4-13. Pantalla inicial de "*Basic vital signs*" en la imagen de la izquierda, datos proporcionados, incluyendo su registro en la imagen central y mensaje de error en la imagen derecha [fuente propia]

A continuación, Jesús accederá a la pestaña de "*Treatment*", donde se le mostrará una pestaña como la de la Figura 4-13. En ella, solo cubrirá los datos correspondientes a "*Fluids*", "*Tourniquet/TQ*" y "*Hypothermia intervention*" ya que a Pepe le ha sido administrada un cierta cantidad de agua y no se le ha realizado ninguna de las dos intervenciones mencionadas. Por tanto, pasaría a tener una pantalla como la que se muestra en la imagen de la derecha de la Figura 4-13.

Al igual que vimos en los casos anteriores, si cualquiera de las horas se rellena con un formato incorrecto, se mostraría un aviso en la pantalla y no se dejaría a Jesús retroceder al menú principal.

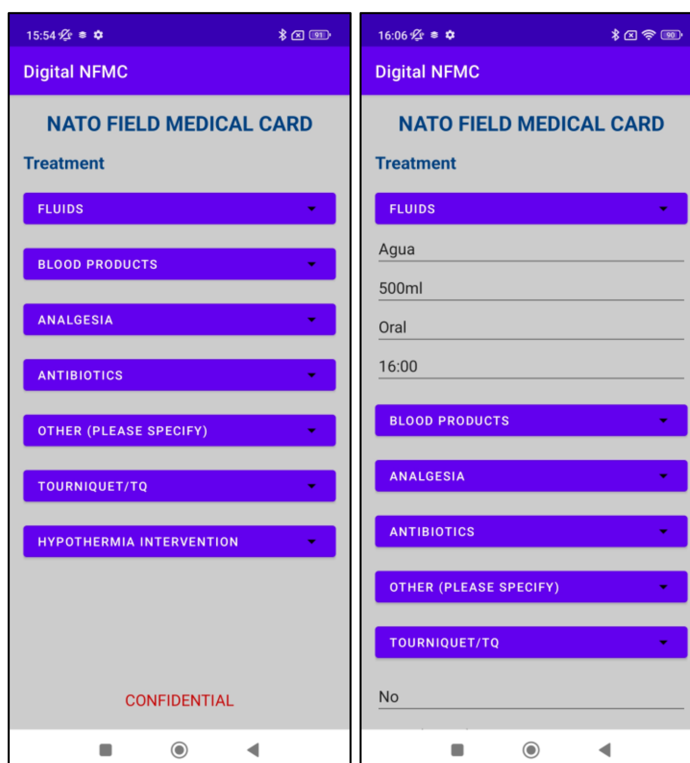


Figura 4-14. Pantalla inicial de "*Treatment*" en la imagen de la izquierda y datos cubiertos en la imagen de la derecha [fuente propia]

En este punto, Jesús se encuentra frente a la tarea de completar los datos correspondientes a la pantalla "*Movement*", cuyo diseño inicial se muestra en la imagen de la izquierda de la Figura 4-15. Es importante destacar que la información relacionada con "*First responder information*" (ver imagen central de la Figura 4-15) se completará automáticamente con datos predefinidos almacenados en el dispositivo móvil de Jesús. En este contexto, la categoría de evacuación se designaría como "*Priority*" de acuerdo con el análisis efectuado por Jesús.

Una vez que Jesús ha completado todos los datos, procederá a ingresarlos en el tag de Pepe, asegurándose así de actualizar la información contenida en él y de enviar los datos al servidor en línea. Esto permitirá que las diferentes FST accedan a la información y analicen la situación, preparando los tratamientos necesarios para Pepe antes de su llegada.

En caso de encontrar algún problema al escribir en el tag de Pepe, Jesús recibirá una notificación en pantalla, como se muestra en la imagen izquierda de la Figura 4-16. Por otro lado, si todo funciona correctamente, Jesús verá un mensaje como el de la imagen derecha de la misma figura.

Dada la situación de estrés en la que se encuentra Jesús, existe la posibilidad de que, al escribir en el tag, pueda sobrescribir accidentalmente el tag de María. Para prevenir este problema, la aplicación implementa un mecanismo de seguridad que consiste en leer el tag antes de escribir en él. Si el número ID en la tarjeta a escribir es diferente al que contiene la aplicación en el dispositivo móvil, se mostrará una advertencia en pantalla, como se ilustra en la Figura 4-17. Por otro lado, si efectivamente contienen el mismo ID, se mostraría el mensaje que se ve la imagen de la derecha de la Figura 4-16.

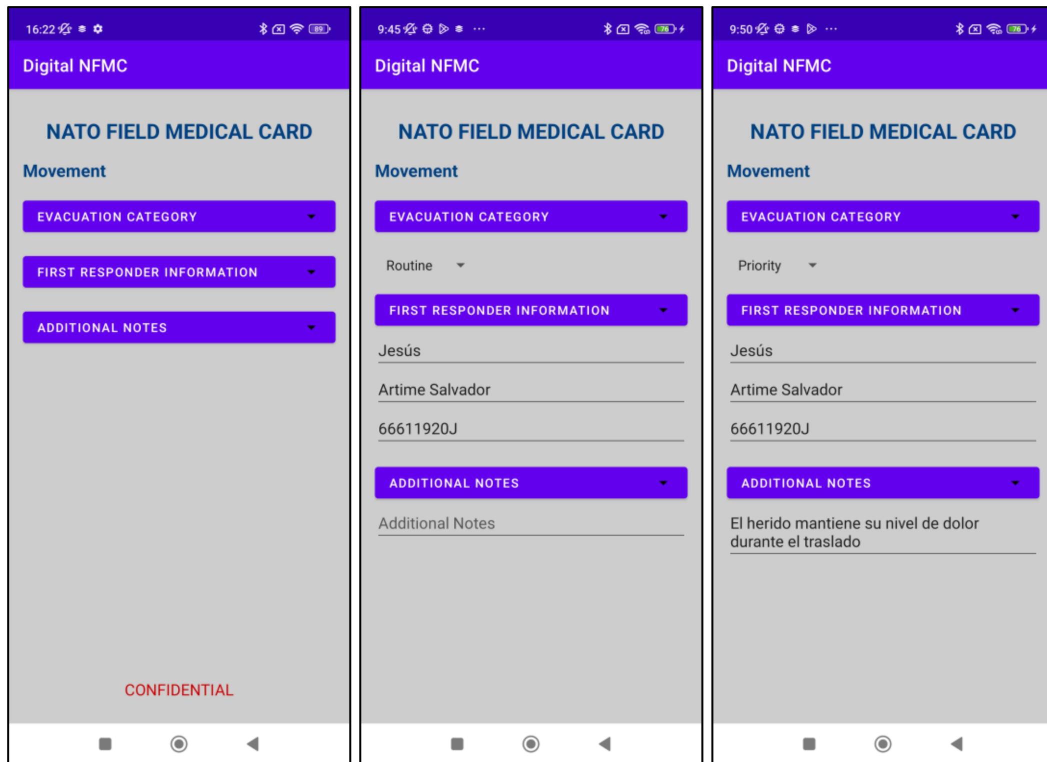


Figura 4-15. Pantallas asociadas a "*Movement*" [fuente propia]

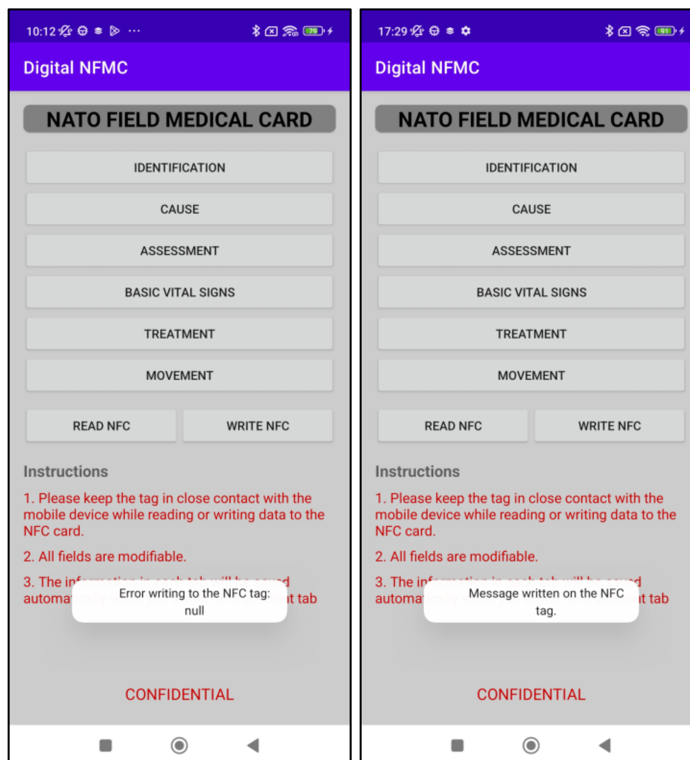


Figura 4-16. Mensaje de error en la imagen de la izquierda y mensaje que indica que se ha realizado correctamente la escritura en la imagen de la derecha [fuente propia]

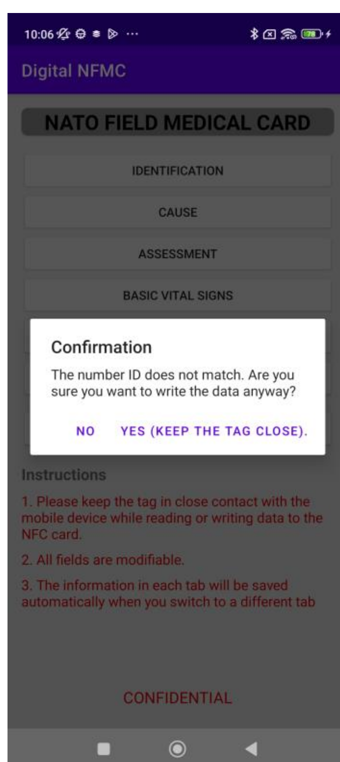


Figura 4-17. Mensaje de advertencia en caso de que los ID de las tarjetas NFC no coincidan [fuente propia]

En el supuesto de que Pepe sea atendido por alguna de las FST disponibles, los profesionales médicos presentes podrían haber accedido a su información previamente, a través del servidor, permitiendo así que los tratamientos necesarios estén preparados para su aplicación inmediata.

En este contexto, todas las intervenciones realizadas sobre el paciente deben ser actualizadas en el tag NFC, es decir, escritas en él. Para ello, el primer paso es leer dicho tag. Inicialmente, el usuario debe presionar el botón "*Read NFC*", y si la operación es exitosa, se mostrará un mensaje de confirmación, como se ilustra en la imagen izquierda de la Figura 4-18.

Además, si la lectura de los datos se realiza correctamente, el usuario recibirá un mensaje indicando este resultado, tal como se muestra en la imagen central de la misma figura. Por otro lado, si el usuario intenta abrir la aplicación con la opción NFC desactivada en el teléfono, también se le notificará de esta situación, como se representa en la imagen derecha de la Figura 4-18.

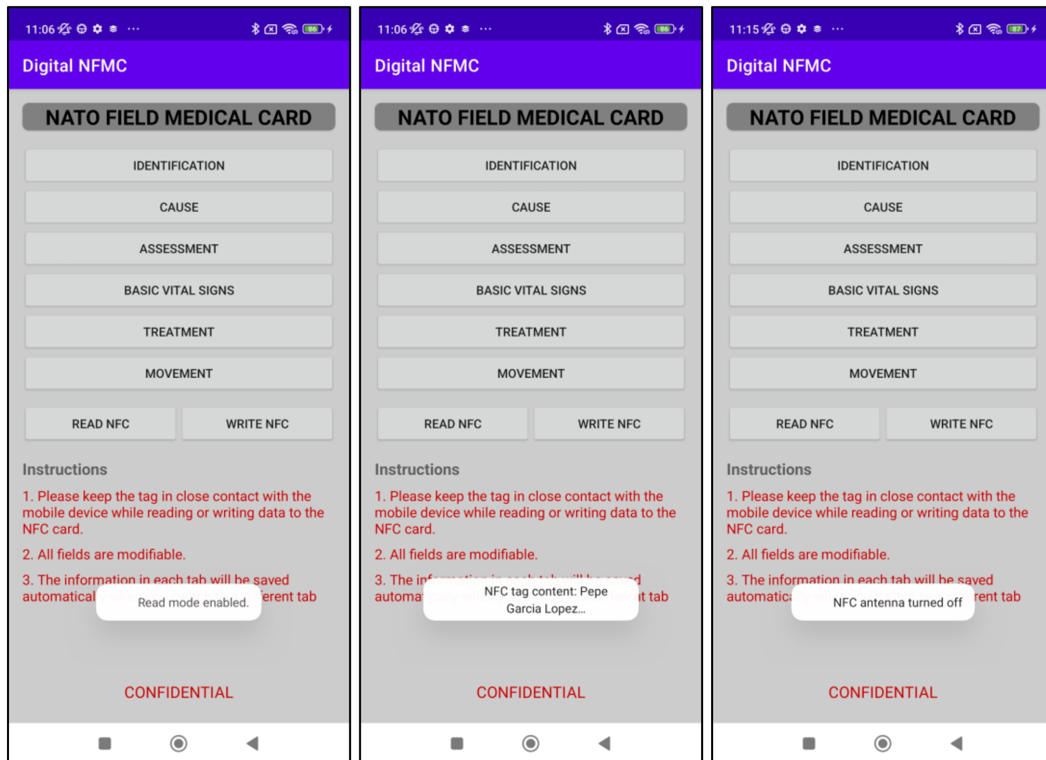


Figura 4-18. Posibles mensajes que podría ver el usuario durante el proceso de lectura de un tag [fuente propia]

4.2 Capacidad de las tarjetas NFC

En esta sección se ilustrará el escaso espacio que requieren los datos almacenados en las tarjetas NFC, con el propósito de demostrar que la capacidad de almacenamiento de dichas tarjetas no representa un factor limitante en el presente estudio. Inicialmente, como se muestra en la imagen izquierda de la Figura 4-19, la capacidad de los tags sin ningún dato almacenado es de 4094 bytes. Considerando los datos escritos durante el ejemplo previamente explicado, se habrían ocupado un total de 343 bytes, como se evidencia en la imagen central de la Figura 4-19, lo que representa menos de una décima parte del espacio total disponible. Por último, en caso de necesitar realizar quince registros, el espacio total ocupado ascendería a 693 bytes (ver imagen de la derecha de la Figura 4-19), lo que indica que aún se podrían incluir 73 registros adicionales (bajo la suposición de que todos tengan un tamaño similar), número que se considera suficiente para garantizar la trazabilidad de la información médica completa del paciente y su correcto seguimiento automatizado, lo cual es el objeto de este trabajo.

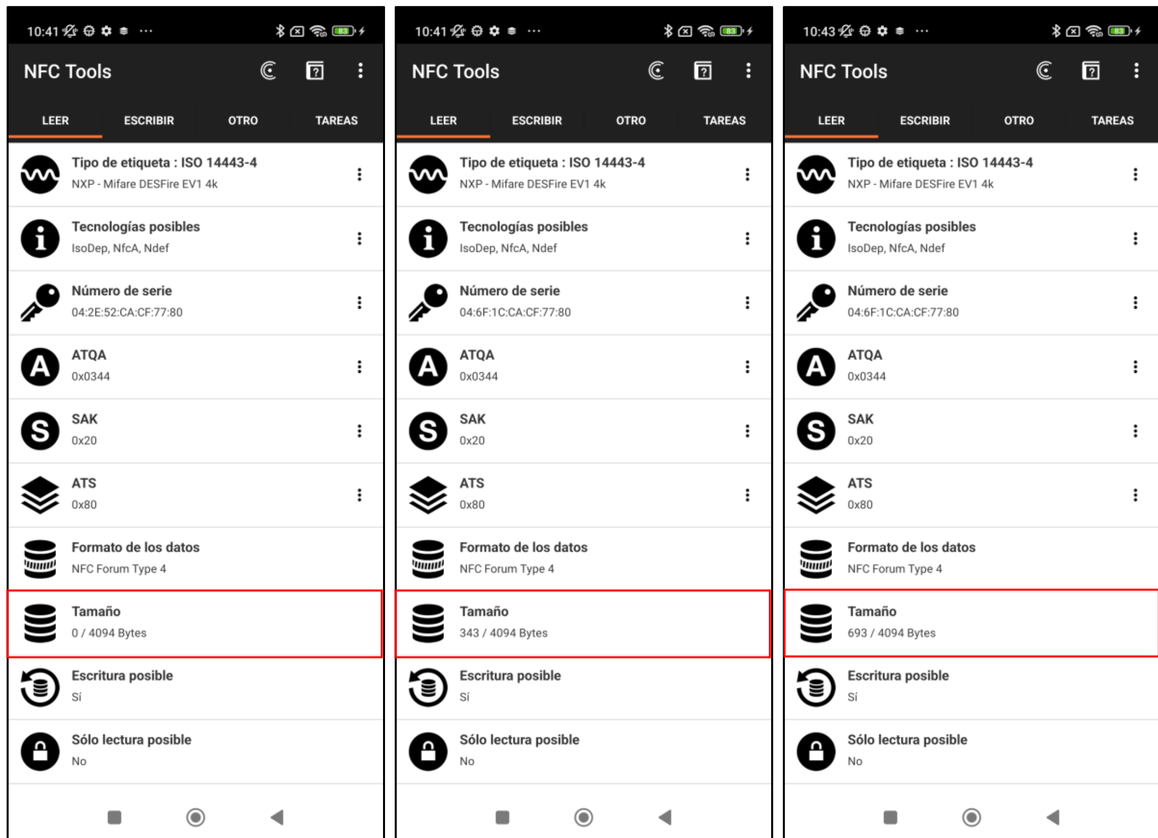


Figura 4-19. Capturas de la aplicación *NFC Tools* que muestran el uso real de la capacidad de almacenamiento del tag NFC en diferentes situaciones [fuente propia]

5 CONCLUSIONES Y LÍNEAS FUTURAS

En este capítulo se hará un resumen de las conclusiones obtenidas en base a los objetivos establecidos en el capítulo 1. Además, se plantearán posibles líneas de investigación futuras.

5.1 Conclusiones

En referencia a los objetivos marcados en el apartado 1.2 se han conseguido los mencionados a continuación:

- Se ha desarrollado una aplicación funcional para dispositivos móviles Android capaz de recopilar y trabajar con la información mínima establecida en el estándar OTAN [3]. Asimismo, se trata de una aplicación intuitiva que solventa los problemas y limitaciones identificados del uso actual de las tarjetas físicas NFMC.
- Se ha verificado la eficacia de las tarjetas NFC en la escritura y lectura de datos, lo que resulta en un ahorro significativo de tiempo, siendo este un punto crucial dado el contexto bélico en el que se encuentra enmarcado el presente trabajo. Además, dichas tarjetas NFC no se verán afectadas por los riesgos inherentes cuando se emplean formularios en papel como los existentes actualmente.
- La remisión de información de los combatientes heridos a un servidor web resulta fundamental dado el amplio abanico de posibilidades que ofrece. Entre ellas destaca la visualización en tiempo real por parte de las distintas FST de los datos modificados por el primer asistente, de tal modo que cada uno de los ROLES implicados en la evacuación de cada baja puede conocer los medios y/o tratamientos necesarios con antelación. Además, el servidor garantiza una redundancia en los datos médicos, siendo este un aspecto esencial tal y como se vio en la doctrina sanitaria en operaciones [6].

Además, se ha comprobado que *Android Studio* es una herramienta potente y versátil a la hora de desarrollar aplicaciones para móvil dada su suave curva de aprendizaje y lo intuitiva que resulta. Durante el desarrollo de este TFG se ha ido consultando la página web oficial de *Android Studio* [20] y alguna fuente más para aprender a manejar tanto la plataforma como el lenguaje de programación, el cual a su vez es también relativamente sencillo de entender y emplear.

Por otro lado, se evidencia el creciente uso y la importancia fundamental de la tecnología NFC en el mundo contemporáneo, dado que se trata de un instrumento con una creciente variedad de aplicaciones y capacidad para simplificar múltiples tareas. En el contexto específico de este TFG, como se expondrá posteriormente, ha posibilitado una mejora en la recopilación de datos de los heridos.

5.2 Líneas futuras

Una vez analizadas las diferentes conclusiones, se proponen las siguientes líneas futuras que permiten seguir ampliando y mejorando la aplicación desarrollada:

- Implementación de seguridad en dos niveles diferentes:
  - A nivel de la aplicación: de este modo se lograría una autenticación de usuarios de manera que podrían registrarse con sus propias credenciales y acceder de manera segura a la plataforma. Actualmente, la aplicación únicamente cuenta con una seguridad "local" simulada consistente en la introducción de un usuario y una contraseña en la pestaña correspondiente al inicio de sesión.
  - A nivel de encriptación de datos podemos diferenciar entre:
    - Cifrado de datos en la tarjeta NFC: para lo cual podría usarse un cifrado simétrico, donde tanto el cifrado como el descifrado se realizan utilizando la misma clave. Esta clave de cifrado se podría generar en la aplicación y luego compartirse entre los usuarios autorizados de manera segura.
    - Cifrado de las comunicaciones con el servidor: para lo que se podría emplear el protocolo HTTPS (*Hypertext Transfer Protocol Secure*), el cual cifra los datos transmitidos entre el cliente (la aplicación móvil) y el servidor, utilizando certificados SSL/TLS para autenticar el servidor y establecer una conexión segura.
    - No se contempla el cifrado de datos en la comunicación entre la aplicación y el tag NFC dado que la reducida distancia de trabajo de la tecnología NFC minimiza el riesgo de interceptación de datos durante la transferencia.

