Trabajo Fin de Grado
Ingeniería de las Tecnologías de la Telecomunicación

Desarrollo de aplicación NFC para manejo de dispositivos IoT: NFClift

Autor: Jaime Palomo Sivianes

Tutor: Juan Manuel Vozmediano Torres

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2022

Trabajo Fin de Grado
Ingeniería de las Tecnologías de la Telecomunicación

Desarrollo de aplicación NFC para manejo de dispositivos IoT: NFClift

Autor:
Jaime Palomo Sivianes

Tutor:
Juan Manuel Vozmediano Torres
Profesor titular

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2022

Trabajo Fin de Grado: Desarrollo de aplicación NFC para manejo de dispositivos IoT: NFClift

Autor: Jaime Palomo Sivianes

Tutor: Juan Manuel Vozmediano Torres

El tribunal nombrado para juzgar el Trabajo arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2022

El Secretario del Tribunal

A mi familia

A mis amigos

A mis compañeros

RESUMEN

En esta memoria se recogerán todos los detalles y decisiones de diseño que he ido tomando a lo largo del desarrollo del software que compone mi Trabajo Fin de Grado. Dicho software consta de dos aplicaciones Android y de un pequeño servidor que se encarga de interactuar con ambas. Estas aplicaciones serán las encargadas de gestionar llamadas a ascensores apoyándose en etiquetas NFC colocadas en éstos. Además, también se recogerá todo el proceso de pruebas sobre el software desarrollado.

ÍNDICE

Resumen vii

Índice ix

Índice de Códigos xi

Índice de Figuras xiii

1.Introducción 1
1.1. Objetivos 1
1.2. Contexto 1
1.3. Situación actual 3

2 Planificación y medios necesarios 5
2.1. Planificación temporal 5
2.2. Medios materiales necesarios 8

3 Aplicaciones 11
3.1 Aplicación de operario 11
3.1.1 Descripción funcional 11
3.1.2 Tecnologías usadas 13
3.1.3 Diseño 13
3.1.4 Flujo 18
3.2 Aplicación de usuario 19
3.2.1 Descripción funcional 19
3.2.2 Tecnologías usadas 20
3.2.3 Diseño 21
3.2.4 Flujo 25
3.3 Servidor 26
3.3.1 Descripción funcional 26
3.3.2 Tecnologías usadas 27
3.3.3 Diseño 28
3.4 Entornos de desarrollo y sus réplicas 35
3.4.1 Aplicaciones Android 35
3.4.2 Servidor 36

4 Pruebas 37
4.1 Materiales utilizados 37
4.2 Herramientas utilizadas 37
4.3 Escenario de pruebas 39
4.4 Plan de pruebas 41
4.4.1 Pruebas NFClift Operator 41
4.4.2 Pruebas NFClift 44
4.4.3 Pruebas NFClift Server 45
4.4.4 Pruebas de la solución completa 51
4.5 Réplica del entorno de pruebas 54

5 Conclusiones 57
5.1 Líneas de continuación 57

Anexo A: Guía de instalación y uso de NFClift 59

Anexo B: Guía de instalación y uso de NFClift Operator 61

Anexo C: Guía de instalación y uso de NFClift Server **65**

Referencias **67**

Enlaces de interés **69**

ÍNDICE DE CÓDIGOS

Código 1: Proceso de lectura de una etiqueta para su posterior registro 15
Código 2: Proceso de registro de una etiqueta de planta 16
Código 3: Código de la llamada para desvincular etiqueta 17
Código 4: Guardado del identificador de operario 17
Código 5: Filtrado de etiqueta NFC 21
Código 6: Obtención de mensajes y registros NDEF del intent 22
Código 7: Función que decodifica un registro NDEF dado 22
Código 8: Decodificación del objeto JSON obtenido de la respuesta del servidor 23
Código 9: Enlazado de llamada a planta con pulsación del botón 24
Código 10: Implementación del cambio de color al pulsar botón 24
Código 11: Función encargada del registro de etiquetas de planta 29
Código 12: Función encargada del registro de etiquetas de cabina 30
Código 13: Comportamiento del servidor al recibir etiqueta de cabina 31
Código 14: Función encargada de recibir las peticiones de llamadas a plantas 32
Código 15: Función encargada de desvincular etiquetas 33
Código 16: Función que inicializa la base de datos 35
Código 17: Línea a añadir al Android Manifest 40
Código 18: Archivo network_security_config.xml 40

ÍNDICE DE FIGURAS

Figura 1: Cartel gubernamental animando la higiene de manos 2

Figura 2: Símbolo del estándar NFC 3

Figura 3: Flujo de las etapas del desarrollo del proyecto 5

Figura 4: Diagrama de Gantt 7

Figura 5: Dispositivo usado durante el desarrollo 8

Figura 6: Etiquetas NFC usadas para el desarrollo 9

Figura 7: Icono de NFClift Operator 11

Figura 8: Lectura y registro de etiqueta ubicada en planta 12

Figura 9: Lectura y registro de etiqueta ubicada en cabina 12

Figura 10: Requerimiento de ID del operario 13

Figura 11: Ejemplo de contenido de una etiqueta NFC 14

Figura 12: Diagrama de flujo de aplicación de operario 18

Figura 13: Icono de NFClift 19

Figura 14: Ejemplo de botonera de un ascensor 20

Figura 15: Ejemplo de llamada exitosa a una planta 20

Figura 16: Diagrama de flujo de la aplicación de usuario 25

Figura 17: Go y su mascota oficial 27

Figura 18: Diagrama E-R. 33

Figura 19: Icono de NFC Tools 38

Figura 20: Logo de Docker 38

Figura 21: Dockerfile usado para la imagen del servidor 38

Figura 22: Logo de Postman 39

Figura 23: Docker Desktop con el contenedor del servidor corriendo 40

Figura 24: Datos en la tabla ascensores 40

Figura 25: Datos en la tabla operarios 41

Figura 26: Diagrama de red del escenario de pruebas 41

Figura 27: Prueba de éxito de lectura y registro de etiqueta en planta 42

Figura 28: Logs de prueba de éxito de lectura y registro de etiqueta en planta 42

Figura 29: Prueba de error de registro de etiqueta de planta 43

Figura 30: Interfaz de desvinculación de etiqueta 43

Figura 31: Logs generados al pulsar el botón desvincular 44

Figura 32: Prueba de éxito de lectura de etiqueta NFC con NFClift 44

Figura 33: Contenido de la etiqueta leído por NFC Tools 45

Figura 34: Petición y respuesta de caso de éxito de registro de etiqueta en planta 46

Figura 35: Estado de la base de datos después de la prueba de éxito 46

Figura 36: Petición y respuesta de caso de fallo de registro de etiqueta en planta 47

Figura 37: Estado de la base de datos después de la prueba de fallo 47

Figura 38: Petición y respuesta de registro de etiqueta de cabina para prueba de éxito 47

Figura 39: Estado de la base de datos después de la prueba 48

Figura 40: Petición y respuesta de registro de etiqueta de cabina para prueba de fallo 48

Figura 41: Estado de la base de datos después de la prueba 48

Figura 42: Petición y respuesta de llamada con etiqueta ubicada en planta para prueba de éxito 49

Figura 43: Petición y respuesta de llamada con etiqueta ubicada en cabina para prueba de éxito 49

Figura 44: *Token* recibido con la respuesta 49

Figura 45: Petición y respuesta de llamada con etiqueta inexistente 50

Figura 46: Petición y respuesta exitosa de llamada a planta de un ascensor 50

Figura 47: Petición y respuesta de una llamada a una planta no disponible. 50

Figura 48: Log en el servidor del registro de la etiqueta de cabina 51

Figura 49: Logs en el servidor de los registros de etiquetas de planta 52

Figura 50: Log generado en el servidor al escanear la etiqueta de planta 52

Figura 51: Resultado en NFClift al escanear etiqueta de planta 53

Figura 52: Log generado en el servidor al escanear la etiqueta de cabina 53

Figura 53: Botonera virtual del ascensor 53

Figura 54: Log generado en el servidor al llamar a una planta desde la botonera 54

Figura 55: Resultado de NFClift al llamar desde botonera 54

Figura 56: Lugar donde insertar la IP del servidor 55

Figura 57: Lugar donde colocar la IP del servidor al que queremos apuntar 56

Figura 58: Opción para generar una apk 56

1. INTRODUCCIÓN

1.1. Objetivos

El objetivo de este proyecto es el desarrollar una solución software para el manejo de ascensores sin que sea necesario el contacto físico directo con la botonera de éste y sustituir así la aplicación actual. Para conseguirlo, este software está desarrollado para Android y hará uso de etiquetas NFC colocadas en las cabinas y las plantas.