  Esto resulta fundamental dado el tipo de datos confidenciales con los que se está trabajando. Además, en muchos países, existen regulaciones estrictas que rigen la protección de la privacidad y seguridad de los datos médicos. Por ejemplo, en España es la *Ley Orgánica 3/2018, del 5 de diciembre, de Protección de Datos Personales y Garantía de los Derechos digitales* [35].

- Diseño, desarrollo y despliegue del servidor web (en este TFG se proporcionó al autor el uso de un servidor en pruebas para asegurarse de que la aplicación remitía correctamente la información). Dicho servidor podría permitir realizar el seguimiento conjunto de todos los heridos en zonas de operaciones que se desee monitorizar, empleando herramientas de visualización gráfica que faciliten la generación de la *Common Operational Picture*.

- Incorporación de una funcionalidad que habilite la alternancia del idioma de la aplicación entre diversas opciones lingüísticas. Actualmente, la aplicación se encuentra exclusivamente disponible en inglés, dado que este idioma prevalece como el más utilizado a nivel de la OTAN.

6 BIBLIOGRAFÍA

[1] B. J. Eastridge *et al.*, «*Death on the battlefield* (2001–2011): *Implications for the future of combat casualty care*», *J. Trauma Acute Care Surg.*, vol. 73, n.º 6, pp. S431-S437, dic. 2012, doi: 10.1097/TA.0b013e3182755dcc.

[2] «Trabajo de Fin de Grado. Universidad de Valladolid Grado en Enfermería. Actuaciones de la sanidad militar en el campo de batalla mediante el protocolo TCCC». Accedido: 22 de enero de 2024. [En línea]. Disponible en: https://uvadoc.uva.es/bitstream/handle/10324/54075/TFG-H2421.pdf?sequence=1

[3] «NATO Standard. AMedP-8.1. *Documentation relative to initial medical treatment and evacuation*». Accedido: 22 de enero de 2024. [En línea]. Disponible en: https://www.coemed.org/files/stanags/03_AMEDP/AMedP-8.1_EDB_V1_E_2132.pdf

[4] «Historia del CICR», Comité Internacional de la Cruz Roja. Accedido: 7 de marzo de 2024. [En línea]. Disponible en: https://www.icrc.org/es/historia-del-cicr

[5] M. M. Stadler, «Florence Nightingale, mucho más que la dama de la lámpara», Mujeres con ciencia. Accedido: 7 de marzo de 2024. [En línea]. Disponible en: https://mujeresconciencia.com/2017/08/22/florence-nightingale-mucho-mas-la-dama-la-lampara/

[6] «Resolución del EMAD sobre la Doctrina Sanitaria en Operaciones_4-10_doctrina sanitaria en operaciones_202103.pdf».

[7] «BOE-A-2005-18933 Ley Orgánica 5/2005, de 17 de noviembre, de la Defensa Nacional.» Accedido: 8 de marzo de 2024. [En línea]. Disponible en: https://www.boe.es/buscar/act.php?id=BOE-A-2005-18933

[8] «BOE-A-2017-2625 Real Decreto 230/2017, de 10 de marzo, por el que se regulan las competencias y cometidos de apoyo a la atención sanitaria del personal militar no regulado por la Ley 44/2003, de 21 de noviembre, de ordenación de las profesiones sanitarias, en el ámbito estrictamente militar.» Accedido: 8 de marzo de 2024. [En línea]. Disponible en: https://www.boe.es/buscar/act.php?id=BOE-A-2017-2625

[9] L. Hillán García, F. Setién Dodero, y A. Del Real Colomo, «El Sistema de Telemedicina Militar en España: una aproximación histórica», *Sanid. Mil.*, vol. 70, n.º 2, pp. 121-131, jun. 2014, doi: 10.4321/S1887-85712014000200010.

[10] «*Tactical combat casualty care* - TCCC», blackbeardtactical. [En línea]. Disponible en: https://blackbeard.mx/blogs/news/tactical-combat-casualty-care-tccc

[11] «BOD 6 de julio de 2023. Implantación en el ámbito del Ministerio de Defensa del STANAG 2132 MEDSTD (Edición 4) "Documentación relativa al tratamiento médico inicial y la evacuaciónAMedP-8.1, Edición B"».

[12] «Curso TCCC (*Tactical Combat Casualty Care*- Atención Táctica de Bajas de Combate) - Lurreko Armada». Accedido: 22 de enero de 2024. [En línea]. Disponible en:

https://ejercito.defensa.gob.es/eu/unidades/Melilla/comgemel/Noticias/2023/19_Curso_TCCC.html

[13] NATO, «*NATO Special Ops train to save lives*», NATO. Accedido: 19 de marzo de 2024. [En línea]. Disponible en: https://www.nato.int/cps/en/natohq/news_108261.htm

[14] «NFC Forum». Accedido: 1 de abril de 2024. [En línea]. Disponible en: https://nfc-forum.org/

[15] «Etiquetas NFC: qué son y siete usos originales que puedes darle». Accedido: 22 de enero de 2024. [En línea]. Disponible en: https://www.xataka.com/otros-dispositivos/etiquetas-nfc-que-siete-usos-originales-que-puedes-darle

[16] «¿Cuándo empezaremos a usar NFC?», El Español. Accedido: 16 de enero de 2024. [En línea]. Disponible en: https://www.elespanol.com/elandroidelibre/20130411/empezaremos-usar-nfc/19998260_0.html

[17] «NFC – ¿Qué es *Near Field Communication*?», IONOS Digital Guide. Accedido: 22 de enero de 2024. [En línea]. Disponible en: https://www.ionos.es/digitalguide/servidores/know-how/nfc-near-field-communication/

[18] Hannah Currey, «*More than 5 billion people now use the internet*», We Are Social UK. Accedido: 22 de enero de 2024. [En línea]. Disponible en: https://wearesocial.com/uk/blog/2022/04/more-than-5-billion-people-now-use-the-internet/

[19] «Descripción general de la arquitectura», Android Open Source Project. Accedido: 18 de enero de 2024. [En línea]. Disponible en: https://source.android.com/docs/core/architecture?hl=es

[20] «Introducción a Android Studio | Android Studio», Android Developers. Accedido: 18 de enero de 2024. [En línea]. Disponible en: https://developer.android.com/studio/intro?hl=es-419

[21] «*Android Project folder Structure*», GeeksforGeeks. Accedido: 20 de enero de 2024. [En línea]. Disponible en: https://www.geeksforgeeks.org/android-project-folder-structure/

[22] «Introducción a las actividades | Desarrolladores de Android», Android Developers. Accedido: 20 de enero de 2024. [En línea]. Disponible en: https://developer.android.com/guide/components/activities/intro-activities?hl=es-419

[23] «Java | Oracle». Accedido: 19 de marzo de 2024. [En línea]. Disponible en: https://www.java.com/es/

[24] «*Kotlin Programming Language*». Accedido: 19 de marzo de 2024. [En línea]. Disponible en: https://kotlinlang.org/

[25] «Descripción general de Kotlin», Android Developers. Accedido: 20 de enero de 2024. [En línea]. Disponible en: https://developer.android.com/kotlin/overview?hl=es-419

[26] «*All 101 announcements from Google I/O '17*», Google. Accedido: 24 de enero de 2024. [En línea]. Disponible en: https://blog.google/technology/developers/all-io17-announcements/

[27] «Programación en Kotlin - Infografía del estado del ecosistema del desarrollador en 2022», JetBrains: Developer Tools for Professionals and Teams. Accedido: 24 de enero de 2024. [En línea]. Disponible en: https://www.jetbrains.com/lp/devecosystem-2022

[28] J. García, «Xiaomi Redmi 10, análisis», Xataka. Accedido: 20 de enero de 2024. [En línea]. Disponible en: https://www.xataka.com/analisis/xiaomi-redmi-10-analisis-caracteristicas-precio-especificaciones

[29] «Tarjetas NFC NXP MIFARE® DESFire® EV2 2k/4k/8k», Shop NFC. Accedido: 13 de marzo de 2024. [En línea]. Disponible en: https://www.shopnfc.com/es/tarjetas-nfc/280-360-tarjetas-nfc-nxp-mifare-desfire-ev2-2k4k8k.html

[30] A. Del Real Colomo, F. Setién Dodero, A. Moreno Caravaca, y A. Hernández Abadía De Barbara, «Ayuda a la clasificación y priorización en la evacuación de bajas de combate: ayuda al proceso asistencial. Proyecto e-SafeTag», *Sanid. Mil.*, vol. 70, n.º 4, pp. 288-292, dic. 2014, doi: 10.4321/S1887-85712014000400009.

[31] «*API level | Phaisarn*». Accedido: 18 de febrero de 2024. [En línea]. Disponible en: https://phaisarn.com/2019/android-api-level/

[32] «*Mobile & Tablet Android Version Market Share Worldwide*», StatCounter Global Stats. Accedido: 13 de marzo de 2024. [En línea]. Disponible en: https://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide

[33] «Descripción general del manifiesto de la app | Android Developers». Accedido: 7 de marzo de 2024. [En línea]. Disponible en: https://developer.android.com/guide/topics/manifest/manifest-intro?hl=es-419

[34] «Tutorial de Layouts en Android». Accedido: 19 de febrero de 2024. [En línea]. Disponible en: https://www.develou.com/tutorial-layouts-en-android/

[35] «BOE-A-2018-16673 Ley Orgánica 3/2018, de 5 de diciembre, de Protección de Datos Personales y garantía de los derechos digitales.» Accedido: 27 de marzo de 2024. [En línea]. Disponible en: https://www.boe.es/buscar/act.php?id=BOE-A-2018-16673&p=20230509&tn=1

ANEXO I: IMPLICACIONES SOCIALES, ECONÓMICAS, Y AMBIENTALES

El desarrollo de una aplicación móvil destinada a sustituir las tarjetas FMC en entornos militares plantea una serie de consideraciones importantes en términos de sus implicaciones sociales, económicas y/o ambientales.

En cuanto a las implicaciones sociales, se espera que esta aplicación facilite el acceso equitativo a la atención médica dentro de las fuerzas armadas, garantizando que la información médica vital esté disponible rápidamente en situaciones críticas y en entornos remotos. Esto podría contribuir significativamente a mejorar la eficiencia del personal médico y a brindar una atención médica más efectiva, lo que, a su vez, podría fortalecer la cohesión y el bienestar del personal militar.

De cara a las posibles implicaciones económicas caben destacar los posibles costes continuos de mantenimiento y actualización de la aplicación dada la rápida evolución de la tecnología, así como la necesidad de mantener la aplicación actualizada para garantizar su compatibilidad con nuevos dispositivos, sistemas operativos y estándares de seguridad. Sin embargo, la sustitución de las tarjetas médicas físicas por una solución digital puede reducir los costes asociados con la impresión, distribución y reposición de tarjetas físicas. En este sentido, cabe destacar que, si bien la compra de los tags NFC tiene también un coste, este es bastante reducido y, además, sería posible la reutilización de un mismo tag NFC para pacientes diferentes.

Desde una perspectiva ambiental, el desarrollo y la implementación de la aplicación requerirán recursos tecnológicos, como hardware y energía eléctrica. Es esencial considerar cómo minimizar el impacto ambiental de estos recursos y adoptar prácticas sostenibles durante el ciclo de vida de la aplicación. Además, la adopción de la aplicación podría reducir la necesidad de tarjetas médicas físicas, lo que podría tener un impacto positivo en términos de reducción de residuos y consumo de papel, contribuyendo así a la conservación del medio ambiente.

En conclusión, el desarrollo de una aplicación móvil para reemplazar las tarjetas médicas de campo en entornos militares representa una oportunidad significativa para mejorar la atención médica y la eficiencia operativa en el ámbito militar. Sin embargo, es crucial abordar de manera responsable y proactiva las implicaciones sociales y ambientales asociadas con su desarrollo e implementación, garantizando que la solución propuesta sea beneficiosa, ética y sostenible a largo plazo.

ANEXO II: REFLEXIONES ÉTICAS Y SOCIALES

En el ámbito ético del desarrollo de esta aplicación, es esencial considerar diversos aspectos que pueden influir en la integridad y el respeto hacia los individuos y sus datos médicos. Una de las consideraciones más importantes es la privacidad y la confidencialidad de la información médica almacenada en la aplicación. Garantizar que los datos médicos estén protegidos contra accesos no autorizados y que se respeten los derechos de privacidad y confidencialidad de los usuarios es crucial para construir una aplicación ética y fiable.

Además, se debe prestar especial atención al consentimiento informado del paciente. Los usuarios deben comprender plenamente cómo se utilizarán sus datos médicos y otorgar su consentimiento de manera voluntaria y consciente. Es fundamental que los individuos tengan la capacidad de controlar y gestionar su información médica de acuerdo con sus preferencias y necesidades, sin presiones ni influencias externas.

La equidad y el acceso igualitario a la atención médica también son consideraciones éticas importantes en el desarrollo de esta aplicación. Es crucial asegurar que la aplicación no contribuya a la exclusión o discriminación de ciertos grupos de personas para garantizar que todos los usuarios puedan beneficiarse de sus funciones y servicios.

Un ejemplo reciente que ilustra estas implicaciones éticas es el debate en torno al uso de tecnologías de seguimiento de contactos durante la pandemia de COVID-19. Muchos países han implementado aplicaciones móviles para rastrear la propagación del virus y notificar a las personas que han estado en contacto cercano con casos confirmados de COVID-19. Si bien estas aplicaciones pueden ser útiles para contener la propagación del virus, también plantean preocupaciones éticas sobre la privacidad, la seguridad de los datos y el consentimiento informado de los usuarios. Es esencial abordar estas preocupaciones éticas de manera responsable y garantizar que se protejan los derechos individuales y se promueva la equidad y la justicia en el acceso a la atención médica.

ANEXO III: CÓDIGO DE LA APLICACIÓN DIGITAL NFMC

InicioSesion.kt

```
package NFMC

import android.content.Intent
import android.os.Bundle
import android.widget.Button
import android.widget.EditText
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import com.example.prueba1.R

class InicioSesion : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_inicio_sesion)

        val editTextDNI = findViewById<EditText>(R.id.editTextDNI)
        val editTextPassword = findViewById<EditText>(R.id.editTextPassword)
        val buttonLogin = findViewById<Button>(R.id.buttonLogin)

        buttonLogin.setOnClickListener {
            val nombre = editTextDNI.text.toString()
            val contraseña = editTextPassword.text.toString()

            if (nombre == "71903419Z" && contraseña == "CUD") {
                val intent = Intent(this@InicioSesion, MenuPrincipal::class.java)
                startActivity(intent)
            } else {
                Toast.makeText(this@InicioSesion, "Acceso denegado", Toast.LENGTH_SHORT).show()
            }
        }
    }
}
```