El proyecto está dividido en tres aplicaciones distintas, cada una con su objetivo propio:

- **NFClift:** Aplicación de usuario para Android. Ésta será la que utilizarán los usuarios de los ascensores para poder manejarlos a través de sus dispositivos móviles. Leerá etiquetas NFC previamente instaladas y configuradas para poder comunicarse con el servidor a través de peticiones HTTPS. Es la aplicación más importante puesto que es la que ofrece el propósito final de todo el proyecto.
- **NFClift Operator:** Aplicación de instalador para Android. Ésta será la que utilizarán los operarios de la empresa encargada de los ascensores para desplegar el servicio y configurarlo inicialmente. NFClift Operator será la encargada de leer las etiquetas NFC que vayamos a instalar para asociarles una acción en el servidor.
- **NFClift Server:** Aplicación de servidor encargada de gestionar las peticiones HTTPS tanto de NFClift como de NFClift Operator. A través de estas peticiones se podrán registrar las etiquetas NFC y realizar llamadas a ascensores. Para todo esto, NFClift Server dispondrá de una base de datos en la que guardará los datos de los ascensores, las etiquetas NFC con su acción vinculada y la ID de los operarios autorizados. A diferencia de las dos aplicaciones anteriores, las cuales ya están preparadas para su puesta en producción, NFClift Server es un servidor de pruebas y necesitaría integrarse debidamente en el servidor de la empresa de ascensores para su puesta en producción.

1.2. Contexto

Este proyecto ha sido ideado y desarrollado durante la pandemia de COVID-19 que ha afectado a todos los países del mundo entre 2020 y 2022. Este virus, al igual que el virus de la gripe, se propaga por vía aérea y, en menor medida, por pequeñas gotas de saliva depositadas en los objetos. Es por ello por lo que reducir el contacto físico y la higiene de manos se ha hecho más importante que nunca. De hecho, durante este periodo, las autoridades han hecho especial hincapié en esto último.

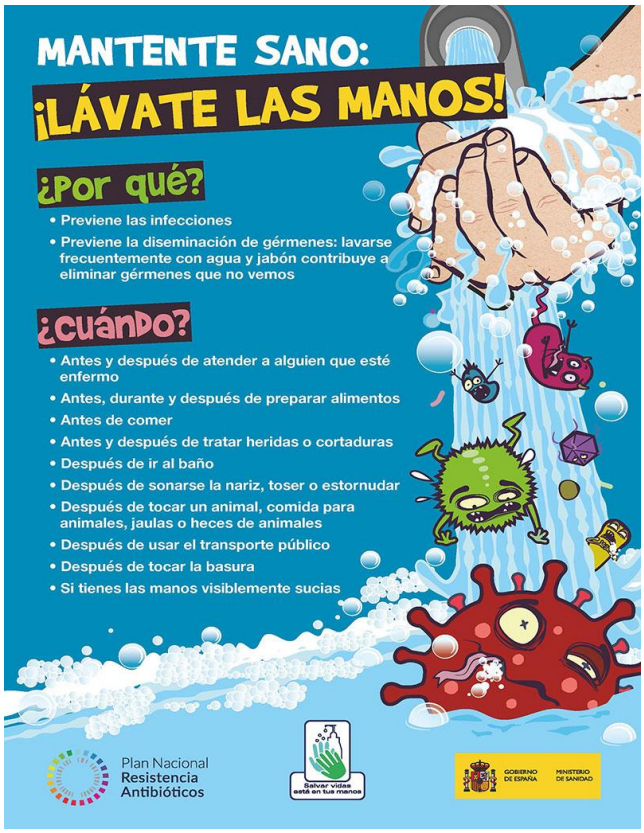


Figura 1: Cartel gubernamental animando la higiene de manos.

Dada esta circunstancia, han surgido nuevas formas de interactuar con los objetos de nuestro alrededor en espacios públicos. Estas nuevas formas de interactuar han sido posibles principalmente gracias a las nuevas tecnologías surgidas en los últimos años. Este es el caso de los códigos QR, cuyo uso se ha extendido notablemente entre la población. Esto es debido a que ofrece la posibilidad de acceder a información con solo hacer una foto. Un ejemplo de uso bastante extendido es el de redirigir a cartas alojadas en la web en establecimientos de hostelería para así evitar el compartir cartas físicas entre comensales.

En este contexto se encaja NFClift, una solución pensada para evitar la necesidad de pulsar botones físicos en los ascensores de espacios públicos y sus riesgos asociados. La tecnología de códigos QR, mencionada anteriormente, podría haber sido igualmente utilizada para la consecución de los objetivos del proyecto, pero posee ciertas desventajas que no se ajustan a nuestras necesidades. Tales desventajas son:

1) La necesidad de una buena iluminación. Para que el lector de QR de un dispositivo móvil funcione correctamente necesita tener una buena imagen del código a escanear. En nuestro caso, la buena iluminación no está siempre garantizada puesto que los ascensores pueden estar instalados en sitios como garajes y sótanos.

2) La velocidad. Para escanear un código QR siempre será necesario el iniciar una aplicación y enfocar bien el código. Esta preparación puede ser contraproducente ya que el usuario del ascensor puede experimentar una sensación de pérdida de tiempo y pulsar directamente el botón del ascensor.

Por estas razones se descartó el uso de la tecnología QR y se apostó por el uso de tecnología NFC para el desarrollo del proyecto.

La tecnología NFC (*Near-field communication*) proporciona el intercambio de información de forma inalámbrica a corto alcance y de manera casi instantánea. Esta tecnología deriva de las etiquetas RFID [1] y está regida por la ISO 18092:2013.

Figura 2: Símbolo del estándar NFC.

Estas características (y la pandemia anteriormente mencionada) han hecho que el uso del NFC se haya popularizado y extendido entre gran parte de la población. De hecho, hay estudios que calculan que el 39% de los poseedores de smartphones utilizan el pago por NFC [2] y todo apunta a que este porcentaje va a seguir creciendo. Es por ello por lo que la inclusión de lector NFC en los dispositivos móviles es cada vez más común y ha dejado de ser una característica exclusiva de los móviles de alta gama.

Todo lo expuesto anteriormente hace que nuestra solución, NFClift, sea fruto del contexto y necesidades actuales y, además, haga uso de una tecnología en auge y cada vez más común como es el NFC.

1.3. Situación actual

La situación actual que se encuentra implementada en los ascensores de la empresa propietaria es la expuesta a continuación.

Se dispone de una aplicación que controla y monitoriza ascensores. Sobre ella se ha desarrollado una aplicación Android que permite realizar llamadas al ascensor desde el dispositivo móvil. Dicha aplicación utiliza la ubicación del dispositivo para determinar cuál es el ascensor al que quiere acceder el usuario basándose en su proximidad. Una vez identificado el ascensor, el servidor le ofrece una botonera al usuario para que realice las llamadas a las plantas que desee. Cuando el usuario pulse un botón en la botonera de la aplicación se realiza una petición a la API del servidor y éste realizará internamente la llamada a través de la red GPRS disponible en los ascensores.

El funcionamiento expuesto adolece de las siguientes desventajas:

- El uso de la ubicación para designar el ascensor que va a usar el usuario basado en la proximidad a éste puede ser bastante inconsistente. Por ejemplo, si en un edificio hay instaladas más de una cabina, la precisión de las medidas de la ubicación no es lo suficientemente buena para identificar de forma inequívoca la cabina que el usuario desea utilizar. Otro ejemplo de inconsistencia se podría dar en el caso de que el dispositivo tuviera algún defecto a la hora de calcular la ubicación y tuviera un margen de error excesivamente grande.
- La ubicación puede fallar en localizaciones como garajes y sótanos.
- El hecho de tener que enviar nuestra ubicación puede hacer que usuarios rechacen su uso debido al temor de compartir datos personales.
- Mantener la ubicación activada reduce considerablemente la batería de los dispositivos móviles.
- Posible brecha de seguridad si un usuario malintencionado falsea su ubicación para utilizar ascensores a distancia.

A pesar de estas desventajas, la solución actual también posee sus ventajas:

- No necesita de despliegue previo en campo puesto que la única herramienta necesaria para su

funcionamiento es el dispositivo móvil del usuario.

- Funciona en prácticamente en cualquier dispositivo puesto que el servicio de ubicación lleva implantado en casi todos los dispositivos móviles desde hace años.
- Solo hace falta el desarrollo de la aplicación de usuario y un pequeño ajuste en el servidor para que la solución funcione.

Dadas las desventajas y ventajas expuestas, se ha decidido el crear una solución que palie la principal desventaja de la que hay actualmente. Nuestra solución será capaz de identificar de manera inequívoca el ascensor que el usuario quiere utilizar.

2 PLANIFICACIÓN Y MEDIOS NECESARIOS

2.1. Planificación temporal

La planificación temporal de un proyecto consiste en hacer un cálculo aproximado del tiempo que necesitará invertirse en él hasta terminarlo. Una de las formas de hacer esto es estimar el tiempo que se invertirá en cada etapa del proyecto en horas. En mi caso, debido a que el tiempo que le puedo dedicar al proyecto es irregular y sin una rutina concreta me es difícil hacer un cálculo de horas. Es por eso por lo que voy a realizar la estimación en quincenas.

El desarrollo del proyecto seguirá las siguientes etapas:

1. **Definición de objetivo y alcance:** En esta fase se definen las funcionalidades de las aplicaciones y las condiciones que deben cumplir para considerarlas terminadas.
2. **Estudio de herramientas y tecnologías:** Aunque ya había programado antes para Android y estaba familiarizado con Java, he necesitado repasar conceptos. Por otro lado, nunca antes había trabajado con la tecnología NFC y he tenido que aprender cómo leer y escribir sus etiquetas.
3. **Diseño del Sistema:** En esta fase se toman las decisiones de diseño necesarias antes de comenzar el desarrollo de las aplicaciones.
4. **Desarrollo:** En esta etapa codificaremos las aplicaciones y seguiremos el diseño propuesto en la etapa anterior.
5. **Pruebas:** Una vez desarrolladas las aplicaciones, en esta fase se hacen las pruebas necesarias para asegurar el buen funcionamiento de éstas y la consecución del objetivo.
6. **Documentación:** Como último paso, se desarrolla la memoria que recoge el proceso de desarrollo del proyecto.

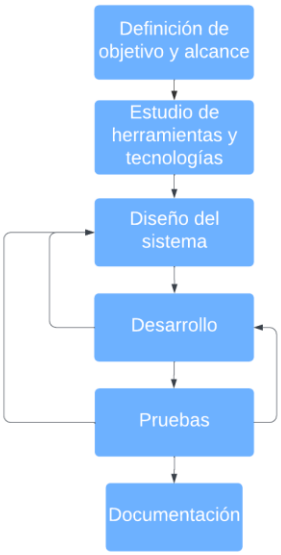


Figura 3: Flujo de las etapas del desarrollo del proyecto.

Aunque las etapas están bastante bien diferenciadas entre sí, por necesidades del desarrollo o por imprevistos el flujo puede no ser descendente. Esto quiere decir que puede que haya que volver a fases anteriores para solucionar errores o rediseñar.

Como he comentado antes, estimaré la planificación temporal en quincenas. Por ello la mejor forma de representarlo es en un diagrama de Gantt. El diagrama de Gantt es una herramienta gráfica que muestra el tiempo previsto que se le va a dedicar a un conjunto de tareas en un tiempo determinado. En la página siguiente se expone un diagrama de Gantt del proyecto.

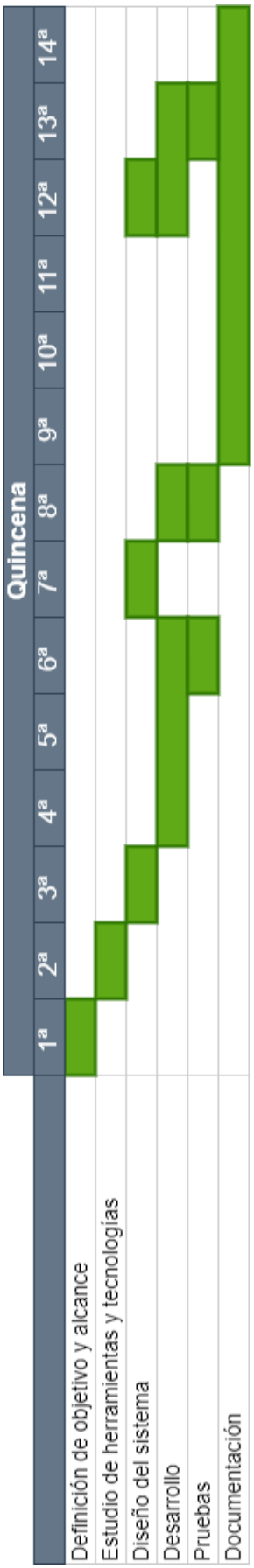


Figura 4: Diagrama de Gantt.

2.2. Medios materiales necesarios

Como proyecto de desarrollo software que es, el único material estrictamente necesario para su codificación es un ordenador con Android Studio instalado para el desarrollo de las aplicaciones Android y un entorno de desarrollo para la codificación del servidor.

Aunque esto sea así, la realidad es que el acceso a más materiales nos facilitará enormemente tanto el desarrollo como las pruebas posteriores que deseemos realizar. A continuación, se detallan los medios mínimos que creo que son indispensables para la correcta realización del proyecto y las variantes concretas que he utilizado:

- **Un ordenador compatible con Android Studio** [3]. Puesto que el núcleo del proyecto son las dos aplicaciones Android, este entorno de desarrollo resulta indispensable para su codificación. En mi caso he usado un ordenador con Windows 10 instalado.
- **Dispositivo Android con lector NFC.** Dado que las aplicaciones están desarrolladas para Android necesitamos un dispositivo con dicho sistema operativo. Ambas aplicaciones hacen uso de la tecnología NFC: realizaremos lecturas de etiquetas NFC en las dos aplicaciones. Por lo tanto, este elemento es imprescindible para la realización de pruebas una vez terminado el desarrollo de las aplicaciones. En mi caso he usado un dispositivo con Android 9.0 y chip NFC; concretamente un Motorola moto g6 plus.

Figura 5: Dispositivo usado durante el desarrollo.

- **Etiquetas NFC NTAG210 o superior.** La categoría NTAG210 permite almacenar hasta 48 bytes de información, suficiente para el uso que le daremos. Aunque en caso de despliegue de la solución en ascensores reales lo conveniente sería usar etiquetas ya escritas de fábrica con identificadores únicos y no escribibles, para el momento del desarrollo lo ideal es usar etiquetas NFC reescribibles. En mi caso he usado etiquetas NFC de categoría NTAG215 reescribibles.

Figura 6: Etiquetas NFC usadas para el desarrollo.

3 APLICACIONES

En este apartado definiremos por completo la solución de NFClift. Describiremos al detalle cada una de las funciones que deberá cumplir cada aplicación. También se explicarán todas las decisiones de diseño tomadas antes y durante el desarrollo de las mismas. Por último, para que quede más claro el funcionamiento de las aplicaciones Android, se detallará el flujo seguido por cada una.

Como cada aplicación posee unas funciones bien definidas y diferenciadas del resto, detallaremos las aplicaciones por separado. Primero empezaremos por la aplicación de operario puesto que a la hora de desplegar el servicio sería la primera en usarse. Después continuaremos con la aplicación de usuario y, por último, la de servidor.

Al final del apartado, para permitir al lector la posibilidad del mantenimiento del código y su mejora, se incluyen unas descripciones y unas pequeñas instrucciones para la réplica de los entornos de desarrollo que he usado.

3.1 Aplicación de operario

- **Nombre:** NFClift Operator.
- **Lenguaje:** Java.
- **Sistema operativo:** Android.
- **Versión de Android mínima:** Android 7.0 Nougat (SDK 24).
- **IDE utilizado:** Android Studio.
- **Descripción:** NFClift Operator es la aplicación que los instaladores de la empresa de ascensores deberán tener instalada en sus dispositivos Android para el despliegue del servicio.

Figura 7: Icono de NFClift Operator.

3.1.1 Descripción funcional

Esta aplicación es la encargada de ofrecer al operador una manera lo más sencilla posible de desplegar el servicio de NFClift en los ascensores. Para ello, cumple las siguientes funciones:

- **Lectura de etiquetas NFC.** El operador deberá leer el contenido de las etiquetas NFC con su dispositivo móvil para obtener el identificador escrito en éstas. Las etiquetas vendrán escritas de fábrica con identificadores numéricos únicos y con tipo *nfc/lift* (más adelante se explicará esto último).
- **Registro de las etiquetas en el servidor.** Además de la lectura de la etiqueta, la aplicación también debe mandar una petición HTTPS al servidor con el identificador de la propia etiqueta y los datos necesarios para que se pueda vincular una acción a ésta. Dependiendo de si la etiqueta va colocada en la planta o en la propia cabina del ascensor los datos necesarios para su registro serán distintos

  En el caso de que la etiqueta vaya a ser colocada en la planta nos hará falta enviar la planta en la que

estará instalada y su número RAE junto con el identificador de la etiqueta.

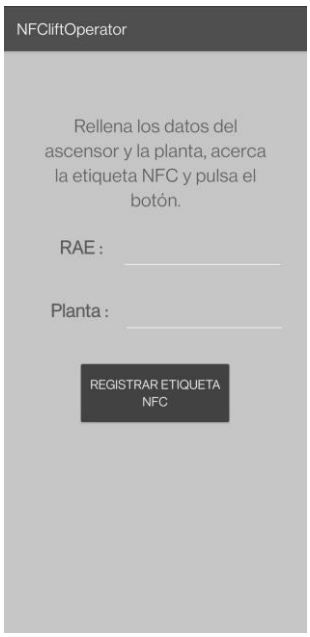


Figura 8: Lectura y registro de etiqueta ubicada en planta.

En caso de que la etiqueta sea para colocarla dentro de la cabina del ascensor, lo único que se deberá enviar junto con su identificador es su número RAE.

Figura 9: Lectura y registro de etiqueta ubicada en cabina.

- **Desvinculación de etiquetas.** Para poder subsanar fallos humanos por parte del operario al registrar etiquetas, la aplicación también contará con una funcionalidad que permita desvincular una etiqueta de su acción. Se realizará leyendo el identificador de la etiqueta y enviándolo en una petición HTTPS al servidor para que la borre de su base de datos.