MenuPrincipal.kt

```kotlin
package NFMC

import android.app.PendingIntent
import android.content.Intent
import android.content.IntentFilter
import android.nfc.NdefMessage
import android.nfc.NdefRecord
import android.nfc.NfcAdapter
import android.nfc.Tag
import android.nfc.tech.Ndef
import android.os.Bundle
import android.widget.Toast
import androidx.activity.result.ActivityResult
import androidx.activity.result.contract.ActivityResultContracts
import androidx.appcompat.app.AlertDialog
import androidx.appcompat.app.AppCompatActivity
import androidx.appcompat.widget.AppCompatButton
import com.example.prueba1.R
import retrofit2.Call
import retrofit2.Callback
import retrofit2.Response
import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory
import java.io.IOException
import java.nio.charset.Charset
import java.util.Arrays


class MenuPrincipal : AppCompatActivity() {

    private lateinit var mNfcAdapter: NfcAdapter
    private var semaforo = 0 // 0 para lectura, 1 para escritura

    val variables = Variables()

    private var imageCounter_fractura = 0
    private var imageCounter_hemorragia = 0
    private var imageCounter_perforacion = 0
    private var imageCounter_no_perforacion = 0
    private var imageCounter_quemadura = 0

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_menu_principal)

        variables.registros = mutableListOf()

        //Botones de movimiento entre aplicaciones
        val boton_datos_basicos = findViewById<AppCompatButton>(R.id.boton_identificacion)
        val boton_causa = findViewById<AppCompatButton>(R.id.boton_causa)
        val boton_evaluacion = findViewById<AppCompatButton>(R.id.boton_evaluacion)
        val boton_constantes = findViewById<AppCompatButton>(R.id.boton_constantes)
        val boton_tratamiento = findViewById<AppCompatButton>(R.id.boton_tratamiento)
        val boton_movimiento = findViewById<AppCompatButton>(R.id.boton_movimiento)

        boton_datos_basicos.setOnClickListener { Navegar_basico() }
        boton_causa.setOnClickListener { Navegar_causa() }
        boton_evaluacion.setOnClickListener { Navegar_evaluacion() }
        boton_constantes.setOnClickListener { Navegar_constantes() }
        boton_tratamiento.setOnClickListener { Navegar_tratamiento() }
        boton_movimiento.setOnClickListener { Navegar_movimiento() }

        mNfcAdapter = NfcAdapter.getDefaultAdapter(this)

        //Comprobamos que el dispositivo cuente con tecnología NFC
        if (mNfcAdapter == null) {
            Toast.makeText(
                this, "This device does not have NFC technology.", Toast.LENGTH_LONG
            ).show()
        } else {
            //Comprobamos que la antena NFC este encendida
            if (!mNfcAdapter.isEnabled) {
                Toast.makeText(this, "NFC antenna turned off", Toast.LENGTH_LONG).show()
            }
        }

        val boton_escribir_nfc = findViewById<AppCompatButton>(R.id.boton_escribir_nfc)
        val boton_lectura = findViewById<AppCompatButton>(R.id.leer_nfc)

        // Asignar OnClickListener para el botón de lectura
        boton_lectura.setOnClickListener {
            semaforo = 0 // Establecer el semáforo en 0 para lectura
            mostrarMensaje("Read mode enabled.")
        }

        // Asignar OnClickListener para el botón de escritura
        boton_escribir_nfc.setOnClickListener {
            semaforo = 1 // Establecer el semáforo en 1 para escritura
            mostrarMensaje("Write mode enabled.")
        }


    }

    override fun onResume() {
        super.onResume()
        // Habilitar la detección de NFC
        val intent = Intent(this, MenuPrincipal::class.java)
        intent.addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP)
        val pendingIntent = PendingIntent.getActivity(this, 0, intent, PendingIntent.FLAG_MUTABLE)
        val filters = arrayOf(IntentFilter(NfcAdapter.ACTION_TAG_DISCOVERED))
```

```
        mNfcAdapter.enableForegroundDispatch(this, pendingIntent, filters, null)
    }

    override fun onPause() {
        super.onPause()
        // Deshabilitar la detección de NFC
        mNfcAdapter.disableForegroundDispatch(this)
    }

    override fun onNewIntent(intent: Intent?) {
        super.onNewIntent(intent)

        when (semaforo) {
            0 -> {
                // Lectura de la tarjeta NFC
                val tag = intent?.getParcelableExtra<Tag>(NfcAdapter.EXTRA_TAG)
                val message = leerNFC(tag)
                mostrarMensaje("NFC tag content: $message")
            }

            1 -> {
                // Escritura en la tarjeta NFC
                val tag = intent?.getParcelableExtra<Tag>(NfcAdapter.EXTRA_TAG)
                val mensajeCompleto = construirMensajeNFC()
                val numeroIdDeseado = variables.numeroId
                escribirNFC(tag, mensajeCompleto, numeroIdDeseado)
            }

            else -> return // Manejar otros casos de semáforo si es necesario
        }
    }

    // Método para leer el contenido de la tarjeta NFC
    private fun leerNFC(tag: Tag?): String {

        val ndef = Ndef.get(tag)
        ndef?.connect()
        val message = ndef?.ndefMessage
        val stringBuilder = StringBuilder()

        if (message != null) {
            val records = message.records
            if (records.isNotEmpty()) {
                for (record in records) {
                    if (record.tnf == NdefRecord.TNF_WELL_KNOWN && Arrays.equals(
                                record.type,
                                NdefRecord.RTD_TEXT
                            )
                    ) {
                        val payload = record.payload
                        val encoding = if ((payload[0].toInt() and 128) == 0) "UTF-8" else "UTF-16"
                        val languageCodeLength = payload[0].toInt() and 51
                        val text = String(
                            payload,
                            languageCodeLength + 1,
                            payload.size - languageCodeLength - 1,
                            Charset.forName(encoding)
                        )
                        stringBuilder.append(text)
                    }
                }
            } else {
                ndef?.close()
                return "No content found on the NFC tag."
            }
        } else {
            ndef?.close()
            return "Failed to read message from the NFC tag."
        }

        clear_listas()
        variables.registros.clear()

        val mensajeCompleto = stringBuilder.toString()
        val mensajesSeparados = mensajeCompleto.split("\n")
        if (mensajesSeparados.size >= 2) {
            //Datos Básicos
            variables.nombre = mensajesSeparados[0]
            variables.apellidos = mensajesSeparados[1]
            variables.sexo = mensajesSeparados[2]
            variables.numeroId = mensajesSeparados[3]
            variables.fechaNacimiento = mensajesSeparados[4]
            variables.lugarOrigen = mensajesSeparados[5]
            variables.fuerzasArmadasOrigen = mensajesSeparados[6]

            // Evaluación
            variables.fecha = mensajesSeparados[7]
            variables.hora = mensajesSeparados[8]
            variables.fechaEvaluacion = mensajesSeparados[9]
            variables.horaEvaluacion = mensajesSeparados[10]
            variables.senales = mensajesSeparados[11]
            variables.alergias = mensajesSeparados[12]
            variables.sangrado = mensajesSeparados[13]
            variables.airway = mensajesSeparados[14]
            variables.respiracion = mensajesSeparados[15]
            variables.circulacion = mensajesSeparados[16]
            variables.head = mensajesSeparados[17]

            // Constantes Vitales
            variables.hora_constantes = mensajesSeparados[18]
            variables.pulso = mensajesSeparados[19]
            variables.presion_sanguinea = mensajesSeparados[20]
            variables.tasa_respiracion = mensajesSeparados[21]
            variables.saturacion_oxigeno = mensajesSeparados[22]
```

```
variables.escala_dolor = mensajesSeparados[23]
variables.estado_herido = mensajesSeparados[24]

// Tratamiento
variables.nombre_t = mensajesSeparados[25]
variables.volumen = mensajesSeparados[26]
variables.via = mensajesSeparados[27]
variables.hora_t = mensajesSeparados[28]
variables.nombre_productos_sanguineos = mensajesSeparados[29]
variables.volumen_productos_sanguineos = mensajesSeparados[30]
variables.via_productos_sanguineos = mensajesSeparados[31]
variables.hora_productos_sanguineos = mensajesSeparados[32]
variables.nombreAnalgesia = mensajesSeparados[33]
variables.dosisAnalgesia = mensajesSeparados[34]
variables.viaAnalgesia = mensajesSeparados[35]
variables.horaAnalgesia = mensajesSeparados[36]
variables.nombreAntibioticos = mensajesSeparados[37]
variables.dosisAntibioticos = mensajesSeparados[38]
variables.viaAntibioticos = mensajesSeparados[39]
variables.horaAntibioticos = mensajesSeparados[40]
variables.nombreOtros = mensajesSeparados[41]
variables.dosisOtros = mensajesSeparados[42]
variables.viaOtros = mensajesSeparados[43]
variables.horaOtros = mensajesSeparados[44]
variables.ubicacionTorniquete = mensajesSeparados[45]
variables.horaTorniquete = mensajesSeparados[46]
variables.hipotermia = mensajesSeparados[47]

// Movimiento
variables.categoria = mensajesSeparados[48]
variables.nombreSocorrista = mensajesSeparados[49]
variables.apellidosSocorrista = mensajesSeparados[50]
variables.IDsocorrista = mensajesSeparados[51]
variables.datosAdicionales = mensajesSeparados[52]

// Causa
// Si hay datos para fractura
if (mensajesSeparados.size >= 54) {
    val coordenadasFractura = mensajesSeparados[53].split(";")
    for (coordenada in coordenadasFractura) {
        val partes = coordenada.split(",")
        if (partes.size == 2) {
            val x = partes[0].toFloat()
            val y = partes[1].toFloat()
            variables.positionsList_fractura.add(x to y)
        }
    }
    // Si hay datos para hemorragia
    if (mensajesSeparados.size >= 55) {
        val coordenadasHemorragia = mensajesSeparados[54].split(";")
        for (coordenada in coordenadasHemorragia) {
            val partes = coordenada.split(",")
            if (partes.size == 2) {
                val x = partes[0].toFloat()
                val y = partes[1].toFloat()
                variables.positionsList_hemorragia.add(x to y)
            }
        }
    }

    // Si hay datos para perforación
    if (mensajesSeparados.size >= 56) {
        val coordenadasPerforacion = mensajesSeparados[55].split(";")
        for (coordenada in coordenadasPerforacion) {
            val partes = coordenada.split(",")
            if (partes.size == 2) {
                val x = partes[0].toFloat()
                val y = partes[1].toFloat()
                variables.positionsList_perforacion.add(x to y)
            }
        }
    }

    // Si hay datos para no perforación
    if (mensajesSeparados.size >= 57) {
        val coordenadasNoPerforacion = mensajesSeparados[56].split(";")
        for (coordenada in coordenadasNoPerforacion) {
            val partes = coordenada.split(",")
            if (partes.size == 2) {
                val x = partes[0].toFloat()
                val y = partes[1].toFloat()
                variables.positionsList_no_perforacion.add(x to y)
            }
        }
    }

    // Si hay datos para quemadura
    if (mensajesSeparados.size >= 58) {
        val coordenadasQuemadura = mensajesSeparados[57].split(";")
        for (coordenada in coordenadasQuemadura) {
            val partes = coordenada.split(",")
            if (partes.size == 2) {
                val x = partes[0].toFloat()
                val y = partes[1].toFloat()
                variables.positionsList_quemadura.add(x to y)
            }

        }
    }
}
//HASTA AQUI HAY 59 POSICIONES, ES DECIR 58 ELEMENTOS (DEL 0 AL 58)
// Registros
val startIndex = 58 // Ajusta esto según la cantidad de datos básicos y coordenadas
if (mensajesSeparados.size >= startIndex + 7) { // Ajusta esto según la cantidad de campos en un registro
```

```
            // Limpiar registros anteriores si es necesario
            variables.registros.clear()

            // Iterar sobre los registros en el mensaje NFC
            var index = startIndex

            while (index < mensajesSeparados.size - 6) { // Ajusta esto según la cantidad de campos en un registro
                val hora_constantes = mensajesSeparados[index]
                val pulso = mensajesSeparados[index + 1]
                val presionSanguinea = mensajesSeparados[index + 2]
                val tasaRespiracion = mensajesSeparados[index + 3]
                val saturacionOxigeno = mensajesSeparados[index + 4]
                val estadoHerido = mensajesSeparados[index + 5]
                val escalaDolor = mensajesSeparados[index + 6]

                // Crear un nuevo registro y agregarlo a la lista
                variables.registros.add(
                    Registro(
                        hora_constantes,
                        pulso,
                        presionSanguinea,
                        tasaRespiracion,
                        saturacionOxigeno,
                        estadoHerido,
                        escalaDolor

                    )
                )

                // Incrementar el índice para el próximo registro
                index += 7 // Ajusta esto según la cantidad de campos en un registro
            }
        }

    }
    ndef?.close()
    return if (stringBuilder.isNotEmpty()) stringBuilder.toString() else "No content found on the NFC tag."
}

private fun enviarServidor() {
    //PARTE DEL SERVIDOR
    val retrofit = this.getRetrofit()
    val servicioAPI = retrofit.create(APIService::class.java)
    //Toast.makeText(this@MenuPrincipal,this.variables.IDsocorrista, Toast.LENGTH_SHORT).show()
    //return
    servicioAPI.crearRegistro(this.variables).enqueue(object : Callback<Void> {
        override fun onResponse(call: Call<Void>, response: Response<Void>) {

            if (response.isSuccessful) {
                // La llamada fue exitosa, se ha creado el usuario
                Toast.makeText(
                    this@MenuPrincipal,
                    "Successful connection to the server.",
                    Toast.LENGTH_SHORT
                )
                    .show()
            } else {
                // La llamada falló por algún motivo
                Toast.makeText(this@MenuPrincipal, "Error connecting to the server.", Toast.LENGTH_SHORT)
                    .show()
            }
        }

        override fun onFailure(call: Call<Void>, t: Throwable) {
            // La llamada falló debido a un error de red u otro tipo de error
            Toast.makeText(this@MenuPrincipal, "Error connecting to the server.", Toast.LENGTH_SHORT).show()
        }
    })
}

private fun escribirNFC(tag: Tag?, message: String, numeroIdDeseado: String) {
    this.enviarServidor()
    val ndef = Ndef.get(tag)
    ndef?.connect()
    try {
        val ndefMessage = ndef?.ndefMessage
        val stringBuilder = StringBuilder()

        if (ndefMessage != null) {
            val records = ndefMessage.records
            if (records.isNotEmpty()) {
                for (record in records) {
                    if (record.tnf == NdefRecord.TNF_WELL_KNOWN && Arrays.equals(
                            record.type,
                            NdefRecord.RTD_TEXT
                        )
                    ) {
                        val payload = record.payload
                        val encoding =
                            if ((payload[0].toInt() and 128) == 0) "UTF-8" else "UTF-16"
                        val languageCodeLength = payload[0].toInt() and 51
                        val text = String(
                            payload,
                            languageCodeLength + 1,
                            payload.size - languageCodeLength - 1,
                            Charset.forName(encoding)
                        )
                        stringBuilder.append(text)
                    }
                }
            } else {
                mostrarMensaje("No content was found on the NFC tag")
                return
            }
```

```
        } else {
            mostrarMensaje("The message on the NFC tag could not be read")
            return
        }

        val mensajeCompleto = stringBuilder.toString()
        val mensajesSeparados = mensajeCompleto.split("\n")
        if (mensajesSeparados.size >= 4) { // Suponiendo que el numeroId está en la cuarta posición
            val numeroIdTag = mensajesSeparados[3] // Obtener el numeroId del tag NFC

            if (numeroIdTag == numeroIdDeseado) { // Comparar con el numeroId deseado
                val ndefRecord = NdefRecord.createTextRecord(null, message)
                val newNdefMessage = NdefMessage(arrayOf(ndefRecord))
                ndef?.writeNdefMessage(newNdefMessage)
                mostrarMensaje("Message written on the NFC tag.")
            } else {
                mostrarDialogoConfirmacion(ndef, message)
            }
        } else {
            mostrarMensaje("Failed to extract the number ID from the NFC tag message.")
        }
    } catch (e: IOException) {
        mostrarMensaje("Error writing to the NFC tag: ${e.message}")
    } finally {
        ndef?.close()
    }
}

private fun construirMensajeNFC(): String {
    val stringBuilder = StringBuilder()

    // Incluir datos básicos
    stringBuilder.append(
        "${variables.nombre}\n${variables.apellidos}\n${variables.sexo}\n${variables.numeroId}\n${variables.fechaNacimiento}\n${variables.lugarOri
                "${variables.fecha}\n${variables.hora}\n${variables.fechaEvaluacion}\n${variables.horaEvaluacion}\n${variables.senales}\n${variabl
                "${variables.hora_constantes}\n${variables.pulso}\n${variables.presion_sanguinea}\n${variables.tasa_respiracion}\n${variables.satu
                "${variables.nombre_t}\n${variables.volumen}\n${variables.via}\n${variables.hora_t}\n${variables.nombre_productos_sanguineos}\n${v
                "${variables.via_productos_sanguineos}\n${variables.hora_productos_sanguineos}\n${variables.nombreAnalgesia}\n${variables.dosisAna
                "${variables.viaAnalgesia}\n${variables.horaAnalgesia}\n${variables.nombreAntibioticos}\n${variables.dosisAntibioticos}\n${variabl
                "${variables.horaAntibioticos}\n${variables.nombreOtros}\n${variables.dosisOtros}\n${variables.viaOtros}\n${variables.horaOtros}\n
                "${variables.horaTorniquete}\n${variables.hipotermia}\n" +
                "${variables.categoria}\n${variables.nombreSocorrista}\n${variables.apellidosSocorrista}\n${variables.IDsocorrista}\n${variables.d
    )

    stringBuilder.append("\n") // Agrega un salto de línea antes de comenzar las coordenadas de fractura
    // Agregar las coordenadas de positionsList_fractura
    for ((index, pair) in variables.positionsList_fractura.withIndex()) {
        val (x, y) = pair
        stringBuilder.append("$x,$y") // Agrega las coordenadas x y y separadas por una coma
        if (index < variables.positionsList_fractura.size - 1) {
            stringBuilder.append(";") // Agrega un punto y coma si no es el último par de coordenadas
        }
    }
    // Agregar las coordenadas de positionsList_hemorragia
    stringBuilder.append("\n") // Agrega un salto de línea antes de comenzar las coordenadas de hemorragia
    for ((index, pair) in variables.positionsList_hemorragia.withIndex()) {
        val (x, y) = pair
        stringBuilder.append("$x,$y") // Agrega las coordenadas x y y separadas por una coma
        if (index < variables.positionsList_hemorragia.size - 1) {
            stringBuilder.append(";") // Agrega un punto y coma si no es el último par de coordenadas
        }
    }

    // Agregar las coordenadas de positionsList_perforacion
    stringBuilder.append("\n") // Agrega un salto de línea antes de comenzar las coordenadas de hemorragia
    for ((index, pair) in variables.positionsList_perforacion.withIndex()) {
        val (x, y) = pair
        stringBuilder.append("$x,$y") // Agrega las coordenadas x y y separadas por una coma
        if (index < variables.positionsList_perforacion.size - 1) {
            stringBuilder.append(";") // Agrega un punto y coma si no es el último par de coordenadas
        }
    }

    // Agregar las coordenadas de positionsList_no_perforacion
    stringBuilder.append("\n") // Agrega un salto de línea antes de comenzar las coordenadas de hemorragia
    for ((index, pair) in variables.positionsList_no_perforacion.withIndex()) {
        val (x, y) = pair
        stringBuilder.append("$x,$y") // Agrega las coordenadas x y y separadas por una coma
        if (index < variables.positionsList_no_perforacion.size - 1) {
            stringBuilder.append(";") // Agrega un punto y coma si no es el último par de coordenadas
        }
    }

    // Agregar las coordenadas de positionsList_quemadura
    stringBuilder.append("\n") // Agrega un salto de línea antes de comenzar las coordenadas de hemorragia
    for ((index, pair) in variables.positionsList_quemadura.withIndex()) {
        val (x, y) = pair
        stringBuilder.append("$x,$y") // Agrega las coordenadas x y y separadas por una coma
        if (index < variables.positionsList_quemadura.size - 1) {
            stringBuilder.append(";") // Agrega un punto y coma si no es el último par de coordenadas
        }
    }

    stringBuilder.append("\n") // Agrega un salto de línea antes de comenzar
    // Agregar los datos del registro
    if (variables.registros != null) {
        for (registro in variables.registros) {
            stringBuilder.append(
                "${registro.hora_constantes}\n${registro.pulso}\n${registro.presionSanguinea}\n" +
                        "${registro.tasaRespiracion}\n${registro.saturacionOxigeno}\n" +
                        "${registro.estadoHerido}\n${registro.escalaDolor}\n"
            )
        }
    }
```

```
    return stringBuilder.toString()
}

fun Navegar_basico() {
    //pasamos datos al edit text de la pestaña de datos basicos
    val intent_datosBasicos = Intent(this, DatosBasicos::class.java)
    intent_datosBasicos.putExtra("nombre_vuelta", variables.nombre)
    intent_datosBasicos.putExtra("apellidos_vuelta", variables.apellidos)
    intent_datosBasicos.putExtra("sexo_vuelta", variables.sexo)
    intent_datosBasicos.putExtra("Id_vuelta", variables.numeroId)
    intent_datosBasicos.putExtra("nacimiento_vuelta", variables.fechaNacimiento)
    intent_datosBasicos.putExtra("origen_vuelta", variables.lugarOrigen)
    intent_datosBasicos.putExtra("fuerzasArmadas_vuelta", variables.fuerzasArmadasOrigen)
    getResultDatosBasicos.launch(intent_datosBasicos)
}

private val getResultDatosBasicos =
    registerForActivityResult(
        ActivityResultContracts.StartActivityForResult()
    ) { result: ActivityResult ->
        if (result.resultCode == RESULT_OK) {
            val data = result.data
            variables.nombre = data?.getStringExtra("nombre") ?: ""
            variables.apellidos = data?.getStringExtra("apellidos") ?: ""
            variables.sexo = data?.getStringExtra("sexo") ?: ""
            variables.numeroId = data?.getStringExtra("numeroID") ?: ""
            variables.fechaNacimiento = data?.getStringExtra("fechaNacimiento") ?: ""
            variables.lugarOrigen = data?.getStringExtra("lugarOrigen") ?: ""
            variables.fuerzasArmadasOrigen = data?.getStringExtra("fuerzasArmadasOrigen") ?: ""
        }
    }

fun Navegar_causa() {
    val intent_causa = Intent(this, Causa::class.java)
    intent_causa.putExtra("positionsList_fractura_vuelta", variables.positionsList_fractura)
    intent_causa.putExtra("positionsList_hemorragia_vuelta", variables.positionsList_hemorragia)
    intent_causa.putExtra(
        "positionsList_perforacion_vuelta",
        variables.positionsList_perforacion
    )
    intent_causa.putExtra(
        "positionsList_no_perforacion_vuelta",
        variables.positionsList_no_perforacion
    )
    intent_causa.putExtra("positionsList_quemadura_vuelta", variables.positionsList_quemadura)
    intent_causa.putExtra("imageCounter_fractura_vuelta", this.imageCounter_fractura)
    intent_causa.putExtra("imageCounter_hemorragia_vuelta", this.imageCounter_hemorragia)
    intent_causa.putExtra("imageCounter_perforacion_vuelta", this.imageCounter_perforacion)
    intent_causa.putExtra(
        "imageCounter_no_perforacion_vuelta",
        this.imageCounter_no_perforacion
    )
    intent_causa.putExtra("imageCounter_quemadura_vuelta", this.imageCounter_quemadura)
    getResultCausa.launch(intent_causa)
}

private val getResultCausa =
    registerForActivityResult(
        ActivityResultContracts.StartActivityForResult()
    ) { result: ActivityResult ->
        if (result.resultCode == RESULT_OK) {
            val data = result.data
            // Extraer los valores de solOriginal.x y solOriginal.y como Float
            variables.positionsList_fractura =
                (data?.getSerializableExtra("positionsList_fractura") as? ArrayList<Pair<Float, Float>>)!!
            if (variables.positionsList_fractura != null) {
                // Recorrer la lista de coordenadas y mostrarlas
                for ((index, pair) in variables.positionsList_fractura.withIndex()) {
                    val (x, y) = pair
                    //toast("Imagen $index  X: $x, Y: $y")
                }
            }
            variables.positionsList_hemorragia =
                (data?.getSerializableExtra("positionsList_hemorragia") as? ArrayList<Pair<Float, Float>>)!!
            if (variables.positionsList_hemorragia != null) {
                // Recorrer la lista de coordenadas y mostrarlas
                for ((index, pair) in variables.positionsList_hemorragia.withIndex()) {
                    val (x, y) = pair
                    //toast("Imagen $index  X: $x, Y: $y")
                }
            }
            variables.positionsList_perforacion =
                (data?.getSerializableExtra("positionsList_perforacion") as? ArrayList<Pair<Float, Float>>)!!
            if (variables.positionsList_perforacion != null) {
                // Recorrer la lista de coordenadas y mostrarlas
                for ((index, pair) in variables.positionsList_perforacion.withIndex()) {
                    val (x, y) = pair
                    //toast("Imagen $index  X: $x, Y: $y")
                }
            }
            variables.positionsList_no_perforacion =
                (data?.getSerializableExtra("positionsList_no_perforacion") as? ArrayList<Pair<Float, Float>>)!!
            if (variables.positionsList_no_perforacion != null) {
                // Recorrer la lista de coordenadas y mostrarlas
                for ((index, pair) in variables.positionsList_no_perforacion.withIndex()) {
                    val (x, y) = pair
                    //toast("Imagen $index  X: $x, Y: $y")
                }
            }
            variables.positionsList_quemadura =
                (data?.getSerializableExtra("positionsList_quemadura") as? ArrayList<Pair<Float, Float>>)!!
            if (variables.positionsList_quemadura != null) {
                // Recorrer la lista de coordenadas y mostrarlas
                for ((index, pair) in variables.positionsList_quemadura.withIndex()) {
```

```kotlin
                    val (x, y) = pair
                    //toast("Imagen $index  X: $x, Y: $y")
                }
            }

            this.imageCounter_fractura = data?.getIntExtra("imageCounter_fractura", 0) ?: 0
            this.imageCounter_hemorragia = data?.getIntExtra("imageCounter_hemorragia", 0) ?: 0
            this.imageCounter_perforacion =
                data?.getIntExtra("imageCounter_perforacion", 0) ?: 0
            this.imageCounter_no_perforacion =
                data?.getIntExtra("imageCounter_no_perforacion", 0) ?: 0
            this.imageCounter_quemadura = data?.getIntExtra("imageCounter_quemadura", 0) ?: 0
        }
    }

fun Navegar_evaluacion() {
    //pasamos datos al edit text de la pestaña de evaluacion
    val intent_evaluacion = Intent(this, Evaluacion::class.java)
    intent_evaluacion.putExtra("fecha_vuelta", variables.fecha)
    intent_evaluacion.putExtra("hora_vuelta", variables.hora)
    intent_evaluacion.putExtra("fechaEvaluacion_vuelta", variables.fechaEvaluacion)
    intent_evaluacion.putExtra("horaEvaluacion_vuelta", variables.horaEvaluacion)
    intent_evaluacion.putExtra("senales_vuelta", variables.senales)
    intent_evaluacion.putExtra("alergias_vuelta", variables.alergias)
    intent_evaluacion.putExtra("sangrado_vuelta", variables.sangrado)
    intent_evaluacion.putExtra("airway_vuelta", variables.airway)
    intent_evaluacion.putExtra("respiracion_vuelta", variables.respiracion)
    intent_evaluacion.putExtra("circulacion_vuelta", variables.circulacion)
    intent_evaluacion.putExtra("head_vuelta", variables.head)


    getResultEvaluacion.launch(intent_evaluacion)
}

private val getResultEvaluacion =
    registerForActivityResult(
        ActivityResultContracts.StartActivityForResult()
    ) { result: ActivityResult ->
        if (result.resultCode == RESULT_OK) {
            val data = result.data
            variables.fecha = data?.getStringExtra("fecha") ?: ""
            variables.hora = data?.getStringExtra("hora") ?: ""
            variables.fechaEvaluacion = data?.getStringExtra("fechaEvaluacion") ?: ""
            variables.horaEvaluacion = data?.getStringExtra("horaEvaluacion") ?: ""
            variables.senales = data?.getStringExtra("senales") ?: ""
            variables.alergias = data?.getStringExtra("alergias") ?: ""
            variables.sangrado = data?.getStringExtra("sangrado") ?: ""
            variables.airway = data?.getStringExtra("airway") ?: ""
            variables.respiracion = data?.getStringExtra("respiracion") ?: ""
            variables.circulacion = data?.getStringExtra("circulacion") ?: ""
            variables.head = data?.getStringExtra("head") ?: ""

        }
    }

fun Navegar_constantes() {
    val intent_constantes = Intent(this, ConstantesVitales::class.java)
    intent_constantes.putExtra("hora_constantes_vuelta", variables.hora_constantes)
    intent_constantes.putExtra("pulso_vuelta", variables.pulso)
    intent_constantes.putExtra("presion_sanguinea_vuelta", variables.presion_sanguinea)
    intent_constantes.putExtra("tasa_respiracion_vuelta", variables.tasa_respiracion)
    intent_constantes.putExtra("saturacion_oxigeno_vuelta", variables.saturacion_oxigeno)
    intent_constantes.putExtra("estado_herido_vuelta", variables.estado_herido)
    intent_constantes.putExtra("escala_dolor_vuelta", variables.escala_dolor)

    // Verificar si la lista de registros no es nula y luego agregarla al intent
    if (variables.registros != null) {
        intent_constantes.putParcelableArrayListExtra(
            "registros_vuelta",
            ArrayList(variables.registros)
        )
    }
    getResultConstantes.launch(intent_constantes)
}

private val getResultConstantes = registerForActivityResult(
    ActivityResultContracts.StartActivityForResult()
) { result: ActivityResult ->
    if (result.resultCode == RESULT_OK) {
        val data = result.data
        variables.hora_constantes = data?.getStringExtra("hora_constantes") ?: ""
        variables.pulso = data?.getStringExtra("pulso") ?: ""
        variables.presion_sanguinea = data?.getStringExtra("presion_sanguinea") ?: ""
        variables.tasa_respiracion = data?.getStringExtra("tasa_respiracion") ?: ""
        variables.saturacion_oxigeno = data?.getStringExtra("saturacion_oxigeno") ?: ""
        variables.estado_herido = data?.getStringExtra("estado_herido") ?: ""
        variables.escala_dolor = data?.getStringExtra("escala_dolor") ?: ""


        val registrosArray = data?.getParcelableArrayExtra("registros")

        // Convertir el array de registros en una lista mutable
        val registrosList = mutableListOf<Registro>()
        registrosArray?.forEach { registro ->
            if (registro is Registro) {
                registrosList.add(registro)
            }
        }
        // Asignar la lista mutable de registros
        variables.registros = registrosList
    }
}

fun Navegar_tratamiento() {
```

```kotlin
    // Pasamos datos al EditText de la pestaña de tratamiento
    val intent_tratamiento = Intent(this, Tratamiento::class.java)
    intent_tratamiento.putExtra("nombre_vuelta", variables.nombre_t)
    intent_tratamiento.putExtra("volumen_vuelta", variables.volumen)
    intent_tratamiento.putExtra("via_vuelta", variables.via)
    intent_tratamiento.putExtra("hora_vuelta", variables.hora_t)

    intent_tratamiento.putExtra(
        "nombre_productos_sanguineos_vuelta",
        variables.nombre_productos_sanguineos
    )
    intent_tratamiento.putExtra(
        "volumen_productos_sanguineos_vuelta",
        variables.volumen_productos_sanguineos
    )
    intent_tratamiento.putExtra(
        "via_productos_sanguineos_vuelta",
        variables.via_productos_sanguineos
    )
    intent_tratamiento.putExtra(
        "hora_productos_sanguineos_vuelta",
        variables.hora_productos_sanguineos
    )

    intent_tratamiento.putExtra("nombreAnalgesia_vuelta", variables.nombreAnalgesia)
    intent_tratamiento.putExtra("dosisAnalgesia_vuelta", variables.dosisAnalgesia)
    intent_tratamiento.putExtra("viaAnalgesia_vuelta", variables.viaAnalgesia)
    intent_tratamiento.putExtra("horaAnalgesia_vuelta", variables.horaAnalgesia)

    intent_tratamiento.putExtra("nombreAntibioticos_vuelta", variables.nombreAntibioticos)
    intent_tratamiento.putExtra("dosisAntibioticos_vuelta", variables.dosisAntibioticos)
    intent_tratamiento.putExtra("viaAntibioticos_vuelta", variables.viaAntibioticos)
    intent_tratamiento.putExtra("horaAntibioticos_vuelta", variables.horaAntibioticos)

    intent_tratamiento.putExtra("nombreOtros_vuelta", variables.nombreOtros)
    intent_tratamiento.putExtra("dosisOtros_vuelta", variables.dosisOtros)
    intent_tratamiento.putExtra("viaOtros_vuelta", variables.viaOtros)
    intent_tratamiento.putExtra("horaOtros_vuelta", variables.horaOtros)

    intent_tratamiento.putExtra("ubicacionTorniquete_vuelta", variables.ubicacionTorniquete)
    intent_tratamiento.putExtra("horaTorniquete_vuelta", variables.horaTorniquete)
    intent_tratamiento.putExtra("hipotermia_vuelta", variables.hipotermia)

    getResultTratamiento.launch(intent_tratamiento)
}


// Manejo del resultado de la actividad de tratamiento
private val getResultTratamiento = registerForActivityResult(
    ActivityResultContracts.StartActivityForResult()
) { result: ActivityResult ->
    if (result.resultCode == RESULT_OK) {
        val data = result.data
        variables.nombre_t = data?.getStringExtra("nombre") ?: ""
        variables.volumen = data?.getStringExtra("volumen") ?: ""
        variables.via = data?.getStringExtra("via") ?: ""
        variables.hora_t = data?.getStringExtra("hora") ?: ""

        variables.nombre_productos_sanguineos =
            data?.getStringExtra("nombre_productos_sanguineos") ?: ""
        variables.volumen_productos_sanguineos =
            data?.getStringExtra("volumen_productos_sanguineos") ?: ""
        variables.via_productos_sanguineos =
            data?.getStringExtra("via_productos_sanguineos") ?: ""
        variables.hora_productos_sanguineos =
            data?.getStringExtra("hora_productos_sanguineos") ?: ""

        variables.nombreAnalgesia = data?.getStringExtra("nombreAnalgesia") ?: ""
        variables.dosisAnalgesia = data?.getStringExtra("dosisAnalgesia") ?: ""
        variables.viaAnalgesia = data?.getStringExtra("viaAnalgesia") ?: ""
        variables.horaAnalgesia = data?.getStringExtra("horaAnalgesia") ?: ""

        variables.nombreAntibioticos = data?.getStringExtra("nombreAntibioticos") ?: ""
        variables.dosisAntibioticos = data?.getStringExtra("dosisAntibioticos") ?: ""
        variables.viaAntibioticos = data?.getStringExtra("viaAntibioticos") ?: ""
        variables.horaAntibioticos = data?.getStringExtra("horaAntibioticos") ?: ""

        variables.nombreOtros = data?.getStringExtra("nombreOtros") ?: ""
        variables.dosisOtros = data?.getStringExtra("dosisOtros") ?: ""
        variables.viaOtros = data?.getStringExtra("viaOtros") ?: ""
        variables.horaOtros = data?.getStringExtra("horaOtros") ?: ""

        variables.ubicacionTorniquete = data?.getStringExtra("ubicacionTorniquete") ?: ""
        variables.horaTorniquete = data?.getStringExtra("horaTorniquete") ?: ""
        variables.hipotermia = data?.getStringExtra("hipotermia") ?: ""


    }
}

fun Navegar_movimiento() {
    //pasamos datos al edit text de la pestaña de evaluacion
    val intent_movimiento = Intent(this, Movimiento::class.java)
    intent_movimiento.putExtra("categoria_vuelta", variables.categoria)
    intent_movimiento.putExtra("nombreSocorrista_vuelta", variables.nombreSocorrista)
    intent_movimiento.putExtra("apellidosSocorrista_vuelta", variables.apellidosSocorrista)
    intent_movimiento.putExtra("IDsocorrista_vuelta", variables.IDsocorrista)
    intent_movimiento.putExtra("datosAdicionales_vuelta", variables.datosAdicionales)
    getResultMovimiento.launch(intent_movimiento)
}

private val getResultMovimiento =
    registerForActivityResult(
        ActivityResultContracts.StartActivityForResult()
```

```
        ) { result: ActivityResult ->
            if (result.resultCode == RESULT_OK) {
                val data = result.data
                variables.categoria = data?.getStringExtra("categoria") ?: ""
                variables.nombreSocorrista = data?.getStringExtra("nombreSocorrista") ?: ""
                variables.apellidosSocorrista = data?.getStringExtra("apellidosSocorrista") ?: ""
                variables.IDsocorrista = data?.getStringExtra("IDsocorrista") ?: ""
                variables.datosAdicionales = data?.getStringExtra("datosAdicionales") ?: ""
            }
        }

    private fun mostrarDialogoConfirmacion(ndef: Ndef?, message: String) {
        val builder = AlertDialog.Builder(this)
        builder.setTitle("Confirmation")
        builder.setMessage("The number ID does not match. Are you sure you want to write the data anyway?")
        builder.setPositiveButton("Yes (Keep the TAG close).") { dialog, which ->
            try {
                ndef?.connect() // Conectar antes de escribir el mensaje
                val ndefRecord = NdefRecord.createTextRecord(null, message)
                val newNdefMessage = NdefMessage(arrayOf(ndefRecord))
                ndef?.writeNdefMessage(newNdefMessage)
                mostrarMensaje("Message written on the NFC tag.")
            } catch (e: IOException) {
                mostrarMensaje("Error writing to the NFC tag: ${e.message}")
            } finally {
                ndef?.close()
            }
        }
        builder.setNegativeButton("No") { dialog, which ->
            mostrarMensaje("Verify data")
        }

        val dialog: AlertDialog = builder.create()
        dialog.show()
    }

    fun clear_listas() {
        variables.positionsList_fractura.clear()
        variables.positionsList_hemorragia.clear()
        variables.positionsList_perforacion.clear()
        variables.positionsList_no_perforacion.clear()
        variables.positionsList_quemadura.clear()
    }

    // Método para mostrar un mensaje Toast
    fun mostrarMensaje(mensaje: String) {
        Toast.makeText(this, mensaje, Toast.LENGTH_SHORT).show()
    }

    private fun getRetrofit(): Retrofit {
        return Retrofit.Builder()
            .baseUrl("http://193.146.212.177/")
            .addConverterFactory(GsonConverterFactory.create())
            .build()
    }
}
```

DatosBasicos.kt

```
package NFMC

import android.content.Intent
import android.os.Bundle
import android.widget.EditText
import androidx.appcompat.app.AlertDialog
import androidx.appcompat.app.AppCompatActivity
import com.example.prueba1.R
import java.text.SimpleDateFormat
import java.util.Calendar
import java.util.Locale

class DatosBasicos<Date> : AppCompatActivity() {

    private lateinit var nombreText: EditText
    private lateinit var apellidosText: EditText
    private lateinit var sexoText: EditText
    private lateinit var numeroIDText: EditText
    private lateinit var fechaNacimientoText: EditText
    private lateinit var lugarOrigenText: EditText
    private lateinit var fuerzasArmadasOrigenText: EditText

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_datos_basicos)

        val bundle = intent.extras
        // nombre
        val nombre = bundle?.getString("nombre_vuelta")
        nombreText = findViewById(R.id.escribir_nombre)
        nombreText.setText(nombre)

        // apellidos
        val apellidos = bundle?.getString("apellidos_vuelta")
        apellidosText = findViewById(R.id.escribir_apellidos)
        apellidosText.setText(apellidos)

        // sexo
        val sexo = bundle?.getString("sexo_vuelta")
        sexoText = findViewById(R.id.escribir_sexo)
        sexoText.setText(sexo)

        // numeroID
        val numeroID = bundle?.getString("Id_vuelta")
        numeroIDText = findViewById(R.id.escribir_numero_id)
        numeroIDText.setText(numeroID)

        // fechaNacimiento
        val fechaNacimiento = bundle?.getString("nacimiento_vuelta")
        fechaNacimientoText = findViewById(R.id.escribir_fecha_nacimiento)
        fechaNacimientoText.setText(fechaNacimiento)

        // lugarOrigen
        val lugarOrigen = bundle?.getString("origen_vuelta")
        lugarOrigenText = findViewById(R.id.escribir_lugar_origen)
        lugarOrigenText.setText(lugarOrigen)

        // fuerzasArmadasOrigen
        val fuerzasArmadasOrigen = bundle?.getString("fuerzasArmadas_vuelta")
        fuerzasArmadasOrigenText = findViewById(R.id.escribir_fuerzas_armadas_origen)
        fuerzasArmadasOrigenText.setText(fuerzasArmadasOrigen)
    }

    override fun onBackPressed() {
        val nombre = nombreText.text.toString().trim()
        val apellidos = apellidosText.text.toString().trim()
        val sexo = sexoText.text.toString().trim()
        val numeroID = numeroIDText.text.toString().trim()
        val fechaNacimiento = fechaNacimientoText.text.toString().trim()
        val lugarOrigen = lugarOrigenText.text.toString().trim()
        val fuerzasArmadasOrigen = fuerzasArmadasOrigenText.text.toString().trim()
```

```kotlin
        // Verificar si los campos están en blanco
        if (nombre.isBlank() && apellidos.isBlank() && sexo.isBlank() &&
            numeroID.isBlank() && fechaNacimiento.isBlank() &&
            lugarOrigen.isBlank() && fuerzasArmadasOrigen.isBlank()
        ) {
            super.onBackPressed() // Si todos los campos están en blanco, se permite volver atrás
            return
        }

        if (!camposModificados()) {
            super.onBackPressed()
            return
        }

        // Verificar si el campo de fecha de nacimiento no está en blanco antes de validar la fecha
        if (fechaNacimiento.isNotBlank()) {
            if (!esFechaValida(fechaNacimiento)) {
                fechaNacimientoText.error = "The date of birth provided is not valid"
                return
            }

            val sdf = SimpleDateFormat("dd/MM/yyyy", Locale.getDefault())
            val fechaNacimientoDate: java.util.Date? = sdf.parse(fechaNacimiento)
            val fechaActual = Calendar.getInstance().time

            if (fechaNacimientoDate != null) {
                if (fechaNacimientoDate.after(fechaActual)) {
                    fechaNacimientoText.error =
                        "The date of birth entered must be before or equal to the current date"
                    return
                }
            }

        }

        mostrarDialogoConfirmacion()
    }

    private fun camposModificados(): Boolean {
        val nombre = nombreText.text.toString().trim()
        val apellidos = apellidosText.text.toString().trim()
        val sexo = sexoText.text.toString().trim()
        val numeroID = numeroIDText.text.toString().trim()
        val fechaNacimiento = fechaNacimientoText.text.toString().trim()
        val lugarOrigen = lugarOrigenText.text.toString().trim()
        val fuerzasArmadasOrigen = fuerzasArmadasOrigenText.text.toString().trim()

        val bundle = intent.extras

        val nombreOriginal = bundle?.getString("nombre_vuelta").toString().trim()
        val apellidosOriginal = bundle?.getString("apellidos_vuelta").toString().trim()
        val sexoOriginal = bundle?.getString("sexo_vuelta").toString().trim()
        val numeroIDOriginal = bundle?.getString("Id_vuelta").toString().trim()
        val fechaNacimientoOriginal = bundle?.getString("nacimiento_vuelta").toString().trim()
        val lugarOrigenOriginal = bundle?.getString("origen_vuelta").toString().trim()
        val fuerzasArmadasOrigenOriginal =
            bundle?.getString("fuerzasArmadas_vuelta").toString().trim()

        return (nombre != nombreOriginal || apellidos != apellidosOriginal || sexo != sexoOriginal ||
                numeroID != numeroIDOriginal || fechaNacimiento != fechaNacimientoOriginal ||
                lugarOrigen != lugarOrigenOriginal || fuerzasArmadasOrigen != fuerzasArmadasOrigenOriginal)
    }

    private fun mostrarDialogoConfirmacion() {
        val alertDialog: AlertDialog.Builder = AlertDialog.Builder(this)
        alertDialog.setTitle("Confirmation")
        alertDialog.setMessage("Please confirm that you wish to make changes to the data")

        alertDialog.setPositiveButton("Yes") { _, _ ->
            enviarDatosModificados()
        }

        alertDialog.setNegativeButton("No") { dialog, _ ->
            dialog.dismiss()
        }
```

```
        alertDialog.show()
    }

    private fun enviarDatosModificados() {
        val intent = Intent(this, MenuPrincipal::class.java)
        // Envía los datos modificados a través del intent
        intent.putExtra("nombre", nombreText.text.toString().trim())
        intent.putExtra("apellidos", apellidosText.text.toString().trim())
        intent.putExtra("sexo", sexoText.text.toString().trim())
        intent.putExtra("numeroID", numeroIDText.text.toString().trim())
        intent.putExtra("fechaNacimiento", fechaNacimientoText.text.toString().trim())
        intent.putExtra("lugarOrigen", lugarOrigenText.text.toString().trim())
        intent.putExtra(
            "fuerzasArmadasOrigen",
            fuerzasArmadasOrigenText.text.toString().trim()
        )

        setResult(RESULT_OK, intent)
        finish()
    }

    // Agregar la validación de la fecha en esFechaValida
    private fun esFechaValida(fecha: String): Boolean {
        val partesFecha = fecha.split("/")
        val dia = partesFecha[0].toInt()
        val mes = partesFecha[1].toInt()
        val año = partesFecha[2].toInt()

        // Validar el rango del año, el mes y el día
        if (año < 1925 || año > 9999 || mes < 1 || mes > 12 || dia < 1) {
            return false
        }
        val diasPorMes = when (mes) {
            1, 3, 5, 7, 8, 10, 12 -> 31
            4, 6, 9, 11 -> 30
            2 -> if (esAnoBisiesto(año)) 29 else 28
            else -> return false // Mes inválido
        }

        return dia <= diasPorMes
    }

    private fun esAnoBisiesto(año: Int): Boolean {
        return año % 4 == 0 && (año % 100 != 0 || año % 400 == 0)
    }


}


/*

POSIBLE VERSION OPTIMIZADA

import android.content.Intent
import android.os.Bundle
import androidx.appcompat.app.AlertDialog
import androidx.appcompat.app.AppCompatActivity
import com.example.prueba1.databinding.ActivityDatosBasicosBinding
import java.text.SimpleDateFormat
import java.util.Locale

class DatosBasicos : AppCompatActivity() {

    private lateinit var binding: ActivityDatosBasicosBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityDatosBasicosBinding.inflate(layoutInflater)
        setContentView(binding.root)

        val bundle = intent.extras
        cargarDatos(bundle)
    }
```

```
override fun onBackPressed() {
    val bundle = intent.extras
    val datosOriginales = obtenerDatosOriginales(bundle)
    val datosActuales = obtenerDatosActuales()

    if (datosActuales == datosOriginales) {
        super.onBackPressed()
        return
    }

    val fechaNacimiento = binding.escribirFechaNacimiento.text.toString().trim()
    if (fechaNacimiento.isNotBlank() && !esFechaValida(fechaNacimiento)) {
        binding.escribirFechaNacimiento.error = "La fecha de nacimiento no es válida"
        return
    }

    mostrarDialogoConfirmacion()
}

private fun cargarDatos(bundle: Bundle?) {
    binding.apply {
        escribirNombre.setText(bundle?.getString("nombre_vuelta"))
        escribirApellidos.setText(bundle?.getString("apellidos_vuelta"))
        escribirSexo.setText(bundle?.getString("sexo_vuelta"))
        escribirNumeroId.setText(bundle?.getString("Id_vuelta"))
        escribirFechaNacimiento.setText(bundle?.getString("nacimiento_vuelta"))
        escribirLugarOrigen.setText(bundle?.getString("origen_vuelta"))
        escribirFuerzasArmadasOrigen.setText(bundle?.getString("fuerzasArmadas_vuelta"))
    }
}

private fun obtenerDatosOriginales(bundle: Bundle?): String {
    return "${bundle?.getString("nombre_vuelta")}" +
            "${bundle?.getString("apellidos_vuelta")}" +
            "${bundle?.getString("sexo_vuelta")}" +
            "${bundle?.getString("Id_vuelta")}" +
            "${bundle?.getString("nacimiento_vuelta")}" +
            "${bundle?.getString("origen_vuelta")}" +
            "${bundle?.getString("fuerzasArmadas_vuelta")}"
}

private fun obtenerDatosActuales(): String {
    return binding.run {
        "${escribirNombre.text}" +
                "${escribirApellidos.text}" +
                "${escribirSexo.text}" +
                "${escribirNumeroId.text}" +
                "${escribirFechaNacimiento.text}" +
                "${escribirLugarOrigen.text}" +
                "${escribirFuerzasArmadasOrigen.text}"
    }
}

private fun mostrarDialogoConfirmacion() {
    AlertDialog.Builder(this)
        .setTitle("Confirmación")
        .setMessage("¿Desea guardar los cambios?")
        .setPositiveButton("Sí") { _, _ -> enviarDatosModificados() }
        .setNegativeButton("No") { dialog, _ -> dialog.dismiss() }
        .show()
}

private fun enviarDatosModificados() {
    val intent = Intent(this, MenuPrincipal::class.java).apply {
        putExtras(Bundle().apply {
            putString("nombre", binding.escribirNombre.text.toString())
            putString("apellidos", binding.escribirApellidos.text.toString())
            putString("sexo", binding.escribirSexo.text.toString())
            putString("numeroID", binding.escribirNumeroId.text.toString())
            putString("fechaNacimiento", binding.escribirFechaNacimiento.text.toString())
            putString("lugarOrigen", binding.escribirLugarOrigen.text.toString())
            putString("fuerzasArmadasOrigen", binding.escribirFuerzasArmadasOrigen.text.toString())
        })
    }
```

```
        setResult(RESULT_OK, intent)
        finish()
    }

    private fun esFechaValida(fecha: String): Boolean {
        val sdf = SimpleDateFormat("dd/MM/yyyy", Locale.getDefault())
        sdf.isLenient = false
        return try {
            sdf.parse(fecha)
            true
        } catch (e: Exception) {
            false
        }
    }
}

 */
```