- **Autenticación del operario.** Debemos asegurarnos de que no cualquier persona con acceso a la aplicación puede registrar etiquetas. Por ello, al iniciarse la aplicación se requerirá una ID de operario que será enviada en todas las peticiones HTTPS al servidor. Si dicho ID no está presente en la base de datos del servidor, la petición será automáticamente rechazada.

Figura 10: Requerimiento de ID del operario.

3.1.2 Tecnologías usadas

Al ser una aplicación Android para su codificación pude elegir entre Java y Kotlin. Aunque Kotlin hoy en día es un lenguaje en auge y muy interesante, nunca he trabajado con él. Aprender Kotlin mientras desarrollaba el proyecto le habría añadido una dificultad extra innecesaria. Es por ello por lo que decidí desarrollar la aplicación en Java.

La principal tecnología usada por esta aplicación y alrededor de la que gira todo el proyecto es el NFC. Como se ha explicado anteriormente, la tecnología NFC permite el intercambio de datos entre dos dispositivos de forma inalámbrica de corto alcance. En esta aplicación haremos uso de esta tecnología solamente para leer.

3.1.3 Diseño

Formato de etiquetas NFC

A la hora de afrontar el diseño de la aplicación, el formato de las etiquetas NFC es una de las decisiones de diseño más importante puesto que el resto de la aplicación (y aplicaciones) dependen de ella. En este caso el formato de las etiquetas NFC será un simple número y no serán reescribibles.

Este número será catalogado como el ID de la etiqueta NFC y vendrá escrito de fábrica en la etiqueta. En el momento del registro de la etiqueta el ID será enviado al servidor junto con otros datos para asignarle una acción en el servidor

La acción que se le asigne a la etiqueta dependerá de donde vaya colocada: en cabina o en planta. Si la etiqueta es de planta, la acción en servidor asignada a ella será realizar una llamada al ascensor para que se desplace a la planta en la que está instalada. En cambio, si la etiqueta es de cabina la acción será devolver los datos del ascensor que posee el servidor en un objeto *json*.

En un primer diseño del formato de las etiquetas, éstas contenían directamente los datos de los ascensores en

formato plano. Esto permitía que desde la aplicación de usuario se pudiera llamar directamente a la planta que se quisiera. Como todos los datos necesarios para realizar llamadas al servidor estaban grabados en las etiquetas no hacía falta un registro previo de las etiquetas por parte del operario. Aunque en un primer momento parecía una buena solución, esto hacía que los *endpoints* del servidor estuvieran demasiado expuestos. Además, esto permitía que un usuario malintencionado modificara las etiquetas para que se mandaran peticiones erróneas al servidor. Por estas razones acabé decantándome por el formato que finalmente se ha adoptado.

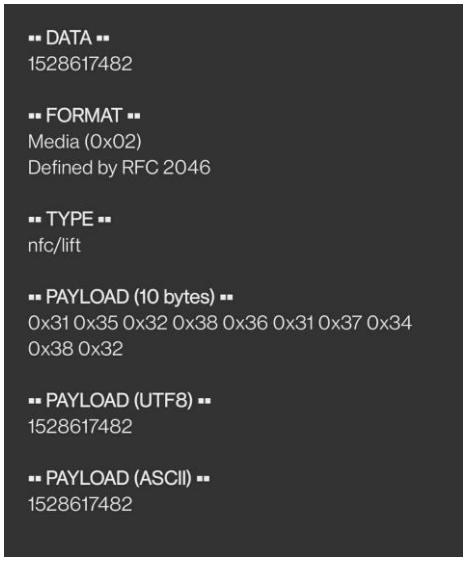


Figura 11: Ejemplo de contenido de una etiqueta NFC.

Las etiquetas NFC poseen un registro de 3 bytes llamado TNF (formato de nombre de tipo). Este registro se encarga de indicarle al sistema Android cómo tiene que interpretar el registro siguiente, llamado campo de tipo. Según el valor asignado al TNF nuestro sistema Android sabrá cómo actuar con el resto de la carga de la etiqueta NFC. En nuestro caso, todas las etiquetas usadas tendrán TNF igual a 0x02, lo cual indica que el campo de tipo es un valor especial.

Como podemos observar en la figura 11, el campo de tipo de la etiqueta NFC no es un tipo estándar, sino que es uno personalizado. Esto está hecho así para que cuando se acerque el dispositivo con la aplicación de usuario instalada, automáticamente se envíe un *intent* a dicha aplicación con los datos de la etiqueta escaneada. El cómo se realiza el filtrado del *intent* vendrá detallado en el apartado de diseño correspondiente a la aplicación de usuario.

Lectura y registro de etiquetas NFC

La función más importante que desempeña la aplicación de operario es la de la lectura y el registro de las etiquetas NFC puesto que es lo único necesario para desplegar el servicio y que la aplicación de usuario sea funcional. La lectura y el registro están englobados en el mismo apartado porque está pensado para que en el momento en el que se lee el ID en la etiqueta se mande una petición al servidor registrándola. El operario tan sólo tendrá que rellenar los datos, acercar el dispositivo móvil a la etiqueta NFC y pulsar el botón que está en pantalla.

En los siguientes bloques de código se puede ver cómo se gestiona en el momento de la lectura y registro de una etiqueta que va a ser colocada en planta. Primero se guarda en una variable el número leído de la etiqueta, el cual es su identificador. Acto seguido se monta la *url* a la que irá dirigida la petición de registro de la etiqueta. Esta *url* contendrá los datos introducidos por el operario: RAE del ascensor y planta en la que irá

colocada la etiqueta; el ID anteriormente leído y el identificador del operario. A través de un *toast* se indicará si la acción ha tenido éxito o no.

```
protected void onNewIntent(Intent intent){
    super.onNewIntent(intent);
    setIntent(intent);
    //Filtramos intent por descubrimiento de etiqueta NFC
    if (NfcAdapter.ACTION_TAG_DISCOVERED.equals(intent.getAction())){
        TagNFC.myTag = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);

        Parcelable[] rawMessages = //Obtenemos los mensajes NDEF de la
etiqueta

intent.getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES);
        if (rawMessages != null){
            NdefMessage[] messages = new NdefMessage[rawMessages.length];
            //Por cada mensaje leemos sus registros
            for (int i = 0; i < rawMessages.length; i++) {
                messages[i] = (NdefMessage) rawMessages[i];
                records = messages[i].getRecords();
                for (NdefRecord record : records) {
                    try{
                        Log.d("I", "Tipo de la etiqueta
"+record.toMimeType());
                        //Comprobamos que el tipo de la etiqueta es el
correcto
                        if (!record.toMimeType().equals("nfc/lift")){
                            Toast.makeText(this, "La etiqueta no es
correcta", Toast.LENGTH_SHORT).show();
                            return;
                        }

                        //Obtenemos el ID de la etiqueta
                        message = readRecord(record);
                        Log.d("I", "Contenido de la etiqueta en planta
activity: "+message);
                        Utils.id_tag = Integer.parseInt(message);
                    } catch(UnsupportedEncodingException err){}
                    catch(NumberFormatException e){
                        Log.d("E", "Id de la etiqueta no es un número");
                    }
                }
            }
        }
    }
}
```

Código 1: Proceso de lectura de una etiqueta para su posterior registro.

```
public void onClick(View view) {
    try{
        if(Utils.id_tag==0){
            Toast.makeText(view.getContext(), NO_TAG,
Toast.LENGTH_SHORT).show();
        }else{
            if(!isPositiveNumber(mEditTextRae.getText().toString() )){
                Toast.makeText(view.getContext(), "Rae debe ser un número
positivo", Toast.LENGTH_SHORT).show();
                Log.d("I", "Número RAE inválido");
            }else{
                //Creamos variables para obtener RAE y planta de lo
introducido por el operario
                String rae = mEditTextRae.getText().toString();
                String planta = mEditTextPlanta.getText().toString();

                Utils.lastRae = rae;

                //Generamos número aleatorio que será el ID de la etiqueta
                Log.d("I", "ID de la etiqueta: " + Utils.id_tag);

                OkHttpClient client = new OkHttpClient();

                //Montamos la url con la que registraremos la etiqueta en el
servidor
                String url =
Utils.urlBaseApiMock+"/registrartag/planta/"+rae+"/"+planta+"/"+Utils.id_tag+
"/"+Utils.id_op;

                RequestBody body = new FormBody.Builder()
                        .build();

                Request request = new Request.Builder()
                        .url(url)
                        .put(body)
                        .build();

                Log.d("I", "Request: " + request.toString());

                client.newCall(request).enqueue(new Callback() {…});
            }
        }
    }catch(Exception e){…}
}
```

Código 2: Proceso de registro de una etiqueta de planta.

En el caso de que la etiqueta vaya ubicada en la cabina del ascensor el proceso es análogo, las únicas diferencias son que el operario solo tendrá que introducir el número RAE del ascensor y el *endpoint* al que irá dirigida la aplicación será otro.

En las peticiones para el registro de las etiquetas, y en general en las dos aplicaciones Android, he usado la librería *OkHttp*. Esta librería ofrece un cliente HTTP sencillo que nos permite realizar las llamadas al servidor de una manera simplificada.

Desvinculación de etiquetas

Para dotar a los operarios de la posibilidad de corregir fallos humanos en los registros de etiquetas, se ha añadido la funcionalidad de desvincular la acción asociada a la etiqueta en el servidor. Aunque a la hora de registrar una etiqueta se comprueba que los datos sean coherentes, es posible que el operario registre por error etiquetas que no necesite. Para evitar tener que desechar esa etiqueta se ha añadido la posibilidad de borrarla de la base de datos del servidor y así poder registrarla en un futuro. Para ello, el operario tendrá que acercar el dispositivo a la etiqueta y pulsar un botón que enviará su identificador a un *endpoint* de la API del servidor que se encargará de borrarlo de su base de datos.

```
OkHttpClient client = new OkHttpClient();

String url =
Utils.urlBaseApiMock+"/desvinculartag/"+Utils.id_tag+"/"+Utils.id_op;
Log.d("I", url);

RequestBody body = new FormBody.Builder()
        .build();


Request request = new Request.Builder()
        .url(url)
        .delete(body)
        .build();
client.newCall(request).enqueue(new Callback() {…}
```

Código 3: Código de la llamada para desvincular etiqueta.

Identificación del operario

Como se ha explicado en el apartado 3.1.1, la identificación del operario es necesaria para que los registros de etiquetas solo lo hagan las personas autorizadas. Esto se ha conseguido con un identificador de operario. Este identificador será un número y será exigido por la aplicación nada más se inicie. Este número se guardará en una variable global común a todas las *activities* de la aplicación. En el momento en el que se vaya a realizar cualquier petición al servidor, el identificador irá incluido en ella y sólo resultará satisfactoria si ese identificador se encuentra en la base de datos del servidor.

```
public void onClick(View view) {

    try {
        //Guardamos el identificador en una variable global
        Utils.id_op = Integer.parseInt(mEditTextIdOp.getText().toString());
        Intent intentInicio = new Intent(view.getContext(),
EditarActivity.class);
        startActivity(intentInicio);
    }catch(NumberFormatException ex){
        Toast.makeText(view.getContext(), "ID del operador debe ser un
número", Toast.LENGTH_SHORT).show();
    }
}
```

Código 4: Guardado del identificador de operario.

3.1.4 Flujo

A continuación, se expone el diagrama de flujo correspondiente a la aplicación de operario:

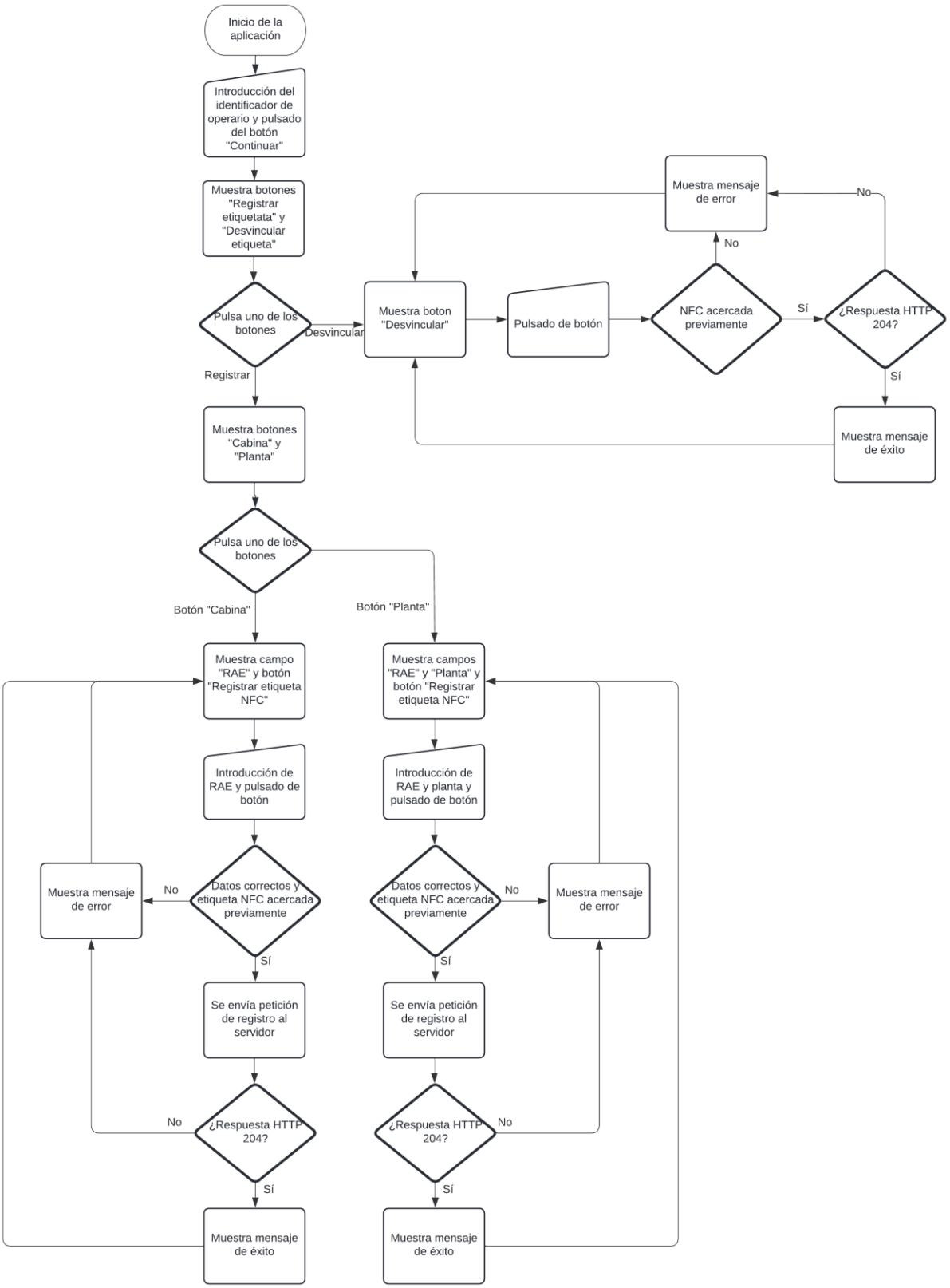


Figura 12: Diagrama de flujo de aplicación de operario.

3.2 Aplicación de usuario

- **Nombre:** NFClift.
- **Lenguaje:** Java.
- **Sistema operativo:** Android.
- **Versión de Android mínima:** Android 7.0 Nougat (SDK 24).
- **IDE utilizado:** Android Studio.
- **Descripción:** NFClift es la aplicación que usarán los usuarios para controlar los ascensores sin necesidad de tocar físicamente las botoneras de éstos.

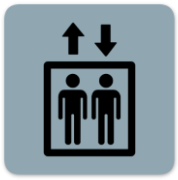


Figura 13: Icono de NFClift.

3.2.1 Descripción funcional

Esta aplicación es la encargada de ofrecer una interfaz de usuario para las llamadas a las plantas de los ascensores y mantener una comunicación con el servidor encargado de ejecutarlas. Además, será la aplicación que usará las etiquetas NFC previamente configuradas con la aplicación de operario. Para lograr lo anterior, la aplicación cumplirá las siguientes funciones:

- **Lanzamiento automático de la aplicación.** Para que nuestra aplicación sea útil para el usuario, debemos hacer que al acercar el dispositivo móvil a las etiquetas NFC se inicie automáticamente. Lo primero que realizará la aplicación una vez lanzada así será mandar una petición al servidor con el identificador de la etiqueta que hemos escaneado. Lo que suceda a continuación dependerá de la respuesta del servidor.
- **Lectura de las etiquetas NFC.** La aplicación debe ser capaz de leer los identificadores que están dentro de las etiquetas NFC.
- **Mostrar botonera del ascensor.** Nuestra aplicación debe mostrar una interfaz con los botones correspondientes a las paradas disponibles en el ascensor después de escanear la etiqueta NFC situada en la cabina.

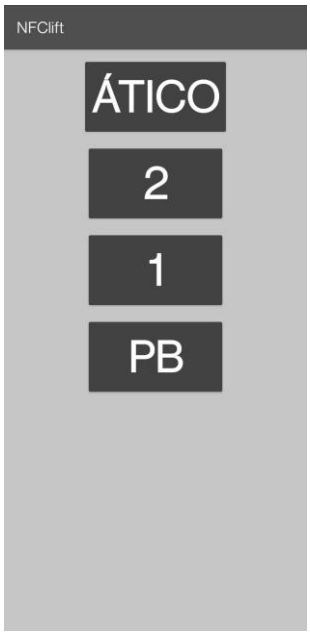


Figura 14: Ejemplo de botonera de un ascensor.

- **Realizar llamadas a planta a través de la API del servidor.** Los botones anteriormente mencionados mandarán peticiones HTTPS al servidor encargado de realizar llamadas a los ascensores. Esto hará que al pulsar el botón de la planta a la que queramos ir el ascensor reciba una llamada a dicha planta.