Causa.kt

```kotlin
package NFMC

import android.app.Activity
import android.content.Intent
import android.graphics.PorterDuff
import android.os.Bundle
import android.view.MotionEvent
import android.view.View
import android.widget.Button
import android.widget.ImageView
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import androidx.constraintlayout.widget.ConstraintLayout
import androidx.core.content.ContextCompat
import com.example.prueba1.R

class Causa : AppCompatActivity() {

    private var initialXFractura: Float = 0f
    private var initialYFractura: Float = 0f
    private var initialXHemorragia: Float = 0f
    private var initialYHemorragia: Float = 0f
    private var initialXPerforacion: Float = 0f
    private var initialYPerforacion: Float = 0f
    private var initialXsinPerforacion: Float = 0f
    private var initialYsinPerforacion: Float = 0f
    private var initialXquemado: Float = 0f
    private var initialYquemado: Float = 0f

    private var initialX: Float = 0f
    private var initialY: Float = 0f
    private var imageCounter_fractura = 0
    private var imageCounter_hemorragia = 0
    private var imageCounter_perforacion = 0
    private var imageCounter_no_perforacion = 0
    private var imageCounter_quemadura = 0

    private lateinit var positionsList_fractura: MutableList<Pair<Float, Float>>
    private lateinit var positionsList_hemorragia: MutableList<Pair<Float, Float>>
    private lateinit var positionsList_perforacion: MutableList<Pair<Float, Float>>
    private lateinit var positionsList_no_perforacion: MutableList<Pair<Float, Float>>
    private lateinit var positionsList_quemadura: MutableList<Pair<Float, Float>>

    private val deleteButtonMap_fractura = mutableMapOf<Button, Int>()
    private val deleteButtonMap_hemorragia = mutableMapOf<Button, Int>()
    private val deleteButtonMap_perforacion = mutableMapOf<Button, Int>()
    private val deleteButtonMap_no_perforacion = mutableMapOf<Button, Int>()
    private val deleteButtonMap_quemadura = mutableMapOf<Button, Int>()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_causa)

        // Inicializar la lista de posiciones
        recuperar_fractura()
        recuperar_hemorragia()
        recuperar_perforacion()
        recuperar_no_perforacion()
        recuperar_quemadura()

        // Configurar el onTouchListener para cada ImageView
        setupImageViewListeners()
    }

    private fun setupImageViewListeners() {
        val fracturaImageView = findViewById<ImageView>(R.id.simbolo_fractura)
        val hemorragiaImageView = findViewById<ImageView>(R.id.simbolo_hemorragia)
        val perforacionImageView = findViewById<ImageView>(R.id.simbolo_herida_perforacion)
        val sinPerforacionImageView = findViewById<ImageView>(R.id.simbolo_herida_no_perforacion)
        val quemadoImageView = findViewById<ImageView>(R.id.simbolo_quemado)

        initialXFractura = fracturaImageView.x
        initialYFractura = fracturaImageView.y
        initialXHemorragia = hemorragiaImageView.x
        initialYHemorragia = hemorragiaImageView.y
        initialXPerforacion = perforacionImageView.x
```

```kotlin
    initialYPerforacion = perforacionImageView.y
    initialXsinPerforacion = sinPerforacionImageView.x
    initialYsinPerforacion = sinPerforacionImageView.x
    initialXquemado = quemadoImageView.x
    initialYquemado = quemadoImageView.y


    fracturaImageView.setOnTouchListener { _, event ->
        handleTouchEvent(fracturaImageView, event)
    }

    hemorragiaImageView.setOnTouchListener { _, event ->
        handleTouchEvent2(hemorragiaImageView, event)
    }

    perforacionImageView.setOnTouchListener { _, event ->
        handleTouchEvent3(perforacionImageView, event)
    }

    sinPerforacionImageView.setOnTouchListener { _, event ->
        handleTouchEvent4(
            sinPerforacionImageView,
            event
        )
    }

    quemadoImageView.setOnTouchListener { _, event ->
        handleTouchEvent5(quemadoImageView, event)
    }

}

private fun handleTouchEvent(imageView: ImageView, event: MotionEvent): Boolean {
    when (event.action) {
        MotionEvent.ACTION_DOWN -> {
            // Guarda las coordenadas iniciales del evento
            initialX = event.rawX - imageView.x
            initialY = event.rawY - imageView.y
        }

        MotionEvent.ACTION_MOVE -> {
            imageView.x = event.rawX - initialX
            imageView.y = event.rawY - initialY
        }

        MotionEvent.ACTION_UP -> {
            // Siempre que se levante el dedo, crea una copia de la imagen movida
            createNewImage1(imageView.x, imageView.y + 248, this.imageCounter_fractura)
            // Restaurar la posición original de la imagen original
            imageView.x = initialXFractura + 44
            imageView.y = initialYFractura + 88
            this.imageCounter_fractura++


        }
    }
    return true
}

private fun handleTouchEvent2(imageView: ImageView, event: MotionEvent): Boolean {
    when (event.action) {
        MotionEvent.ACTION_DOWN -> {
            // Guarda las coordenadas iniciales del evento
            initialX = event.rawX - imageView.x
            initialY = event.rawY - imageView.y
        }

        MotionEvent.ACTION_MOVE -> {
            imageView.x = event.rawX - initialX
            imageView.y = event.rawY - initialY
        }

        MotionEvent.ACTION_UP -> {
            // Siempre que se levante el dedo, crea una copia de la imagen movida
            createNewImage2(imageView.x, imageView.y + 248, this.imageCounter_hemorragia)
            // Restaurar la posición original de la imagen original
            imageView.x = initialXHemorragia + 242
            imageView.y = initialYHemorragia + 88
            this.imageCounter_hemorragia++
```

```kotlin
            }
        }
        return true
    }

    private fun handleTouchEvent3(imageView: ImageView, event: MotionEvent): Boolean {
        when (event.action) {
            MotionEvent.ACTION_DOWN -> {
                // Guarda las coordenadas iniciales del evento
                initialX = event.rawX - imageView.x
                initialY = event.rawY - imageView.y
            }

            MotionEvent.ACTION_MOVE -> {
                imageView.x = event.rawX - initialX
                imageView.y = event.rawY - initialY
            }

            MotionEvent.ACTION_UP -> {
                // Siempre que se levante el dedo, crea una copia de la imagen movida
                createNewImage3(imageView.x, imageView.y + 248, this.imageCounter_perforacion)
                // Restaurar la posición original de la imagen original
                imageView.x = initialXPerforacion + 441
                imageView.y = initialYPerforacion + 88
                this.imageCounter_perforacion++

            }
        }
        return true
    }

    private fun handleTouchEvent4(imageView: ImageView, event: MotionEvent): Boolean {
        when (event.action) {
            MotionEvent.ACTION_DOWN -> {
                // Guarda las coordenadas iniciales del evento
                initialX = event.rawX - imageView.x
                initialY = event.rawY - imageView.y
            }

            MotionEvent.ACTION_MOVE -> {
                imageView.x = event.rawX - initialX
                imageView.y = event.rawY - initialY
            }

            MotionEvent.ACTION_UP -> {
                // Siempre que se levante el dedo, crea una copia de la imagen movida
                createNewImage4(imageView.x, imageView.y + 248, this.imageCounter_no_perforacion)
                // Restaurar la posición original de la imagen original
                imageView.x = initialXsinPerforacion + 639
                imageView.y = initialYsinPerforacion + 88
                this.imageCounter_no_perforacion++

            }
        }
        return true
    }

    private fun handleTouchEvent5(imageView: ImageView, event: MotionEvent): Boolean {
        when (event.action) {
            MotionEvent.ACTION_DOWN -> {
                // Guarda las coordenadas iniciales del evento
                initialX = event.rawX - imageView.x
                initialY = event.rawY - imageView.y
            }

            MotionEvent.ACTION_MOVE -> {
                imageView.x = event.rawX - initialX
                imageView.y = event.rawY - initialY
            }

            MotionEvent.ACTION_UP -> {
                // Siempre que se levante el dedo, crea una copia de la imagen movida
                createNewImage5(imageView.x, imageView.y + 254, this.imageCounter_quemadura)
                // Restaurar la posición original de la imagen original
                imageView.x = initialXquemado + 838
                imageView.y = initialYquemado + 88
                this.imageCounter_quemadura++
```

```
        }
    }
    return true
}


private fun createNewImage1(x: Float, y: Float, index: Int) {
    val originalImage = findViewById<ImageView>(R.id.simbolo_fractura)
    val newImage = ImageView(this)// Obtener la imagen original

    // Establecer la imagen de origen y las dimensiones
    newImage.setImageResource(R.drawable.simbolo_fractura_bueno)
    newImage.layoutParams =
        ConstraintLayout.LayoutParams(originalImage.width, originalImage.height)

    // Establecer el color de la imagen
    newImage.setColorFilter(
        ContextCompat.getColor(this, R.color.amarillo),
        PorterDuff.Mode.SRC_ATOP
    )

    // Establecer la posición de la nueva imagen
    newImage.x = x
    newImage.y = y

    // Asignar un ID y una etiqueta a la nueva imagen si es necesario
    newImage.id = View.generateViewId()
    newImage.tag = "Image $index"

    // Agregar la nueva imagen al diseño
    val layout = findViewById<ConstraintLayout>(R.id.causa)
    layout.addView(newImage)

    // Guardar la posición de la nueva imagen en la lista
    positionsList_fractura.add(Pair(newImage.x, newImage.y))

    //Activar visibilidad
    newImage.visibility = View.VISIBLE

    // Crear un botón para eliminar la imagen y sus coordenadas asociadas
    val deleteButton = Button(this)
    deleteButton.text = "Delete fracture"
    deleteButton.layoutParams = ConstraintLayout.LayoutParams(
        200.dpToPx(), //ancho
        ConstraintLayout.LayoutParams.WRAP_CONTENT //alto
    )

    // Establecer el color del texto del botón
    deleteButton.setTextColor(ContextCompat.getColor(this, R.color.amarillo))

    // Agregar el botón de eliminar al diseño
    layout.addView(deleteButton)

    // Agregar el botón al HashMap junto con su índice correspondiente
    deleteButtonMap_fractura[deleteButton] = index // Asignar el índice correspondiente

    // Configurar el OnClickListener para el botón de eliminar
    deleteButton.setOnClickListener {
        val idx = deleteButtonMap_fractura[it] ?: -1
        if (idx != -1) {
            layout.removeView(newImage)
            positionsList_fractura.removeAt(idx)
            layout.removeView(deleteButton)
            deleteButtonMap_fractura.remove(deleteButton)
            if (imageCounter_fractura > 0) {
                imageCounter_fractura--
            }
        }
    }

    // Posicionar el botón de eliminar en la esquina inferior derecha
    val params = deleteButton.layoutParams as ConstraintLayout.LayoutParams
    params.bottomToBottom = ConstraintLayout.LayoutParams.PARENT_ID
    params.startToStart = ConstraintLayout.LayoutParams.PARENT_ID
    params.bottomMargin = 140.dpToPx()
    deleteButton.layoutParams = params

}
```

```kotlin
private fun createNewImage2(x: Float, y: Float, index: Int) {
    val originalImage = findViewById<ImageView>(R.id.simbolo_hemorragia)
    val newImage = ImageView(this)// Obtener la imagen original

    // Establecer la imagen de origen y las dimensiones
    newImage.setImageResource(R.drawable.simbolo_hemorragia_bueno)
    newImage.layoutParams =
        ConstraintLayout.LayoutParams(originalImage.width, originalImage.height)

    // Establecer el color de la imagen
    newImage.setColorFilter(
        ContextCompat.getColor(this, R.color.rojo),
        PorterDuff.Mode.SRC_ATOP
    )

    // Establecer la posición de la nueva imagen
    newImage.x = x
    newImage.y = y

    // Asignar un ID y una etiqueta a la nueva imagen si es necesario
    newImage.id = View.generateViewId()
    newImage.tag = "Image $index"

    // Agregar la nueva imagen al diseño
    val layout = findViewById<ConstraintLayout>(R.id.causa)
    layout.addView(newImage)

    // Guardar la posición de la nueva imagen en la lista si es necesario
    positionsList_hemorragia.add(Pair(newImage.x, newImage.y))

    //Activar visibilidad
    newImage.visibility = View.VISIBLE

    // Crear un botón para eliminar la imagen y sus coordenadas asociadas
    val deleteButton = Button(this)
    deleteButton.text = "Delete hemorrhage"
    deleteButton.layoutParams = ConstraintLayout.LayoutParams(
        200.dpToPx(),
        ConstraintLayout.LayoutParams.WRAP_CONTENT
    )

    // Establecer el color del texto del botón
    deleteButton.setTextColor(ContextCompat.getColor(this, R.color.rojo))

    // Agregar el botón de eliminar al diseño
    layout.addView(deleteButton)

    // Agregar el botón al HashMap junto con su índice correspondiente
    deleteButtonMap_hemorragia[deleteButton] = index // Asignar el índice correspondiente

    // Configurar el OnClickListener para el botón de eliminar
    deleteButton.setOnClickListener {
        val idx = deleteButtonMap_hemorragia[it] ?: -1
        if (idx != -1) {
            layout.removeView(newImage)
            positionsList_hemorragia.removeAt(idx)
            layout.removeView(deleteButton)
            deleteButtonMap_hemorragia.remove(deleteButton)
            if (imageCounter_hemorragia > 0) {
                imageCounter_hemorragia--
            }
        }
    }

    // Posicionar el botón de eliminar en relación con el botón de eliminar fractura
    val params = deleteButton.layoutParams as ConstraintLayout.LayoutParams
    params.bottomToBottom = ConstraintLayout.LayoutParams.PARENT_ID
    params.startToStart = ConstraintLayout.LayoutParams.PARENT_ID
    params.topMargin = 5.dpToPx()
    params.bottomMargin = 100.dpToPx()
    deleteButton.layoutParams = params

}

private fun createNewImage3(x: Float, y: Float, index: Int) {
    val originalImage = findViewById<ImageView>(R.id.simbolo_herida_perforacion)
    val newImage = ImageView(this)// Obtener la imagen original
```

```
    // Establecer la imagen de origen y las dimensiones
    newImage.setImageResource(R.drawable.simbolo_herida_perforacion_bueno)
    newImage.layoutParams =
        ConstraintLayout.LayoutParams(originalImage.width, originalImage.height)

    // Establecer el color de la imagen
    newImage.setColorFilter(
        ContextCompat.getColor(this, R.color.purple_200),
        PorterDuff.Mode.SRC_ATOP
    )

    // Establecer la posición de la nueva imagen
    newImage.x = x
    newImage.y = y

    // Asignar un ID y una etiqueta a la nueva imagen si es necesario
    newImage.id = View.generateViewId()
    newImage.tag = "Image $index"

    // Agregar la nueva imagen al diseño
    val layout = findViewById<ConstraintLayout>(R.id.causa)
    layout.addView(newImage)

    // Guardar la posición de la nueva imagen en la lista si es necesario
    positionsList_perforacion.add(Pair(newImage.x, newImage.y))

    //Activar visibilidad
    newImage.visibility = View.VISIBLE

    // Crear un botón para eliminar la imagen y sus coordenadas asociadas
    val deleteButton = Button(this)
    deleteButton.text = "Delete perforation"
    deleteButton.layoutParams = ConstraintLayout.LayoutParams(
        200.dpToPx(),
        ConstraintLayout.LayoutParams.WRAP_CONTENT
    )

    // Establecer el color del texto del botón
    deleteButton.setTextColor(ContextCompat.getColor(this, R.color.purple_200))

    // Agregar el botón de eliminar al diseño
    layout.addView(deleteButton)

    // Agregar el botón al HashMap junto con su índice correspondiente
    deleteButtonMap_perforacion[deleteButton] = index // Asignar el índice correspondiente

    // Configurar el OnClickListener para el botón de eliminar
    deleteButton.setOnClickListener {
        val idx = deleteButtonMap_perforacion[it] ?: -1
        if (idx != -1) {
            layout.removeView(newImage)
            positionsList_perforacion.removeAt(idx)
            layout.removeView(deleteButton)
            deleteButtonMap_perforacion.remove(deleteButton)
            if (imageCounter_perforacion > 0) {
                imageCounter_perforacion--
            }
        }
    }

    // Posicionar el botón de eliminar en la esquina inferior derecha
    val params = deleteButton.layoutParams as ConstraintLayout.LayoutParams
    params.bottomToBottom = ConstraintLayout.LayoutParams.PARENT_ID
    params.endToEnd = ConstraintLayout.LayoutParams.PARENT_ID
    params.bottomMargin = 140.dpToPx()
    deleteButton.layoutParams = params

}

private fun createNewImage4(x: Float, y: Float, index: Int) {
    val originalImage = findViewById<ImageView>(R.id.simbolo_herida_no_perforacion)
    val newImage = ImageView(this)// Obtener la imagen original

    // Establecer la imagen de origen y las dimensiones
    newImage.setImageResource(R.drawable.simbolo_herida_sin_perforacion_bueno)
    newImage.layoutParams =
        ConstraintLayout.LayoutParams(originalImage.width, originalImage.height)

    // Establecer el color de la imagen
```

```
    newImage.setColorFilter(
        ContextCompat.getColor(this, R.color.teal_200),
        PorterDuff.Mode.SRC_ATOP
    )

    // Establecer la posición de la nueva imagen
    newImage.x = x
    newImage.y = y

    // Asignar un ID y una etiqueta a la nueva imagen si es necesario
    newImage.id = View.generateViewId()
    newImage.tag = "Image $index"

    // Agregar la nueva imagen al diseño
    val layout = findViewById<ConstraintLayout>(R.id.causa)
    layout.addView(newImage)

    // Guardar la posición de la nueva imagen en la lista si es necesario
    positionsList_no_perforacion.add(Pair(newImage.x, newImage.y))

    //Activar visibilidad
    newImage.visibility = View.VISIBLE

    // Crear un botón para eliminar la imagen y sus coordenadas asociadas
    val deleteButton = Button(this)
    deleteButton.text = "Delete no perforation"
    deleteButton.layoutParams = ConstraintLayout.LayoutParams(
        ConstraintLayout.LayoutParams.WRAP_CONTENT,
        ConstraintLayout.LayoutParams.WRAP_CONTENT
    )

    // Establecer el color del texto del botón
    deleteButton.setTextColor(ContextCompat.getColor(this, R.color.teal_200))

    // Agregar el botón de eliminar al diseño
    layout.addView(deleteButton)

    // Agregar el botón al HashMap junto con su índice correspondiente
    deleteButtonMap_no_perforacion[deleteButton] = index // Asignar el índice correspondiente

    // Configurar el OnClickListener para el botón de eliminar
    deleteButton.setOnClickListener {
        val idx = deleteButtonMap_no_perforacion[it] ?: -1
        if (idx != -1) {
            layout.removeView(newImage)
            positionsList_no_perforacion.removeAt(idx)
            layout.removeView(deleteButton)
            deleteButtonMap_no_perforacion.remove(deleteButton)
            if (imageCounter_no_perforacion > 0) {
                imageCounter_no_perforacion--
            }
        }
    }

    // Posicionar el botón de eliminar en relación con el botón de eliminar fractura
    val params = deleteButton.layoutParams as ConstraintLayout.LayoutParams
    params.bottomToBottom = ConstraintLayout.LayoutParams.PARENT_ID
    params.startToStart = ConstraintLayout.LayoutParams.PARENT_ID
    params.endToEnd = ConstraintLayout.LayoutParams.PARENT_ID
    params.topMargin = 5.dpToPx()
    params.bottomMargin = 60.dpToPx()
    deleteButton.layoutParams = params

}

private fun createNewImage5(x: Float, y: Float, index: Int) {
    val originalImage = findViewById<ImageView>(R.id.simbolo_quemado)
    val newImage = ImageView(this)// Obtener la imagen original

    // Establecer la imagen de origen y las dimensiones
    newImage.setImageResource(R.drawable.simbolo_quemado_bueno)
    newImage.layoutParams =
        ConstraintLayout.LayoutParams(originalImage.width, originalImage.height)

    // Establecer el color de la imagen
    newImage.setColorFilter(
        ContextCompat.getColor(this, R.color.naranja_chillon),
        PorterDuff.Mode.SRC_ATOP
    )
```

```
    // Establecer la posición de la nueva imagen
    newImage.x = x
    newImage.y = y

    // Asignar un ID y una etiqueta a la nueva imagen si es necesario
    newImage.id = View.generateViewId()
    newImage.tag = "Image $index"

    // Agregar la nueva imagen al diseño
    val layout = findViewById<ConstraintLayout>(R.id.causa)
    layout.addView(newImage)

    // Guardar la posición de la nueva imagen en la lista si es necesario
    positionsList_quemadura.add(Pair(newImage.x, newImage.y))

    //Activar visibilidad
    newImage.visibility = View.VISIBLE

    // Crear un botón para eliminar la imagen y sus coordenadas asociadas
    val deleteButton = Button(this)
    deleteButton.text = "Delete burned area"
    deleteButton.layoutParams = ConstraintLayout.LayoutParams(
        200.dpToPx(),
        ConstraintLayout.LayoutParams.WRAP_CONTENT
    )

    // Establecer el color del texto del botón
    deleteButton.setTextColor(ContextCompat.getColor(this, R.color.naranja_chillon))

    // Agregar el botón de eliminar al diseño
    layout.addView(deleteButton)

    // Agregar el botón al HashMap junto con su índice correspondiente
    deleteButtonMap_quemadura[deleteButton] = index // Asignar el índice correspondiente

    // Configurar el OnClickListener para el botón de eliminar
    deleteButton.setOnClickListener {
        val idx = deleteButtonMap_quemadura[it] ?: -1
        if (idx != -1) {
            layout.removeView(newImage)
            positionsList_quemadura.removeAt(idx)
            layout.removeView(deleteButton)
            deleteButtonMap_quemadura.remove(deleteButton)
            if (imageCounter_quemadura > 0) {
                imageCounter_quemadura--
            }
        }
    }

    // Posicionar el botón de eliminar en relación con el botón de eliminar fractura
    val params = deleteButton.layoutParams as ConstraintLayout.LayoutParams
    params.bottomToBottom = ConstraintLayout.LayoutParams.PARENT_ID
    params.endToEnd = ConstraintLayout.LayoutParams.PARENT_ID
    params.topMargin = 5.dpToPx()
    params.bottomMargin = 100.dpToPx()
    deleteButton.layoutParams = params


}

private fun recuperar_fractura() {

    val simbolo_fractura = findViewById<ImageView>(R.id.simbolo_fractura)

    // Guardar las coordenadas iniciales de la imagen original
    initialXFractura = simbolo_fractura.x
    initialYFractura = simbolo_fractura.y

    //Recibir valor imageCounter
    this.imageCounter_fractura = intent.getIntExtra("imageCounter_fractura_vuelta", 0)

    // Recibir el array de posiciones enviado por la actividad NFC
    positionsList_fractura =
        intent.getSerializableExtra("positionsList_fractura_vuelta") as? MutableList<Pair<Float, Float>>
            ?: mutableListOf()


    // Mostrar las imágenes y sus copias
```

```
    positionsList_fractura.forEachIndexed { index, pair ->
        val (x, y) = pair
        val newImage = ImageView(this)// Obtener la imagen original

        // Establecer la imagen de origen y las dimensiones
        newImage.setImageResource(R.drawable.simbolo_fractura_bueno)
        newImage.layoutParams = ConstraintLayout.LayoutParams(80.dpToPx(), 80.dpToPx())

        // Establecer el color de la imagen
        newImage.setColorFilter(
            ContextCompat.getColor(this, R.color.amarillo),
            PorterDuff.Mode.SRC_ATOP
        )

        // Establecer la posición de la nueva imagen
        newImage.x = x
        newImage.y = y

        // Agregar la nueva imagen al diseño
        val layout = findViewById<ConstraintLayout>(R.id.causa)
        layout.addView(newImage)

        //Activar visibilidad
        newImage.visibility = View.VISIBLE

        // Crear un botón para eliminar la imagen y sus coordenadas asociadas
        val deleteButton = Button(this)
        deleteButton.text = "Delete fracture"
        deleteButton.layoutParams = ConstraintLayout.LayoutParams(
            200.dpToPx(), //ancho
            ConstraintLayout.LayoutParams.WRAP_CONTENT //alto
        )

        // Establecer el color del texto del botón
        deleteButton.setTextColor(ContextCompat.getColor(this, R.color.amarillo))

        // Agregar el botón de eliminar al diseño
        layout.addView(deleteButton)

        // Agregar el botón al HashMap junto con su índice correspondiente
        deleteButtonMap_fractura[deleteButton] = index // Asignar el índice correspondiente

        // Configurar el OnClickListener para el botón de eliminar
        deleteButton.setOnClickListener {
            val idx = deleteButtonMap_fractura[it] ?: -1
            if (idx != -1) {
                layout.removeView(newImage)
                positionsList_fractura.removeAt(idx)
                layout.removeView(deleteButton)
                deleteButtonMap_fractura.remove(deleteButton)
                if (imageCounter_fractura > 0) {
                    imageCounter_fractura--
                }
            }
        }

        // Posicionar el botón de eliminar en la esquina inferior derecha
        val params = deleteButton.layoutParams as ConstraintLayout.LayoutParams
        params.bottomToBottom = ConstraintLayout.LayoutParams.PARENT_ID
        params.startToStart = ConstraintLayout.LayoutParams.PARENT_ID
        params.bottomMargin = 140.dpToPx()
        deleteButton.layoutParams = params
    }

}

private fun recuperar_hemorragia() {

    val simbolo_hemorragia = findViewById<ImageView>(R.id.simbolo_hemorragia)

    // Guardar las coordenadas iniciales de la imagen original
    initialXHemorragia = simbolo_hemorragia.x
    initialYHemorragia = simbolo_hemorragia.y

    //Recibir valor imageCounter
    this.imageCounter_hemorragia = intent.getIntExtra("imageCounter_hemorragia_vuelta", 0)

    // Recibir el array de posiciones enviado por la actividad NFC
    positionsList_hemorragia =
```

```kotlin
        intent.getSerializableExtra("positionsList_hemorragia_vuelta") as? MutableList<Pair<Float, Float>>
            ?: mutableListOf()


    // Mostrar las imágenes y sus copias
    positionsList_hemorragia.forEachIndexed { index, pair ->
        val (x, y) = pair
        val newImage = ImageView(this)// Obtener la imagen original

        // Establecer la imagen de origen y las dimensiones
        newImage.setImageResource(R.drawable.simbolo_hemorragia_bueno)
        newImage.layoutParams = ConstraintLayout.LayoutParams(80.dpToPx(), 80.dpToPx())

        // Establecer el color de la imagen
        newImage.setColorFilter(
            ContextCompat.getColor(this, R.color.rojo),
            PorterDuff.Mode.SRC_ATOP
        )

        // Establecer la posición de la nueva imagen
        newImage.x = x
        newImage.y = y

        // Agregar la nueva imagen al diseño
        val layout = findViewById<ConstraintLayout>(R.id.causa)
        layout.addView(newImage)

        //Activar visibilidad
        newImage.visibility = View.VISIBLE

        // Crear un botón para eliminar la imagen y sus coordenadas asociadas
        val deleteButton = Button(this)
        deleteButton.text = "Delete hemorrhage"
        deleteButton.layoutParams = ConstraintLayout.LayoutParams(
            200.dpToPx(),
            ConstraintLayout.LayoutParams.WRAP_CONTENT
        )

        // Establecer el color del texto del botón
        deleteButton.setTextColor(ContextCompat.getColor(this, R.color.rojo))

        // Agregar el botón de eliminar al diseño
        layout.addView(deleteButton)

        // Agregar el botón al HashMap junto con su índice correspondiente

        // Agregar el botón al HashMap junto con su índice correspondiente
        deleteButtonMap_hemorragia[deleteButton] = index // Asignar el índice correspondiente

        // Configurar el OnClickListener para el botón de eliminar
        deleteButton.setOnClickListener {
            val idx = deleteButtonMap_hemorragia[it] ?: -1
            if (idx != -1) {
                layout.removeView(newImage)
                positionsList_hemorragia.removeAt(idx)
                layout.removeView(deleteButton)
                deleteButtonMap_hemorragia.remove(deleteButton)
                if (imageCounter_hemorragia > 0) {
                    imageCounter_hemorragia--
                }
            }
        }

        // Posicionar el botón de eliminar en relación con el botón de eliminar fractura
        val params = deleteButton.layoutParams as ConstraintLayout.LayoutParams
        params.bottomToBottom = ConstraintLayout.LayoutParams.PARENT_ID
        params.startToStart = ConstraintLayout.LayoutParams.PARENT_ID
        params.topMargin = 5.dpToPx()
        params.bottomMargin = 100.dpToPx()
        deleteButton.layoutParams = params
    }

}

private fun recuperar_perforacion() {

    val simbolo_perforacion = findViewById<ImageView>(R.id.simbolo_herida_perforacion)

    // Guardar las coordenadas iniciales de la imagen original
```

```kotlin
initialXPerforacion = simbolo_perforacion.x
initialYPerforacion = simbolo_perforacion.y

//Recibir valor imageCounter
this.imageCounter_perforacion = intent.getIntExtra("imageCounter_perforacion_vuelta", 0)

// Recibir el array de posiciones enviado por la actividad NFC
positionsList_perforacion =
    intent.getSerializableExtra("positionsList_perforacion_vuelta") as? MutableList<Pair<Float, Float>>
        ?: mutableListOf()


// Mostrar las imágenes y sus copias
positionsList_perforacion.forEachIndexed { index, pair ->
    val (x, y) = pair
    val newImage = ImageView(this)// Obtener la imagen original

    // Establecer la imagen de origen y las dimensiones
    newImage.setImageResource(R.drawable.simbolo_herida_perforacion_bueno)
    newImage.layoutParams = ConstraintLayout.LayoutParams(80.dpToPx(), 80.dpToPx())

    // Establecer el color de la imagen
    newImage.setColorFilter(
        ContextCompat.getColor(this, R.color.purple_200),
        PorterDuff.Mode.SRC_ATOP
    )

    // Establecer la posición de la nueva imagen
    newImage.x = x
    newImage.y = y

    // Agregar la nueva imagen al diseño
    val layout = findViewById<ConstraintLayout>(R.id.causa)
    layout.addView(newImage)

    //Activar visibilidad
    newImage.visibility = View.VISIBLE

    // Crear un botón para eliminar la imagen y sus coordenadas asociadas
    val deleteButton = Button(this)
    deleteButton.text = "Delete perforation"
    deleteButton.layoutParams = ConstraintLayout.LayoutParams(
        200.dpToPx(),
        ConstraintLayout.LayoutParams.WRAP_CONTENT
    )

    // Establecer el color del texto del botón
    deleteButton.setTextColor(ContextCompat.getColor(this, R.color.purple_200))

    // Agregar el botón de eliminar al diseño
    layout.addView(deleteButton)

    // Agregar el botón al HashMap junto con su índice correspondiente

    // Agregar el botón al HashMap junto con su índice correspondiente
    deleteButtonMap_perforacion[deleteButton] = index // Asignar el índice correspondiente

    // Configurar el OnClickListener para el botón de eliminar
    deleteButton.setOnClickListener {
        val idx = deleteButtonMap_perforacion[it] ?: -1
        if (idx != -1) {
            layout.removeView(newImage)
            positionsList_perforacion.removeAt(idx)
            layout.removeView(deleteButton)
            deleteButtonMap_perforacion.remove(deleteButton)
            if (imageCounter_perforacion > 0) {
                imageCounter_perforacion--
            }
        }
    }

    // Posicionar el botón de eliminar en la esquina inferior derecha
    val params = deleteButton.layoutParams as ConstraintLayout.LayoutParams
    params.bottomToBottom = ConstraintLayout.LayoutParams.PARENT_ID
    params.endToEnd = ConstraintLayout.LayoutParams.PARENT_ID
    params.bottomMargin = 140.dpToPx()
    deleteButton.layoutParams = params
}
```

```kotlin
    }

    private fun recuperar_no_perforacion() {

        val simbolo_no_perforacion = findViewById<ImageView>(R.id.simbolo_herida_no_perforacion)

        // Guardar las coordenadas iniciales de la imagen original
        initialXsinPerforacion = simbolo_no_perforacion.x
        initialYsinPerforacion = simbolo_no_perforacion.y

        //Recibir valor imageCounter
        this.imageCounter_no_perforacion =
            intent.getIntExtra("imageCounter_no_perforacion_vuelta", 0)

        // Recibir el array de posiciones enviado por la actividad NFC
        positionsList_no_perforacion =
            intent.getSerializableExtra("positionsList_no_perforacion_vuelta") as? MutableList<Pair<Float, Float>>
                ?: mutableListOf()


        // Mostrar las imágenes y sus copias
        positionsList_no_perforacion.forEachIndexed { index, pair ->
            val (x, y) = pair
            val newImage = ImageView(this)// Obtener la imagen original

            // Establecer la imagen de origen y las dimensiones
            newImage.setImageResource(R.drawable.simbolo_herida_sin_perforacion_bueno)
            newImage.layoutParams = ConstraintLayout.LayoutParams(80.dpToPx(), 80.dpToPx())

            // Establecer el color de la imagen
            newImage.setColorFilter(
                ContextCompat.getColor(this, R.color.teal_200),
                PorterDuff.Mode.SRC_ATOP
            )

            // Establecer la posición de la nueva imagen
            newImage.x = x
            newImage.y = y

            // Agregar la nueva imagen al diseño
            val layout = findViewById<ConstraintLayout>(R.id.causa)
            layout.addView(newImage)

            //Activar visibilidad
            newImage.visibility = View.VISIBLE

            // Crear un botón para eliminar la imagen y sus coordenadas asociadas
            val deleteButton = Button(this)
            deleteButton.text = "Delete no perforation"
            deleteButton.layoutParams = ConstraintLayout.LayoutParams(
                ConstraintLayout.LayoutParams.WRAP_CONTENT,
                ConstraintLayout.LayoutParams.WRAP_CONTENT
            )

            // Establecer el color del texto del botón
            deleteButton.setTextColor(ContextCompat.getColor(this, R.color.teal_200))

            // Agregar el botón de eliminar al diseño
            layout.addView(deleteButton)

            // Agregar el botón al HashMap junto con su índice correspondiente

            // Agregar el botón al HashMap junto con su índice correspondiente
            deleteButtonMap_no_perforacion[deleteButton] =
                index // Asignar el índice correspondiente

            // Configurar el OnClickListener para el botón de eliminar
            deleteButton.setOnClickListener {
                val idx = deleteButtonMap_no_perforacion[it] ?: -1
                if (idx != -1) {
                    layout.removeView(newImage)
                    positionsList_no_perforacion.removeAt(idx)
                    layout.removeView(deleteButton)
                    deleteButtonMap_no_perforacion.remove(deleteButton)
                    if (imageCounter_no_perforacion > 0) {
                        imageCounter_no_perforacion--
                    }
                }
            }
```

```
        // Posicionar el botón de eliminar en relación con el botón de eliminar fractura
        val params = deleteButton.layoutParams as ConstraintLayout.LayoutParams
        params.bottomToBottom = ConstraintLayout.LayoutParams.PARENT_ID
        params.startToStart = ConstraintLayout.LayoutParams.PARENT_ID
        params.endToEnd = ConstraintLayout.LayoutParams.PARENT_ID
        params.topMargin = 5.dpToPx()
        params.bottomMargin = 60.dpToPx()
        deleteButton.layoutParams = params
    }

}

private fun recuperar_quemadura() {

    val simbolo_quemado = findViewById<ImageView>(R.id.simbolo_quemado)

    // Guardar las coordenadas iniciales de la imagen original
    initialXquemado = simbolo_quemado.x
    initialYquemado = simbolo_quemado.y

    //Recibir valor imageCounter
    this.imageCounter_quemadura = intent.getIntExtra("imageCounter_quemadura_vuelta", 0)

    // Recibir el array de posiciones enviado por la actividad NFC
    positionsList_quemadura =
        intent.getSerializableExtra("positionsList_quemadura_vuelta") as? MutableList<Pair<Float, Float>>
            ?: mutableListOf()


    // Mostrar las imágenes y sus copias
    positionsList_quemadura.forEachIndexed { index, pair ->
        val (x, y) = pair
        val newImage = ImageView(this)// Obtener la imagen original

        // Establecer la imagen de origen y las dimensiones
        newImage.setImageResource(R.drawable.simbolo_quemado_bueno)
        newImage.layoutParams = ConstraintLayout.LayoutParams(80.dpToPx(), 80.dpToPx())

        // Establecer el color de la imagen
        newImage.setColorFilter(
            ContextCompat.getColor(this, R.color.naranja_chillon),
            PorterDuff.Mode.SRC_ATOP
        )

        // Establecer la posición de la nueva imagen
        newImage.x = x
        newImage.y = y

        // Agregar la nueva imagen al diseño
        val layout = findViewById<ConstraintLayout>(R.id.causa)
        layout.addView(newImage)

        //Activar visibilidad
        newImage.visibility = View.VISIBLE

        // Crear un botón para eliminar la imagen y sus coordenadas asociadas
        val deleteButton = Button(this)
        deleteButton.text = "Delete burned area"
        deleteButton.layoutParams = ConstraintLayout.LayoutParams(
            200.dpToPx(),
            ConstraintLayout.LayoutParams.WRAP_CONTENT
        )

        // Establecer el color del texto del botón
        deleteButton.setTextColor(ContextCompat.getColor(this, R.color.naranja_chillon))

        // Agregar el botón de eliminar al diseño
        layout.addView(deleteButton)

        // Agregar el botón al HashMap junto con su índice correspondiente

        // Agregar el botón al HashMap junto con su índice correspondiente
        deleteButtonMap_quemadura[deleteButton] = index // Asignar el índice correspondiente

        // Configurar el OnClickListener para el botón de eliminar
        deleteButton.setOnClickListener {
            val idx = deleteButtonMap_quemadura[it] ?: -1
            if (idx != -1) {
```

```
                layout.removeView(newImage)
                positionsList_quemadura.removeAt(idx)
                layout.removeView(deleteButton)
                deleteButtonMap_quemadura.remove(deleteButton)
                if (imageCounter_quemadura > 0) {
                    imageCounter_quemadura--
                }
            }
        }

        // Posicionar el botón de eliminar en relación con el botón de eliminar fractura
        val params = deleteButton.layoutParams as ConstraintLayout.LayoutParams
        params.bottomToBottom = ConstraintLayout.LayoutParams.PARENT_ID
        params.endToEnd = ConstraintLayout.LayoutParams.PARENT_ID
        params.topMargin = 5.dpToPx()
        params.bottomMargin = 100.dpToPx()
        deleteButton.layoutParams = params
    }

}


override fun onBackPressed() {
    val intent = Intent(this, MenuPrincipal::class.java)
    intent.putExtra("positionsList_fractura", ArrayList(positionsList_fractura))
    intent.putExtra("positionsList_hemorragia", ArrayList(positionsList_hemorragia))
    intent.putExtra("positionsList_perforacion", ArrayList(positionsList_perforacion))
    intent.putExtra("positionsList_no_perforacion", ArrayList(positionsList_no_perforacion))
    intent.putExtra("positionsList_quemadura", ArrayList(positionsList_quemadura))

    intent.putExtra("imageCounter_fractura", imageCounter_fractura)
    intent.putExtra("imageCounter_fractura", imageCounter_hemorragia)
    intent.putExtra("imageCounter_perforacion", imageCounter_perforacion)
    intent.putExtra("imageCounter_no_perforacion", imageCounter_no_perforacion)
    intent.putExtra("imageCounter_quemadura", imageCounter_quemadura)

    setResult(Activity.RESULT_OK, intent)
    finish()
}

private fun Int.dpToPx(): Int {
    val density = resources.displayMetrics.density
    return (this * density).toInt()
}

private fun toast(message: String) {
    Toast.makeText(this, message, Toast.LENGTH_SHORT).show()
}
}
```