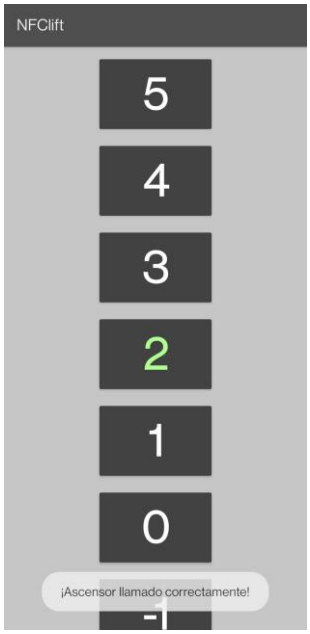


Figura 15: Ejemplo de llamada exitosa a una planta.

3.2.2 Tecnologías usadas

Al igual que en la aplicación de operario, la de usuario está codificada en Java y hará uso de la tecnología NFC. Al igual que en la otra aplicación, haremos uso de la tecnología NFC para la lectura de etiquetas.

3.2.3 Diseño

Lanzamiento automático de la aplicación

Como he expuesto en el apartado 3.2.1, el lanzamiento automático de NFClift al acercar el móvil a la etiqueta es esencial para que sea una aplicación útil para el usuario. Aquí entra en juego el diseño del formato de la etiqueta, explicado en el apartado 3.1.3. El registro TNF igual a 0x02 indicará que el registro de tipo será especial. Esto significa que cuando el dispositivo detecte la etiqueta con este TNF se fijará en el registro de tipo para saber a qué aplicación debe enviar el *intent* con dicha etiqueta. En nuestro caso el registro de tipo contendrá una cadena de caracteres igual a *nfc/lift*.

```
<activity
    android:name=".NFCnear"
    android:theme="@android:style/Theme.NoDisplay"
    android:exported="true">
    <intent-filter>
        <category android:name="android.intent.category.LAUNCHER" />

        <action android:name="android.nfc.action.NDEF_DISCOVERED" />

        <category android:name="android.intent.category.DEFAULT" />

        <data android:mimeType="nfc/lift" />
    </intent-filter>
</activity>
```

Código 5: Filtrado de etiqueta NFC.

En el bloque de código anterior se muestra un fragmento del archivo cómo se filtra la etiqueta NFC para lanzar la actividad que deseamos al escanear una etiqueta NFC con el tipo correcto. Este bloque de código se encuentra dentro del archivo *AndroidManifest.xml* y hará que el *intent* con la etiqueta sea enviado a la actividad NFCnear, la cual no tiene interfaz gráfica. Esto enlaza con el siguiente apartado, la lectura de las etiquetas NFC.

Lectura de etiquetas NFC y petición a servidor

Existen diferentes tipos de etiquetas NFC y distintos estándares para controlarlas. En el caso de Android, la mayor compatibilidad posible se obtiene utilizando el estándar NDEF, definido por el Foro NFC, el cual usaremos en nuestro caso. Los datos NDEF se encuentran dentro de uno o más mensajes (clase *NdefMessage* en Java*)* que contienen uno o varios registros (clase *NdefRecord*).

En el ámbito de este proyecto, las etiquetas tan solo contendrán un mensaje (*NdefMessage*) y unregistro (*NdefRecord*) en el cual vendrá encapsulado el ID de la etiqueta que usaremos para comunicarnos con el servidor. Aunque solo tengamos un mensaje, los analizaremos todos para evitar posibles fallos.

El mensaje NDEF vendrá encapsulado dentro del *intent* que inicia la actividad NFCnear, que será la encargada de la lectura de las etiquetas. En el siguiente bloque de código se puede observar cómo obtenemos todos los posibles mensajes presentes en la etiqueta y todos los registros dentro de éstos.

```
//Comprobamos el contenido del intent
if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals(intent.getAction())) {
    Log.d("I", "Etiqueta NFC de tipo nfc/lift leída");
    Parcelable[] rawMessages = //Obtenemos los mensajes NDEF de la etiqueta
            intent.getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES);
    if (rawMessages != null) {
        NdefMessage[] messages = new NdefMessage[rawMessages.length];
        //Por cada mensaje leemos sus registros
        for (int i = 0; i < rawMessages.length; i++) {
            messages[i] = (NdefMessage) rawMessages[i];
            records = messages[i].getRecords();
            for (NdefRecord record : records){…}
        }
    }
}
```

Código 6: Obtención de mensajes y registros NDEF del intent.

Una vez hemos obtenido el registro, deberemos decodificarlo para obtener el ID de la etiqueta. La función usada para esta decodificación es la siguiente.

```
/*Función que recibiendo un NdefRecord coge el mensaje en bytes y lo
decodifica en UTF-8
* o UTF-16*/
private String readRecord(NdefRecord record) throws
UnsupportedEncodingException {

    byte[] byteRecord = record.getPayload();
    String textEncoding = ((byteRecord[0] & 128) == 0) ? "UTF-8" : "UTF-16";

    return new String(byteRecord, 0, byteRecord.length , textEncoding);
}
```

Código 7: Función que decodifica un registro NDEF dado.

En este punto ya hemos conseguido leer la etiqueta NFC y obtener el ID que necesitamos para comunicarnos con el servidor. Con este identificador realizaremos una llamada HTTPS de método PUT al servidor. La respuesta del servidor dependerá de dónde va colocada la etiqueta NFC:

- En caso de que la etiqueta vaya colocada en planta, el servidor realizará una llamada al ascensor a dicha planta y devolverá una respuesta HTTP 204 *No Content*. En este punto, la aplicación avisará al usuario de que el ascensor ha sido llamado correctamente y le instará a escanear la etiqueta NFC colocada en cabina.
- Si la etiqueta NFC era de cabina, el servidor devolverá una respuesta HTTP 200 *OK* junto con un objeto JSON que encapsula los datos del ascensor: número RAE, número de paradas, textos de las paradas, paradas bloqueadas, etc. Esta información será utilizada para el siguiente apartado que se expondrá a continuación; la creación de la interfaz de botonera del ascensor. Este objeto *json* que nos es devuelto coincide con la estructura de datos que ya está implementada en el servidor de la compañía de ascensores para poder así reutilizar parte de la situación actual.

```
else if(response.code() == 200) {
    String resStr = response.body().string();
    String aux =  response.header("set-cookie");
    aux = aux.split(";")[0];
    aux = aux.substring(6);
    LiftInfo.authToken = aux;

    Gson gson = new Gson();

    /*Guardamos los datos obtenidos de la respuesta
    * del servidor en una clase global */
    LiftInfo.lift = gson.fromJson(resStr, Lift.class);

    Intent intentCabin = new Intent(NFCnear.this, CabinActivity.class);
    startActivity(intentCabin);

}
```

Código 8: Decodificación del objeto JSON obtenido de la respuesta del servidor.

En el anterior bloque de código, a parte del guardado del objeto *json*, también podemos ver el guardado del *token* de autorización que nos devuelve el servidor al haber escaneado una etiqueta ubicada en cabina. Utilizaremos este *token* para que el servidor nos autentique cuando realicemos llamadas desde la botonera.

El objeto *json* encapsulado en la respuesta del servidor se convierte a una clase de java gracias a la librería *gson*, la cual nos permite montar una clase definida previamente con solo una cadena de texto conteniendo dicho objeto *json.*

Botonera y llamadas a planta

Para mostrar la botonera del ascensor en nuestro dispositivo móvil previamente hemos debido escanear la etiqueta NFC colocada en la cabina del mismo. Como se ha explicado en el apartado anterior, al hacerlo obtendremos los datos del ascensor. Con estos datos crearemos una lista de botones que contengan los textos de las paradas que posee el ascensor, como se muestra en la Figura 14. Estos botones al ser pulsados realizarán una petición HTTPS al servidor conteniendo el número RAE del ascensor y el texto del botón pulsado. En el siguiente bloque de código se puede ver cómo se le ha añadido esta función a cada botón de la lista. Además, también se envía el *token* obtenido antes para autenticarnos.

```
public void onClick(View view) {

    OkHttpClient client = new OkHttpClient();

    //Creamos llamada a planta con el RAE y el número del botón
    String url =
LiftInfo.urlBaseApiMock+"/"+LiftInfo.lift.getRae()+"/stops/"+viewHolder.getBu
tton().getText()+"/call";

    RequestBody body = new FormBody.Builder()
            .build();

    Request request = new Request.Builder()
            .header("token", LiftInfo.authToken)
            .url(url)
            .put(body)
            .build();
    client.newCall(request).enqueue(new Callback() {…});
}
```

Código 9: Enlazado de llamada a planta con pulsación del botón.

Para que el usuario que use la aplicación obtenga una experiencia más real y sea fácilmente distinguible el haber llamado a una planta, se ha añadido al pulsado de botón un cambio de color. Cuando el usuario pulse un botón y la llamada haya sido satisfactoria, éste se volverá de color verde durante 20 segundos (duración arbitraria) para que se sepa que ha sido pulsado. Esto lo podemos ver en la Figura 15. En el bloque de código siguiente se muestra su implementación.

```
public void onResponse(@NonNull Call call, @NonNull Response response) throws
IOException {
    if (response.isSuccessful()){
        ContextCompat.getMainExecutor(view.getContext()).execute(() -> {
            Toast.makeText(view.getContext(),"¡Ascensor llamado
correctamente!", Toast.LENGTH_SHORT).show();

            /*Ponemos en color verde el botón pulsado para generar un
feedback
            * hacia el usuario */
            viewHolder.getButton().setTextColor(Color.parseColor("#B2FF96"));
            new CountDownTimer(20000, 5000){
                @Override
                public void onTick(long l) {

                }

                @Override
                public void onFinish() {
                    viewHolder.getButton().setTextColor(Color.WHITE);
                }
            }.start();
        });
    }
```

Código 10: Implementación del cambio de color al pulsar botón.

3.2.4 Flujo

A continuación, se expone el diagrama de flujo correspondiente a la aplicación de usuario:

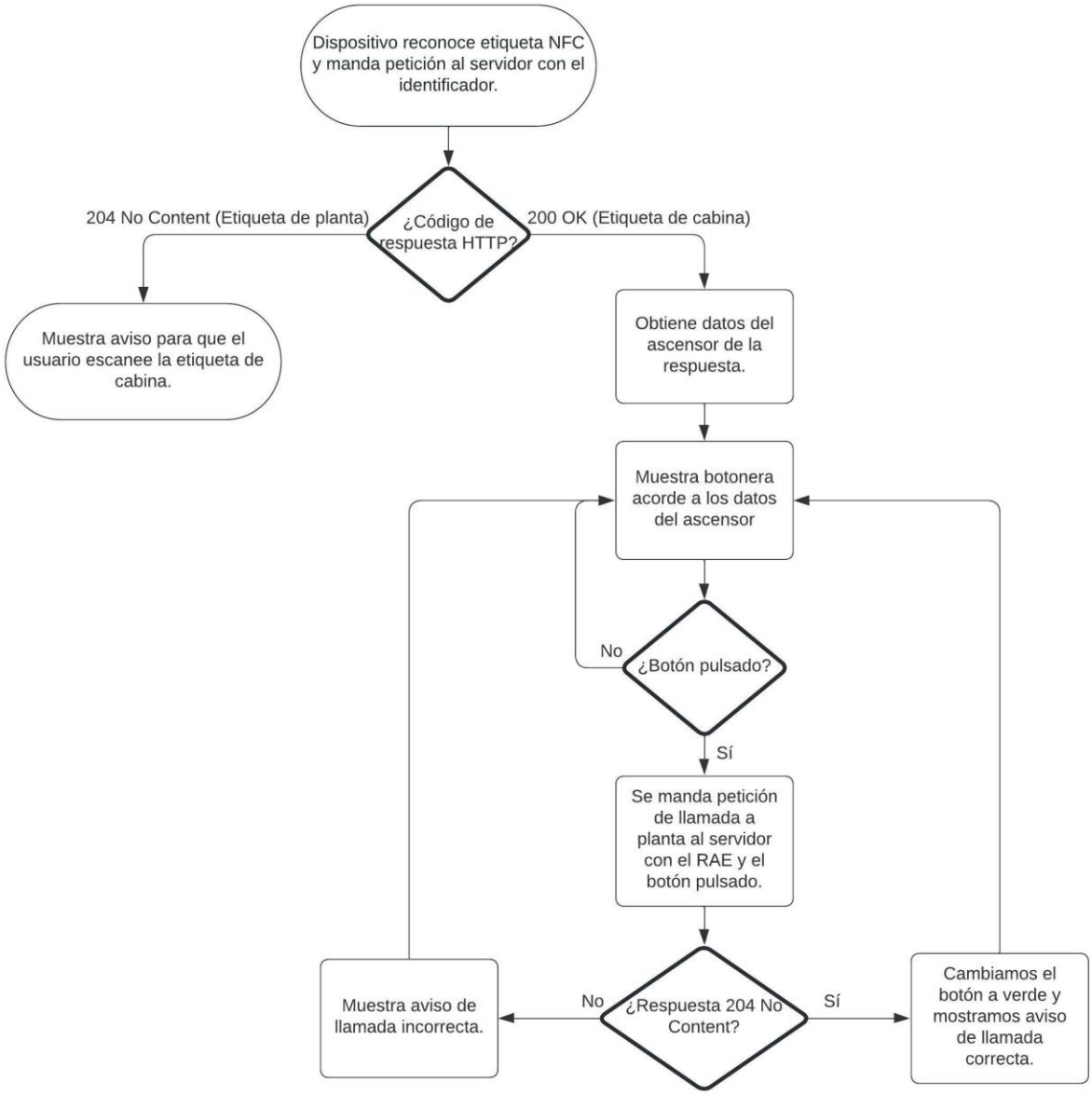


Figura 16: Diagrama de flujo de la aplicación de usuario.

3.3 Servidor

En una primera definición del alcance del proyecto, el diseño del servidor encargado de recibir las peticiones no formaba parte del desarrollo puesto que se daba por hecho que sería responsabilidad de la empresa dueña de los ascensores. Sin embargo, una vez fui avanzando en el desarrollo del proyecto caí en la cuenta de que el diseño de la API del servidor y el diseño de su base de dato eran partes esenciales para el correcto funcionamiento del resto de aplicaciones. Es por ello por lo que en este apartado se detallará el diseño de la API del servidor y su base de datos. Para que la empresa dueña de los ascensores no tenga que realizar un cambio demasiado traumático en su servidor, reutilizaremos, en la medida de lo posible, las estructuras de datos y *endpoints* ya presentes en el servidor.

La aplicación que se expondrá a continuación se trata de un servidor de pruebas en el que está implementado la API y la base de datos, pero que la llamada a los ascensores deberá implementarla la empresa dueña de los ascensores en una hipotética puesta en producción.

En esta implementación damos por hecho de que los ascensores están ya registrados en el servidor y poseen todos los datos de éstos. De la misma forma, también daremos por hecho que los identificadores de los operarios autorizados para realizar el despliegue de la solución están presentes en el servidor.

- **Nombre:** NFClift Server.
- **Lenguaje:** Go.
- **Sistema operativo:** Linux (encapsulado en Docker).
- **IDE utilizado:** GoLand.
- **Descripción:** NFClift Server es la aplicación de servidor a la que irán dirigidas las peticiones HTTPS tanto de NFClift como de NFClift Operator.

3.3.1 Descripción funcional

El servidor será el encargado de manejar las llamadas HTTPS recibidas por el resto de las aplicaciones. Estas llamadas harán que el servidor interactúe de alguna manera con su base de datos para informar a las aplicaciones del éxito o el fracaso de las mismas.

Para realizar lo anterior, el servidor debe implementar una API REST que permita realizar las siguientes acciones:

- **Registrar etiquetas.** Cuando el operario instale las etiquetas NFC debe poder registrarlas en el servidor indicando dónde irá colocada y algunos datos necesarios.
- **Desvincular etiquetas.** El servidor deberá disponer de un *endpoint* en la API que permita a los operarios desvincular una etiqueta de su acción.
- **Llamadas de etiquetas.** Cuando un usuario escanee una etiqueta NFC, se mandará automáticamente una petición HTTPS con el identificador de dicha etiqueta al servidor. El servidor debe saber dónde va colocada esa etiqueta y actuar en consecuencia.
- **Llamadas a planta.** Un usuario con la botonera del ascensor en su móvil mandará peticiones HTTPS al pulsar los botones y el servidor debe realizar la llamada al ascensor para que vaya a la planta pulsada.

Además de la API, el servidor también deberá tener una base de datos para guardar la siguiente información:

- Ascensores con sus datos: número RAE, número de paradas, textos de las paradas, paradas bloqueadas, etc.
- Identificadores de etiquetas vinculadas a la acción que debe realizar.

- Identificadores de operarios autorizados para registrar etiquetas.

3.3.2 Tecnologías usadas

Para el desarrollo del servidor he utilizado como lenguaje de programación Go. Go es un lenguaje compilado y concurrente. Tiene una sintaxis muy parecida a C y está desarrollado por Google. Le elección de este lenguaje para el desarrollo del servidor ha sido mayormente por razones personales puesto que actualmente mi carrera profesional está enfocada en el desarrollo con este lenguaje.

De todos modos, Go es un lenguaje potente, rápido y diseñado expresamente para ser utilizado en servidores y desarrollo *backend*. Una de sus características más destacable es la facilidad que ofrece para mantener procesos en paralelo de manera muy eficiente. Además, un mismo código se puede compilar para distintas arquitecturas y sistemas operativos sin tener que realizar ningún cambio, lo que lo hace muy versátil a la hora del despliegue. Todo esto hace que su uso en el desarrollo del servidor de nuestro proyecto esté más que justificado.