Evaluacion.kt

```
package NFMC


import android.content.Intent
import android.os.Bundle
import android.view.View
import android.widget.EditText
import android.widget.LinearLayout
import androidx.appcompat.app.AppCompatActivity
import com.example.prueba1.R
import java.text.SimpleDateFormat
import java.util.Calendar
import java.util.Locale

private lateinit var formularioHoraEvaluacion: LinearLayout
private lateinit var formularioFechaHoraLesion: LinearLayout
private lateinit var formularioSenales: LinearLayout
private lateinit var formularioAlergias: LinearLayout
private lateinit var formularioIntervenciones: LinearLayout

class Evaluacion : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_evaluacion)

        formularioHoraEvaluacion = findViewById(R.id.formulario_hora_evaluacion)
        formularioFechaHoraLesion = findViewById(R.id.formulario_FechaHora)
        formularioSenales = findViewById(R.id.formulario_senales)
        formularioAlergias = findViewById(R.id.formulario_alergias)
        formularioIntervenciones = findViewById(R.id.formulario_intervenciones)

        val bundle = intent.extras

        // Obtener valores del bundle para las variables

        val fecha = bundle?.getString("fecha_vuelta")
        val fechaMostrar = if (fecha.isNullOrBlank()) obtenerFechaActual() else fecha

        val hora = bundle?.getString("hora_vuelta")
        val horaMostar = if (hora.isNullOrBlank()) obtenerHoraActual() else hora

        val fecha_evaluacion = bundle?.getString("fechaEvaluacion_vuelta")
        val fecha_evaluacion_Mostrar =
            if (fecha.isNullOrBlank()) obtenerFechaActual() else fecha_evaluacion

        val horaEvaluacion = bundle?.getString("horaEvaluacion_vuelta")
        val horaEvaluacionMostrar =
            if (horaEvaluacion.isNullOrBlank()) obtenerHoraActual() else horaEvaluacion

        val senales = bundle?.getString("senales_vuelta")
        val alergias = bundle?.getString("alergias_vuelta")
        val sangrado = bundle?.getString("sangrado_vuelta")
        val airway = bundle?.getString("airway_vuelta")
        val respiracion = bundle?.getString("respiracion_vuelta")
        val circulacion = bundle?.getString("circulacion_vuelta")
        val head = bundle?.getString("head_vuelta")

        // Asignar los valores a los EditText correspondientes
        findViewById<EditText>(R.id.escribir_fecha).setText(fechaMostrar)
        findViewById<EditText>(R.id.escribir_hora).setText(horaMostar)
        findViewById<EditText>(R.id.escribir_fecha_evaluacion).setText(fecha_evaluacion_Mostrar)
        findViewById<EditText>(R.id.escribir_hora_evaluacion).setText(horaEvaluacionMostrar)
        findViewById<EditText>(R.id.escribir_senales).setText(senales)
        findViewById<EditText>(R.id.escribir_alergias).setText(alergias)
        findViewById<EditText>(R.id.escribir_sangrado).setText(sangrado)
```

```
        findViewById<EditText>(R.id.escribir_airway).setText(airway)
        findViewById<EditText>(R.id.escribir_respiracion).setText(respiracion)
        findViewById<EditText>(R.id.escribir_circulacion).setText(circulacion)
        findViewById<EditText>(R.id.escribir_head).setText(head)
    }

    override fun onBackPressed() {
        val intent = Intent(this, MenuPrincipal::class.java)

        // Obtener el texto de los EditText y pasarlos al Intent
        intent.putExtra("senales", findViewById<EditText>(R.id.escribir_senales).text.toString())
        intent.putExtra("alergias", findViewById<EditText>(R.id.escribir_alergias).text.toString())
        intent.putExtra("sangrado", findViewById<EditText>(R.id.escribir_sangrado).text.toString())
        intent.putExtra("airway", findViewById<EditText>(R.id.escribir_airway).text.toString())
        intent.putExtra(
            "respiracion",
            findViewById<EditText>(R.id.escribir_respiracion).text.toString()
        )
        intent.putExtra(
            "circulacion",
            findViewById<EditText>(R.id.escribir_circulacion).text.toString()
        )
        intent.putExtra("head", findViewById<EditText>(R.id.escribir_head).text.toString())

        // Obtener el texto de los EditText
        val fechaEditText = findViewById<EditText>(R.id.escribir_fecha)
        val horaEditText = findViewById<EditText>(R.id.escribir_hora)
        val fechaEvaluacionEditext = findViewById<EditText>(R.id.escribir_fecha_evaluacion)
        val horaEvaluacionEditText = findViewById<EditText>(R.id.escribir_hora_evaluacion)

        val fecha = fechaEditText.text.toString()
        val hora = horaEditText.text.toString()
        val fechaEvaluacion = fechaEvaluacionEditext.text.toString()
        val horaEvaluacion = horaEvaluacionEditText.text.toString()

        val fechaActual = obtenerFechaActual()

        // Validar la fecha
        if (!esFechaValida(fecha)) {
            fechaEditText.error = "Invalid date"
            return
        }

        // Validar la hora
        if (!esHoraValida(hora)) {
            horaEditText.error = "Invalid time"
            return
        }

        // Validar la fecha
        if (!esFechaValida(fechaEvaluacion)) {
            fechaEvaluacionEditext.error = "Invalid date"
            return
        }

        // Validar la hora de evaluación
        if (!esHoraValida(horaEvaluacion)) {
            horaEvaluacionEditText.error = "Invalid valuation time"
            return
        }

        // Comprobar si la fecha ingresadas son posteriores a la actual
        if (esFechaPosterior(fecha, fechaActual)) {
            // Mostrar un mensaje de error
            fechaEditText.error =
                "Please enter a date or time that is before or equal to the current date and time"
            return
        }
```

```
        // Pasar los valores al Intent si todo es correcto
        intent.putExtra("fecha", fecha)
        intent.putExtra("hora", hora)
        intent.putExtra("fechaEvaluacion", fechaEvaluacion)
        intent.putExtra("horaEvaluacion", horaEvaluacion)

        setResult(RESULT_OK, intent)
        finish()
    }

    // Métodos para mostrar los formularios al hacer clic en los botones correspondientes

    fun mostrarFormularioFechaHora(view: View) {
        toggleVisibility(formularioFechaHoraLesion)
    }

    fun mostrarFormularioEvaluacion(view: View) {
        toggleVisibility(formularioHoraEvaluacion)
    }

    fun mostrarFormularioSenales(view: View) {
        toggleVisibility(formularioSenales)
    }

    fun mostrarFormularioAlergias(view: View) {
        toggleVisibility(formularioAlergias)
    }

    fun mostrarFormularioIntervenciones(view: View) {
        toggleVisibility(formularioIntervenciones)

    }

    // Método para alternar la visibilidad del formulario
    private fun toggleVisibility(layout: LinearLayout) {
        if (layout.visibility == View.VISIBLE) {
            layout.visibility = View.GONE
        } else {
            layout.visibility = View.VISIBLE
        }
    }

    private fun obtenerHoraActual(): String {
        val formato = SimpleDateFormat("HH:mm", Locale.getDefault())
        val calendario = Calendar.getInstance()
        return formato.format(calendario.time)
    }

    private fun obtenerFechaActual(): String {
        val calendar = Calendar.getInstance()
        val dateFormat = SimpleDateFormat("dd/MM/yyyy", Locale.getDefault())
        return dateFormat.format(calendar.time)
    }

    private fun esFechaValida(fecha: String): Boolean {

        val partesFecha = fecha.split("/")
        val dia = partesFecha[0].toInt()
        val mes = partesFecha[1].toInt()
        val año = partesFecha[2].toInt()

        // Validar el rango del año, el mes y el día
        if (año < 1925 || año > 9999 || mes < 1 || mes > 12 || dia < 1) {
            return false
        }

        val diasPorMes = when (mes) {
            1, 3, 5, 7, 8, 10, 12 -> 31
            4, 6, 9, 11 -> 30
```

```
            2 -> if (esAnoBisiesto(año)) 29 else 28
            else -> return false // Mes inválido
        }

        return dia <= diasPorMes
    }

    private fun esAnoBisiesto(año: Int): Boolean {
        return año % 4 == 0 && (año % 100 != 0 || año % 400 == 0)
    }

    private fun esHoraValida(hora: String): Boolean {
        val regexHora = Regex("^([01]?[0-9]|2[0-3]):[0-5][0-9]\$") // Formato HH:MM en formato 24h

        return regexHora.matches(hora)
    }

    // Método para verificar si una fecha es posterior a otra
    private fun esFechaPosterior(fechaIngresada: String, fechaActual: String): Boolean {
        // Convertir las fechas a objetos Calendar para compararlas
        val formato = SimpleDateFormat("dd/MM/yyyy", Locale.getDefault())
        val calFechaIngresada = Calendar.getInstance()
        val calFechaActual = Calendar.getInstance()

        calFechaIngresada.time = formato.parse(fechaIngresada)
        calFechaActual.time = formato.parse(fechaActual)

        return calFechaIngresada.after(calFechaActual)
    }


}
```

ConstantesVitales.kt

```kotlin
package NFMC

import android.content.Intent
import android.os.Bundle
import android.view.View
import android.widget.AdapterView
import android.widget.ArrayAdapter
import android.widget.Button
import android.widget.EditText
import android.widget.Spinner
import androidx.appcompat.app.AppCompatActivity
import androidx.recyclerview.widget.LinearLayoutManager
import androidx.recyclerview.widget.RecyclerView
import com.example.prueba1.R
import java.text.SimpleDateFormat
import java.util.Calendar
import java.util.Locale


class ConstantesVitales : AppCompatActivity() {

    private lateinit var registros: MutableList<Registro>
    private lateinit var adapter: RegistroAdapter

    // Variables para almacenar los valores seleccionados de los Spinners
    private var escala_dolor: String? = null
    private var estado_herido: String? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_constantes_vitales)

        // Inicializar la lista de registros
        registros = mutableListOf()

        // Configurar el RecyclerView y el adaptador
        val recyclerView = findViewById<RecyclerView>(R.id.registro_constantes)
        adapter = RegistroAdapter(registros)
        recyclerView.adapter = adapter
        recyclerView.layoutManager = LinearLayoutManager(this)

        // Configurar los Spinners
        spinnerEscalaDolor()
        spinnerEstadoHerido()

        // Verificar si hay registros enviados desde MenuPrincipal
        val registrosRecibidos = intent.getParcelableArrayListExtra<Registro>("registros_vuelta")
        if (registrosRecibidos != null && registrosRecibidos.isNotEmpty()) {
            // Agregar los registros recibidos a la lista
            registros.addAll(registrosRecibidos)
            // Notificar al adaptador que los datos han cambiado
            adapter.notifyDataSetChanged()
        }

        // Obtener los valores recibidos desde la actividad MenuPrincipal
        val bundle = intent.extras
        val hora_constantes = obtenerHoraActual()
        val pulso = bundle?.getString("pulso_vuelta")
        val presion_sanguinea = bundle?.getString("presion_sanguinea_vuelta")
        val tasa_respiracion = bundle?.getString("tasa_respiracion_vuelta")
        val saturacion_oxigeno = bundle?.getString("saturacion_oxigeno_vuelta")

        // Establecer los valores en los EditText correspondientes
        findViewById<EditText>(R.id.hora_constantes).setText(hora_constantes)
        findViewById<EditText>(R.id.pulso).setText(pulso)
        findViewById<EditText>(R.id.presion_sanguinea).setText(presion_sanguinea)
```

```
        findViewById<EditText>(R.id.tasa_respiracion).setText(tasa_respiracion)
        findViewById<EditText>(R.id.saturacion_oxigeno).setText(saturacion_oxigeno)

        // Configurar el clic del botón "Guardar Registro"
        val btnGuardarRegistro = findViewById<Button>(R.id.btnGuardarRegistro)
        btnGuardarRegistro.setOnClickListener {
            val hora_constantesText: EditText = findViewById<EditText>(R.id.hora_constantes)
            val hora_constantes = hora_constantesText.text.toString()
            val isValidHora = esHoraValida(hora_constantes) || hora_constantes.isBlank()
            if (isValidHora) {
                guardarRegistro()
            } else {
                hora_constantesText.error = "Invalid time"
            }
        }
        val btnEliminarRegistro = findViewById<Button>(R.id.btnEliminarRegistro)
        btnEliminarRegistro.setOnClickListener {
            eliminarRegistro()
        }
    }

    // Función para configurar el Spinner de la Escala de Dolor
    private fun spinnerEscalaDolor() {
        val spinnerEscalaDolor = findViewById<Spinner>(R.id.spinnerEscalaDolor)
        val adapter = ArrayAdapter.createFromResource(
            this,
            R.array.opciones_escala_dolor,
            android.R.layout.simple_spinner_item
        )
        adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item)
        spinnerEscalaDolor.adapter = adapter

        // Obtener el valor seleccionado del Intent si existe
        if (intent.hasExtra("escala_dolor_vuelta")) {
            escala_dolor = intent.getStringExtra("escala_dolor_vuelta")
            val position = adapter.getPosition(escala_dolor)
            if (position != -1) {
                spinnerEscalaDolor.setSelection(position)
            }
        }

        spinnerEscalaDolor.onItemSelectedListener = object : AdapterView.OnItemSelectedListener {
            override fun onItemSelected(
                parent: AdapterView<*>,
                view: View,
                position: Int,
                id: Long
            ) {
                escala_dolor = parent.getItemAtPosition(position).toString()
            }

            override fun onNothingSelected(parent: AdapterView<*>?) {
            }
        }
    }

    // Función para configurar el Spinner del Estado del Herido
    private fun spinnerEstadoHerido() {
        val spinnerEstadoHerido = findViewById<Spinner>(R.id.spinnerEstadoHerido)
        val adapter = ArrayAdapter.createFromResource(
            this,
            R.array.opciones_estado_herido,
            android.R.layout.simple_spinner_item
        )
        adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item)
        spinnerEstadoHerido.adapter = adapter

        // Obtener el valor seleccionado del Intent si existe
        if (intent.hasExtra("estado_herido_vuelta")) {
```

```kotlin
            estado_herido = intent.getStringExtra("estado_herido_vuelta")
            val position = adapter.getPosition(estado_herido)
            if (position != -1) {
                spinnerEstadoHerido.setSelection(position)
            }
        }

        spinnerEstadoHerido.onItemSelectedListener = object : AdapterView.OnItemSelectedListener {
            override fun onItemSelected(
                parent: AdapterView<*>,
                view: View,
                position: Int,
                id: Long
            ) {
                estado_herido = parent.getItemAtPosition(position).toString()
            }

            override fun onNothingSelected(parent: AdapterView<*>?) {
            }
        }
    }

    // Método para obtener el valor seleccionado de un Spinner
    private fun obtenerValorSpinner(idSpinner: Int): String {
        val spinner = findViewById<Spinner>(idSpinner)
        return spinner.selectedItem.toString()
    }

    // Método para guardar el registro y mostrar los datos ingresados en el RecyclerView
    fun guardarRegistro() {
        val hora = findViewById<EditText>(R.id.hora_constantes).text.toString()
        val pulso = findViewById<EditText>(R.id.pulso).text.toString()
        val presionSanguinea = findViewById<EditText>(R.id.presion_sanguinea).text.toString()
        val tasaRespiracion = findViewById<EditText>(R.id.tasa_respiracion).text.toString()
        val saturacionOxigeno = findViewById<EditText>(R.id.saturacion_oxigeno).text.toString()
        val estadoHerido = obtenerValorSpinner(R.id.spinnerEstadoHerido)
        val escalaDolor = obtenerValorSpinner(R.id.spinnerEscalaDolor)


        // Crear un nuevo registro
        val registro = Registro(
            hora,
            pulso,
            presionSanguinea,
            tasaRespiracion,
            saturacionOxigeno,
            estadoHerido,
            escalaDolor
        )

        // Agregar el registro a la lista
        registros.add(registro)

        // Notificar al adaptador que se agregó un nuevo registro
        adapter.notifyDataSetChanged()
    }

    fun eliminarRegistro() {
        if (registros.isNotEmpty()) {
            registros.removeAt(registros.size - 1) // Eliminar el último registro agregado
            adapter.notifyDataSetChanged() // Notificar al adaptador que los datos han cambiado
        }
    }

    private fun obtenerHoraActual(): String {
        val formato = SimpleDateFormat("HH:mm", Locale.getDefault())
        val calendario = Calendar.getInstance()
        return formato.format(calendario.time)
    }
```

```kotlin
    override fun onBackPressed() {
        val hora_constantesText: EditText = findViewById<EditText>(R.id.hora_constantes)
        val pulsoText: EditText = findViewById<EditText>(R.id.pulso)
        val presion_sanguineaText: EditText = findViewById<EditText>(R.id.presion_sanguinea)
        val tasa_respiracionText: EditText = findViewById<EditText>(R.id.tasa_respiracion)
        val saturacion_oxigenoText: EditText = findViewById<EditText>(R.id.saturacion_oxigeno)

        val hora_constantes = hora_constantesText.text.toString()
        val pulso = pulsoText.text.toString()
        val presion_sanguinea = presion_sanguineaText.text.toString()
        val tasa_respiracion = tasa_respiracionText.text.toString()
        val saturacion_oxigeno = saturacion_oxigenoText.text.toString()

        val isValidHora = esHoraValida(hora_constantes) || hora_constantes.isBlank()

        if (isValidHora) {
            val intent = Intent(this, MenuPrincipal::class.java).apply {
                putExtra("hora_constantes", hora_constantes)
                putExtra("pulso", pulso)
                putExtra("presion_sanguinea", presion_sanguinea)
                putExtra("tasa_respiracion", tasa_respiracion)
                putExtra("saturacion_oxigeno", saturacion_oxigeno)
                putExtra("escala_dolor", escala_dolor)
                putExtra("estado_herido", estado_herido)
                putExtra("registros", registros.toTypedArray())
            }
            setResult(RESULT_OK, intent)
            super.onBackPressed()
        } else {
            hora_constantesText.error = "Invalid time"
        }
    }

    private fun esHoraValida(hora: String): Boolean {
        val regexHora = Regex("^([01]?[0-9]|2[0-3]):[0-5][0-9]\$") // Formato HH:MM en formato 24h
        return regexHora.matches(hora)
    }

}
```

```
package NFMC

import android.content.Intent
import android.os.Bundle
import android.view.View
import android.widget.EditText
import android.widget.LinearLayout
import androidx.appcompat.app.AppCompatActivity
import com.example.prueba1.R

private lateinit var formularioFluidos: LinearLayout
private lateinit var formulario_productos_sanguineos: LinearLayout
private lateinit var formularioAnalgesia: LinearLayout
private lateinit var formularioAntibioticos: LinearLayout
private lateinit var formularioOtros: LinearLayout
private lateinit var formularioTorniquete: LinearLayout
private lateinit var formularioHipotermia: LinearLayout

class Tratamiento : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_tratamiento)

        formularioFluidos = findViewById(R.id.formulario_fluidos)
        formulario_productos_sanguineos = findViewById(R.id.formulario_productos_sanguineos)
        formularioAnalgesia = findViewById(R.id.formulario_analgesia)
        formularioAntibioticos = findViewById(R.id.formulario_antibioticos)
        formularioOtros = findViewById(R.id.formulario_otros)
        formularioTorniquete = findViewById(R.id.formulario_torniquete)
        formularioHipotermia = findViewById(R.id.formulario_hipotermia)

        val bundle = intent.extras

        val nombre = bundle?.getString("nombre_vuelta")
        val volumen = bundle?.getString("volumen_vuelta")
        val via = bundle?.getString("via_vuelta")
        val hora = bundle?.getString("hora_vuelta")

        val nombre_productos_sanguineos = bundle?.getString("nombre_productos_sanguineos_vuelta")
        val volumen_productos_sanguineos = bundle?.getString("volumen_productos_sanguineos_vuelta")
        val via_productos_sanguineos = bundle?.getString("via_productos_sanguineos_vuelta")
        val hora_productos_sanguineos = bundle?.getString("hora_productos_sanguineos_vuelta")

        val nombreAnalgesia = bundle?.getString("nombreAnalgesia_vuelta")
        val dosisAnalgesia = bundle?.getString("dosisAnalgesia_vuelta")
        val viaAnalgesia = bundle?.getString("viaAnalgesia_vuelta")
        val horaAnalgesia = bundle?.getString("horaAnalgesia_vuelta")

        val nombreAntibioticos = bundle?.getString("nombreAntibioticos_vuelta")
        val dosisAntibioticos = bundle?.getString("dosisAntibioticos_vuelta")
        val viaAntibioticos = bundle?.getString("viaAntibioticos_vuelta")
        val horaAntibioticos = bundle?.getString("horaAntibioticos_vuelta")

        val nombreOtros = bundle?.getString("nombreOtros_vuelta")
        val dosisOtros = bundle?.getString("dosisOtros_vuelta")
        val viaOtros = bundle?.getString("viaOtros_vuelta")
        val horaOtros = bundle?.getString("horaOtros_vuelta")

        val ubicacionTorniquete = bundle?.getString("ubicacionTorniquete_vuelta")
        val horaTorniquete = bundle?.getString("horaTorniquete_vuelta")
        val hipotermia = bundle?.getString("hipotermia_vuelta")

        val NombreText: EditText = findViewById(R.id.escribir_nombre)
        NombreText.setText(nombre)

        val volumenText: EditText = findViewById(R.id.escribir_volumen)
        volumenText.setText(volumen)

        val viaText: EditText = findViewById(R.id.escribir_via)
        viaText.setText(via)

        val horaText: EditText = findViewById(R.id.escribir_hora)
        horaText.setText(hora)

        val NombreProductosSanguineosText: EditText =
            findViewById(R.id.escribir_nombre_productos_sanguineos)
        NombreProductosSanguineosText.setText(nombre_productos_sanguineos)

        val volumenProductosSanguineosText: EditText =
            findViewById(R.id.escribir_volumen_productos_sanguineos)
        volumenProductosSanguineosText.setText(volumen_productos_sanguineos)

        val viaProductosSanguineosText: EditText =
            findViewById(R.id.escribir_via_productos_sanguineos)
        viaProductosSanguineosText.setText(via_productos_sanguineos)

        val horaProductosSanguineosText: EditText =
            findViewById(R.id.escribir_hora_productos_sanguineos)
        horaProductosSanguineosText.setText(hora_productos_sanguineos)

        val nombreAnalgesiaText: EditText = findViewById(R.id.escribir_nombre_analgesicos)
        nombreAnalgesiaText.setText(nombreAnalgesia)

        val dosisAnalgesiaText: EditText = findViewById(R.id.escribir_dosis_analgesia)
        dosisAnalgesiaText.setText(dosisAnalgesia)

        val viaAnalgesiaText: EditText = findViewById(R.id.escribir_via_analgesia)
        viaAnalgesiaText.setText(viaAnalgesia)

        val horaAnalgesiaText: EditText = findViewById(R.id.escribir_hora_analgesia)
```

```
        horaAnalgesiaText.setText(horaAnalgesia)

        val nombreAntibioticosText: EditText = findViewById(R.id.escribir_nombre_antibioticos)
        nombreAntibioticosText.setText(nombreAntibioticos)

        val dosisAntibioticosText: EditText = findViewById(R.id.escribir_dosis_antibioticos)
        dosisAntibioticosText.setText(dosisAntibioticos)

        val viaAntibioticosText: EditText = findViewById(R.id.escribir_via_antibioticos)
        viaAntibioticosText.setText(viaAntibioticos)

        val horaAntibioticosText: EditText = findViewById(R.id.escribir_hora_antibioticos)
        horaAntibioticosText.setText(horaAntibioticos)

        val nombreOtrosText: EditText = findViewById(R.id.escribir_nombre_otros)
        nombreOtrosText.setText(nombreOtros)

        val dosisOtrosText: EditText = findViewById(R.id.escribir_dosis_otros)
        dosisOtrosText.setText(dosisOtros)

        val viaOtrosText: EditText = findViewById(R.id.escribir_via_otros)
        viaOtrosText.setText(viaOtros)

        val horaOtrosText: EditText = findViewById(R.id.escribir_hora_otros)
        horaOtrosText.setText(horaOtros)

        val ubicacionTorniqueteText: EditText = findViewById(R.id.escribir_ubicacion_torniquete)
        ubicacionTorniqueteText.setText(ubicacionTorniquete)

        val horaTorniqueteText: EditText = findViewById(R.id.escribir_hora_torniquete)
        horaTorniqueteText.setText(horaTorniquete)

        val hipotermiaText: EditText = findViewById(R.id.escribir_hipotermia)
        hipotermiaText.setText(hipotermia)

    }

    override fun onBackPressed() {
        // Validación de las horas
        val horaText: EditText = findViewById<EditText>(R.id.escribir_hora)
        val hora = horaText.text.toString()

        val hora_productos_sanguineosText: EditText = findViewById<EditText>(R.id.escribir_hora_productos_sanguineos)
        val hora_productos_sanguineos = hora_productos_sanguineosText.text.toString()

        val horaAnalgesiaText: EditText = findViewById<EditText>(R.id.escribir_hora_analgesia)
        val horaAnalgesia = horaAnalgesiaText.text.toString()

        val horaAntibioticosText: EditText = findViewById<EditText>(R.id.escribir_hora_antibioticos)
        val horaAntibioticos = horaAntibioticosText.text.toString()

        val horaOtrosText: EditText = findViewById<EditText>(R.id.escribir_hora_otros)
        val horaOtros = horaOtrosText.text.toString()

        val horaTorniqueteText: EditText = findViewById<EditText>(R.id.escribir_hora_torniquete)
        val horaTorniquete = horaTorniqueteText.text.toString()

        when {
            hora.isNotEmpty() && !esHoraValida(hora) -> horaText.error = "Invalid time"
            hora_productos_sanguineos.isNotEmpty() && !esHoraValida(hora_productos_sanguineos) -> hora_productos_sanguineosText.error = "Invalid time"
            horaAnalgesia.isNotEmpty() && !esHoraValida(horaAnalgesia) -> horaAnalgesiaText.error = "Invalid time"
            horaAntibioticos.isNotEmpty() && !esHoraValida(horaAntibioticos) -> horaAntibioticosText.error = "Invalid time"
            horaOtros.isNotEmpty() && !esHoraValida(horaOtros) -> horaOtrosText.error = "Invalid time"
            horaTorniquete.isNotEmpty() && !esHoraValida(horaTorniquete) -> horaTorniqueteText.error = "Invalid time"
            else -> {
                val intent = Intent(this, MenuPrincipal::class.java)
                // Agrega los extras al intent
                //FLUIDOS
                val nombreText: EditText = findViewById(R.id.escribir_nombre)
                val nombre = nombreText.text.toString()
                intent.putExtra("nombre", nombre)

                val volumenText: EditText = findViewById(R.id.escribir_volumen)
                val volumen = volumenText.text.toString()
                intent.putExtra("volumen", volumen)

                val viaText: EditText = findViewById<EditText>(R.id.escribir_via)
                val via = viaText.text.toString()
                intent.putExtra("via", via)

                intent.putExtra("hora", hora)

                //PRODUCTOS SANGUINEOS
                val nombre_productos_sanguineosText: EditText = findViewById(R.id.escribir_nombre_productos_sanguineos)
                val nombre_productos_sanguineos = nombre_productos_sanguineosText.text.toString()
                intent.putExtra("nombre_productos_sanguineos", nombre_productos_sanguineos)

                val volumen_productos_sanguineosText: EditText = findViewById(R.id.escribir_volumen_productos_sanguineos)
                val volumen_productos_sanguineos = volumen_productos_sanguineosText.text.toString()
                intent.putExtra("volumen_productos_sanguineos", volumen_productos_sanguineos)

                val via_productos_sanguineosText: EditText = findViewById<EditText>(R.id.escribir_via_productos_sanguineos)
                val via_productos_sanguineos = via_productos_sanguineosText.text.toString()
                intent.putExtra("via_productos_sanguineos", via_productos_sanguineos)

                intent.putExtra("hora_productos_sanguineos", hora_productos_sanguineos)

                //ANALGESICOS
                val nombreAnalgesiaText: EditText = findViewById<EditText>(R.id.escribir_nombre_analgesicos)
                val nombreAnalgesia = nombreAnalgesiaText.text.toString()
                intent.putExtra("nombreAnalgesia", nombreAnalgesia)

                val dosisAnalgesiaText: EditText = findViewById<EditText>(R.id.escribir_dosis_analgesia)
                val dosisAnalgesia = dosisAnalgesiaText.text.toString()
```

```
            intent.putExtra("dosisAnalgesia", dosisAnalgesia)

            val viaAnalgesiaText: EditText = findViewById<EditText>(R.id.escribir_via_analgesia)
            val viaAnalgesia = viaAnalgesiaText.text.toString()
            intent.putExtra("viaAnalgesia", viaAnalgesia)

            intent.putExtra("horaAnalgesia", horaAnalgesia)

            //ANTIBIOTICOS
            val nombreAntibioticosText: EditText = findViewById<EditText>(R.id.escribir_nombre_antibioticos)
            val nombreAntibioticos = nombreAntibioticosText.text.toString()
            intent.putExtra("nombreAntibioticos", nombreAntibioticos)

            val dosisAntibioticosText: EditText = findViewById<EditText>(R.id.escribir_dosis_antibioticos)
            val dosisAntibioticos = dosisAntibioticosText.text.toString()
            intent.putExtra("dosisAntibioticos", dosisAntibioticos)

            val viaAntibioticosText: EditText = findViewById<EditText>(R.id.escribir_via_antibioticos)
            val viaAntibioticos = viaAntibioticosText.text.toString()
            intent.putExtra("viaAntibioticos", viaAntibioticos)

            intent.putExtra("horaAntibioticos", horaAntibioticos)

            //OTROS
            val nombreOtrosText: EditText = findViewById<EditText>(R.id.escribir_nombre_otros)
            val nombreOtros = nombreOtrosText.text.toString()
            intent.putExtra("nombreOtros", nombreOtros)

            val dosisOtrosText: EditText = findViewById<EditText>(R.id.escribir_dosis_otros)
            val dosisOtros = dosisOtrosText.text.toString()
            intent.putExtra("dosisOtros", dosisOtros)

            val viaOtrosText: EditText = findViewById<EditText>(R.id.escribir_via_otros)
            val viaOtros = viaOtrosText.text.toString()
            intent.putExtra("viaOtros", viaOtros)

            intent.putExtra("horaOtros", horaOtros)

            //TORNIQUETE
            val ubicacionTorniqueteText: EditText = findViewById<EditText>(R.id.escribir_ubicacion_torniquete)
            val ubicacionTorniquete = ubicacionTorniqueteText.text.toString()
            intent.putExtra("ubicacionTorniquete", ubicacionTorniquete)

            intent.putExtra("horaTorniquete", horaTorniquete)

            //HIPOTERMIA
            val hipotermiaText: EditText = findViewById<EditText>(R.id.escribir_hipotermia)
            val hipotermia = hipotermiaText.text.toString()
            intent.putExtra("hipotermia", hipotermia)

            setResult(RESULT_OK, intent)
            finish()
        }
    }
}


fun mostrarFormularioFluidos(view: View) {
    toggleVisibility(formularioFluidos)
}

fun mostrarFormularioFlujoSanguineo(view: View) {
    toggleVisibility(formulario_productos_sanguineos)
}

fun mostrarFormularioAnalgesia(view: View) {
    toggleVisibility(formularioAnalgesia)
}

fun mostrarFormularioAntibioticos(view: View) {
    toggleVisibility(formularioAntibioticos)
}

fun mostrarFormularioOtros(view: View) {
    toggleVisibility(formularioOtros)
}

fun mostrarFormularioTorniquete(view: View) {
    toggleVisibility(formularioTorniquete)
}

fun mostrarFormularioHipotermia(view: View) {
    toggleVisibility(formularioHipotermia)
}

private fun esHoraValida(hora: String): Boolean {
    val regexHora = Regex("^([01]?[0-9]|2[0-3]):[0-5][0-9]\$") // Formato HH:MM en formato 24h

    return regexHora.matches(hora)
}

private fun toggleVisibility(layout: LinearLayout) {
    if (layout.visibility == View.VISIBLE) {
        layout.visibility = View.GONE
    } else {
        layout.visibility = View.VISIBLE
    }
}


}
```

Movimiento.kt

```kotlin
package NFMC

import android.content.Context
import android.content.Intent
import android.content.SharedPreferences
import android.os.Bundle
import android.view.View
import android.widget.AdapterView
import android.widget.ArrayAdapter
import android.widget.EditText
import android.widget.LinearLayout
import android.widget.Spinner
import androidx.appcompat.app.AppCompatActivity
import com.example.prueba1.R

private lateinit var formularioEvacuacion: LinearLayout
private lateinit var formularioSocorrista: LinearLayout
private lateinit var formularioDatosAdicionales: LinearLayout

private var categoria_evacuacion: String? = null


class Movimiento : AppCompatActivity() {

    // Constantes para valores preestablecidos
    companion object {
        const val KEY_NOMBRE_SOCORRISTA = "nombre_socorrista"
        const val KEY_APELLIDOS_SOCORRISTA = "apellidos_socorrista"
        const val KEY_ID_SOCORRISTA = "sexo_socorrista"

        const val DEFAULT_NOMBRE = "Pablo"
        const val DEFAULT_APELLIDOS = "García Rodríguez"
        const val DEFAULT_ID = "71903419Z"
    }

    private lateinit var sharedPreferences: SharedPreferences

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_movimiento)

        // Inicializar SharedPreferences
        sharedPreferences = getSharedPreferences("my_preferences", Context.MODE_PRIVATE)

        formularioEvacuacion = findViewById<LinearLayout>(R.id.formulario_evacuacion)
        formularioSocorrista = findViewById<LinearLayout>(R.id.formulario_socorrista)
        formularioDatosAdicionales = findViewById<LinearLayout>(R.id.formulario_datosAdicionales)

        val bundle = intent.extras

        //categoria
        spinnerCategoriaEvacuacion()

        // Obtener los valores predeterminados y establecerlos en los EditText
        val nombreSocorristaVuelta = bundle?.getString("nombreSocorrista_vuelta")
        val apellidosSocorristaVuelta = bundle?.getString("apellidosSocorrista_vuelta")
        val idSocorristaVuelta = bundle?.getString("IDsocorrista_vuelta")

        val nombreSocorrista =
            if (nombreSocorristaVuelta.isNullOrEmpty()) {
                sharedPreferences.getString(KEY_NOMBRE_SOCORRISTA, DEFAULT_NOMBRE)
            } else {
                nombreSocorristaVuelta
            }

        val apellidosSocorrista =
            if (apellidosSocorristaVuelta.isNullOrEmpty()) {
                sharedPreferences.getString(KEY_APELLIDOS_SOCORRISTA, DEFAULT_APELLIDOS)
            } else {
                apellidosSocorristaVuelta
```

```
        }

    val idSocorrista =
        if (idSocorristaVuelta.isNullOrEmpty()) {
            sharedPreferences.getString(KEY_ID_SOCORRISTA, DEFAULT_ID)
        } else {
            idSocorristaVuelta
        }

    findViewById<EditText>(R.id.escribir_nombre).setText(nombreSocorrista)
    findViewById<EditText>(R.id.escribir_apellidos).setText(apellidosSocorrista)
    findViewById<EditText>(R.id.escribir_ID).setText(idSocorrista)

    //datosAdicionales
    val datosAdicionales = bundle?.getString("datosAdicionales_vuelta")
    var datosAdicionalesText: EditText = findViewById(R.id.escribir_datos)
    datosAdicionalesText.setText(datosAdicionales)


}

override fun onBackPressed() {
    val intent = Intent(this, MenuPrincipal::class.java)

    intent.putExtra("categoria", categoria_evacuacion)
    //nombreSocorrista
    val nombreSocorristaText: EditText = findViewById<EditText>(R.id.escribir_nombre)
    val nombreSocorrista = nombreSocorristaText.text.toString()
    intent.putExtra("nombreSocorrista", nombreSocorrista)
    //apellidosSocorrista
    val apellidosSocorristaText: EditText = findViewById<EditText>(R.id.escribir_apellidos)
    val apellidosSocorrista = apellidosSocorristaText.text.toString()
    intent.putExtra("apellidosSocorrista", apellidosSocorrista)
    //IDsocorrista
    val IDsocorristaText: EditText = findViewById<EditText>(R.id.escribir_ID)
    val IDsocorrista = IDsocorristaText.text.toString()
    intent.putExtra("IDsocorrista", IDsocorrista)
    //fecha
    val datosAdicionalesText: EditText = findViewById<EditText>(R.id.escribir_datos)
    val datosAdicionales = datosAdicionalesText.text.toString()
    intent.putExtra("datosAdicionales", datosAdicionales)

    setResult(RESULT_OK, intent)
    finish()
}

// Funciones para mostrar/ocultar formularios
fun mostrarFormularioEvacuacion(view: View?) {
    toggleVisibility(formularioEvacuacion)
}

fun mostrarFormularioSocorrista(view: View?) {
    toggleVisibility(formularioSocorrista)
}

fun mostrarFormularioDatosAdicionales(view: View?) {
    toggleVisibility(formularioDatosAdicionales)
}

private fun spinnerCategoriaEvacuacion() {
    val spinnerCategoriaEvacuacion = findViewById<Spinner>(R.id.spinnerCategoriaEvacuacion)
    val adapter = ArrayAdapter.createFromResource(
        this,
        R.array.categoria_evacuacion,
        android.R.layout.simple_spinner_item
    )
    adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item)
    spinnerCategoriaEvacuacion.adapter = adapter

    // Obtener el valor seleccionado del Intent si existe
    if (intent.hasExtra("categoria_vuelta")) {
        categoria_evacuacion = intent.getStringExtra("categoria_vuelta")
        val position = adapter.getPosition(categoria_evacuacion)
```

```
            if (position != -1) {
                spinnerCategoriaEvacuacion.setSelection(position)
            }
        }

        spinnerCategoriaEvacuacion.onItemSelectedListener = object : AdapterView.OnItemSelectedListener {
            override fun onItemSelected(parent: AdapterView<*>, view: View, position: Int, id: Long) {
                categoria_evacuacion = parent.getItemAtPosition(position).toString()
            }

            override fun onNothingSelected(parent: AdapterView<*>?) {
            }
        }
    }

    private fun toggleVisibility(layout: LinearLayout?) {
        if (layout!!.visibility == View.VISIBLE) {
            layout.visibility = View.GONE
        } else {
            layout.visibility = View.VISIBLE
        }
    }

}
```

RegistroAdapter.kt

```kotlin
package NFMC

import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView
import com.example.prueba1.R

class RegistroAdapter(private val registros: MutableList<Registro>) :
    RecyclerView.Adapter<RegistroAdapter.RegistroViewHolder>() {


    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): RegistroViewHolder {
        val itemView = LayoutInflater.from(parent.context)
            .inflate(R.layout.activity_registro_constantes, parent, false)
        return RegistroViewHolder(itemView)
    }

    override fun onBindViewHolder(holder: RegistroViewHolder, position: Int) {
        val registro = registros[position]
        holder.bind(registro)
    }

    override fun getItemCount(): Int {
        return registros.size
    }

    inner class RegistroViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        fun bind(registro: Registro) {
            // Obtener referencias a las vistas del elemento de la lista
            val horaTextView: TextView = itemView.findViewById(R.id.textViewHoraValue)
            val pulsoTextView: TextView = itemView.findViewById(R.id.textViewPulsoValue)
            val presionSanguineaTextView: TextView = itemView.findViewById(R.id.textViewPresionSanguineaValue)
            val tasaRespiracionTextView: TextView = itemView.findViewById(R.id.textViewTasaRespiracionValue)
            val saturacionOxigenoTextView: TextView = itemView.findViewById(R.id.textViewSaturacionOxigenoValue)
            val estadoHeridoTextView: TextView = itemView.findViewById(R.id.textViewEstadoHeridoValue)
            val escalaDolorTextView: TextView = itemView.findViewById(R.id.textViewEscalaDolorValue)


            // Asignar los valores del registro a las vistas correspondientes
            horaTextView.text = registro.hora_constantes
            pulsoTextView.text = registro.pulso
            presionSanguineaTextView.text = registro.presionSanguinea
            tasaRespiracionTextView.text = registro.tasaRespiracion
            saturacionOxigenoTextView.text = registro.saturacionOxigeno
            estadoHeridoTextView.text = registro.estadoHerido
            escalaDolorTextView.text = registro.escalaDolor

        }
    }
}
```

Registro_constantes.kt

```kotlin
package NFMC

import android.os.Parcel
import android.os.Parcelable
import com.google.gson.annotations.SerializedName

data class Registro(
    @SerializedName("hora_constantes") var hora_constantes: String,
    @SerializedName("pulso") var pulso: String,
    @SerializedName("presion_sanguinea") var presionSanguinea: String,
    @SerializedName("tasa_respiracion") var tasaRespiracion: String,
    @SerializedName("saturacion_oxigeno") var saturacionOxigeno: String,
    @SerializedName("escala_dolor") var escalaDolor: String,
    @SerializedName("estado_herido") var estadoHerido: String,

) : Parcelable {

    // crear un objeto Registro a partir de un Parcel.
    // Lee los datos del Parcel en el mismo orden en que se escribieron en el método writeToParcel()
    // y los asigna a los atributos de la clase
    constructor(parcel: Parcel) : this(
        parcel.readString() ?: "",
        parcel.readString() ?: "",
        parcel.readString() ?: "",
        parcel.readString() ?: "",
        parcel.readString() ?: "",
        parcel.readString() ?: "",
        parcel.readString() ?: ""
    )

    //escribe los valores de los atributos de la clase en el Parcel
    // en el mismo orden que se lee en el constructor secundario
    override fun writeToParcel(parcel: Parcel, flags: Int) {
        parcel.writeString(hora_constantes)
        parcel.writeString(pulso)
        parcel.writeString(presionSanguinea)
        parcel.writeString(tasaRespiracion)
        parcel.writeString(saturacionOxigeno)
        parcel.writeString(estadoHerido)
        parcel.writeString(escalaDolor)
    }

    override fun describeContents(): Int {
        return 0
    }

    //Esta sección define un objeto CREATOR que implementa la interfaz Parcelable.Creator y
    // proporciona dos funciones para crear un array de objetos Registro
    // y para crear un objeto Registro a partir de un Parcel
    companion object CREATOR : Parcelable.Creator<Registro> {
        override fun createFromParcel(parcel: Parcel): Registro {
            return Registro(parcel)
        }

        override fun newArray(size: Int): Array<Registro?> {
            return arrayOfNulls(size)
        }
    }
}
```

APIService.kt

```
package NFMC

import retrofit2.http.Body
import retrofit2.http.POST

interface APIService {
    @POST("/")
    fun crearRegistro(@Body nfcregister: Variables): retrofit2.Call<Void>

}
```

Causa.kt
Variables.kt
APIService.kt
Evaluacion.kt
Movimiento.kt
Tratamiento.kt
DatosBasicos.kt
InicioSesion.kt
MenuPrincipal.kt
RegistroAdapter.kt
ConstantesVitales.kt
Registro_constantes.kt