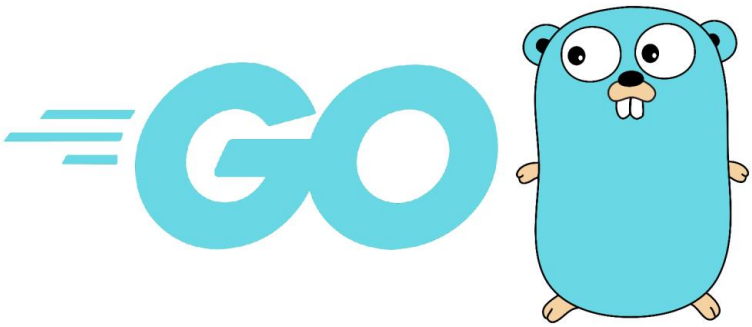


Figura 17: Go y su mascota oficial.

3.3.3 Diseño

API REST

Como se ha expuesto en apartados anteriores, las aplicaciones de operario y usuario se comunicarán con el servidor a través de su API REST. Esta API ofrece una serie de *endpoints* que las aplicaciones usarán para realizar acciones concretas en el servidor. A la hora de diseñarla tomé los siguientes requisitos que debería cumplir dicha API:

1. Un método que permita a los operarios registrar una etiqueta que vaya ubicada en planta. Para ello deberá enviar el identificador de la etiqueta, el RAE del ascensor y la planta en la que irá colocada.
2. Otro método que permita a los operarios registrar etiquetas que vayan a ir ubicadas en cabina. Deberá enviar al servidor el identificador de la etiqueta y el RAE del ascensor al que va a ir vinculada.
3. Un método que será el que usará la aplicación de usuario al escanear una etiqueta. La aplicación enviará el identificador de la etiqueta y el servidor actuará según si la etiqueta es de cabina o de planta. Si es una etiqueta de planta, el servidor llamará al ascensor a la planta vinculada a esa etiqueta y no le devolverá nada (204 No Content) al usuario. Por el contrario, si la etiqueta es de cabina, el servidor le devolverá al usuario los datos del ascensor para que pueda montar la botonera.
4. Un método para realizar las llamadas a planta.

Teniendo en cuenta los anteriores requisitos e intentando respetar lo máximo posible lo implementado en la situación actual, se ha diseñado la siguiente API:

- **[PUT] /sigma/rs/remotecall/registrartag/planta/{rae}/{planta}/{etiqueta}/{idOp}.** Endpoint que realizará el registro de las etiquetas que vayan a ser ubicadas en las plantas. Donde:
    - *{rae}* es el número RAE del ascensor asociado a la etiqueta.
    - *{planta}* es el número de la planta en la que irá colocada la etiqueta.
    - *{etiqueta}* es el identificador de la etiqueta que se va a registrar.
    - *{idOp}* es el identificador de operario que va a realizar el registro de la etiqueta.

  Cuando el servidor reciba una petición HTTPS a este *endpoint* se insertará en la base de datos el número RAE, el número de la planta y el identificador de la etiqueta; datos presentes en *path* de la petición. Para poder registrar una etiqueta, debe existir un ascensor asociado a dicho número RAE en la base de datos del servidor. El proceso anterior sólo se realizará si el identificador de operario está presente en la base de datos del servidor. La respuesta del servidor será un 204 *No Content* en caso de éxito, 500 *Internal Server Error* en caso de error en la escritura de la base de datos o planta no válida, 400 *Bad Request* en caso de parámetro erróneo o 405 *Method Not Allowed* en caso de identificador de operario no válido.

```
func handlerRegEtiquetaPlanta(w http.ResponseWriter, r *http.Request) {
   variables := mux.Vars(r)
   //Obtenemos rae, planta, id de etiqeuta e id de operador del path
   rae := variables["rae"]
   planta := variables["planta"]
   etiqueta := variables["etiqueta"]
   idOp := variables["idOp"]
   {…}
   //Comprobamos que el ID del operario está en nuestra base de datos
   if !existeOperario(idOpInt) {
      w.WriteHeader(405)
      return
   }
   //Comprobamos que el ascensor al qeu se refiere la petición tiene esa
planta
   isOk := isPlantaOk(raeInt, planta)

   if !isOk {
      w.WriteHeader(500)
      return
   }

   //Guardamos etiqueta en la base de datos
   _, err = bdGlobal.Exec("INSERT INTO plantas(etiqueta, rae, planta) VALUES
(?, ?, ?)", etiquetaInt, raeInt, planta)
   if err != nil {
      w.WriteHeader(500)
      return
   } else {
      w.WriteHeader(204)
      fmt.Println("Registrada etiqueta en planta: ", etiquetaInt, raeInt,
planta)
      return
   }
}
```

Código 11: Función encargada del registro de etiquetas de planta.

- **[PUT] /sigma/rs/remotecall/registrartag/cabina/{rae}/{etiqueta}/{idOp}.** Endpoint que realizará el registro de las etiquetas que vayan a ser ubicadas en las cabinas de los ascensores. Donde:
  - *{rae}* es el número RAE del ascensor asociado a la etiqueta.
  - *{etiqueta}* es el identificador de la etiqueta que se va a registrar.
  - *{idOp}* es el identificador de operario que va a realizar el registro de la etiqueta.

  Cuando el servidor reciba una petición HTTPS a este *endpoint* se insertará en la base de datos el número RAE y el identificador de la etiqueta; datos presentes en *path* de la petición. Para poder registrar una etiqueta, debe existir un ascensor asociado a dicho número RAE en la base de datos del servidor. El proceso anterior sólo se realizará si el identificador de operario está presente en la base de datos del servidor. La respuesta del servidor será un 204 *No Content* en caso de éxito, 500 *Internal Server Error* en caso de error en la escritura de la base de datos o planta no válida, 400 *Bad Request* en caso de parámetro erróneo o 405 *Method Not Allowed* en caso de identificador de operario no válido.

```
func handlerRegEtiquetaCabina(w http.ResponseWriter, r *http.Request) {
   variables := mux.Vars(r)
   //Obtenemos rae, id de etiqueta e id de operador del path
   rae := variables["rae"]
   etiqueta := variables["etiqueta"]
   idOp := variables["idOp"]
   {…}
   //Comprobamos que el ID del operario está en nuestra base de datos
   if !existeOperario(idOpInt) {
      w.WriteHeader(405)
      return
   }
   //Comprobamos que el ascensor al que se refiere la petición existe
   isOk := isPlantaOk(raeInt, "")
   if !isOk {
      w.WriteHeader(500)
      return
   }
   //Guardamos etiqueta en la base de datos
   _, err = bdGlobal.Exec("INSERT INTO cabinas(etiqueta, rae) VALUES (?, ?)",
etiquetaInt, raeInt)
   if err != nil {
      w.WriteHeader(500)
      return
   } else {
      w.WriteHeader(204)
      fmt.Println("Registrada etiqueta en cabina: ", etiquetaInt, raeInt)
      return
   }
}
```

Código 12: Función encargada del registro de etiquetas de cabina.

- **[GET] /sigma/rs/remotecall/tag/{etiqueta}.** Endpoint que recibirá las peticiones HTTPS de los usuarios cuando escaneen las etiquetas NFC, donde *{etiqueta}* es el identificador de dicha etiqueta.

  Cuando el servidor reciba una petición HTTPS a este *endpoint* buscará en su base de datos si el identificador de la etiqueta pertenece a una etiqueta colocada en planta, o si por el contrario pertenece a una colocada en cabina. En el caso de que la etiqueta sea de planta, el servidor devolverá una respuesta HTTP 204 *No Content* y realizará una llamada al ascensor y planta vinculados a esa etiqueta. Por otro lado, si la etiqueta está colocada en la cabina, el servidor devolverá una respuesta HTTP 200 *OK* y montará un objeto *json* con los datos del ascensor vinculado a dicha etiqueta para enviarlo en la respuesta. Este *json* será el mismo que el ya presente en la solución actual a excepción de que presentará el número RAE en lugar del ICC ID. Además, en el caso de que sea una etiqueta ubicada en cabina, el servidor devolverá un *token* de autorización con validez de 15 minutos que el usuario deberá enviar en sus llamadas con la botonera.

```
if isCabina {
   row := bdGlobal.QueryRow("SELECT * FROM ascensores WHERE rae = ?",
cabina.Rae)
   if err := row.Scan(&asc.Rae, &asc.Stops, &asc.Description, &asc.Address,
&asc.Company, &asc.AppDescription, &asc.StopTexts, &asc.StopMask,
&asc.Distance); err != nil {
      w.WriteHeader(500)
      return
   } else {
      //Procesamos las cadenas de texto que indican las plantas
      plantas := strings.Split(asc.StopTexts, ",")
      aux := ""
      for _, text := range plantas {
         text = strings.Trim(text, " ")
         if aux != "" {
            aux = aux + "," + text
         } else {
            aux = text
         }
      }
      if aux != "" {
         asc.StopTexts = aux
      }

      mask := strings.Split(asc.StopMask, ",")
      aux = ""
      for _, text := range mask {
         text = strings.Trim(text, " ")
         if aux != "" {
            aux = aux + "," + text
         } else {
            aux = text
         }
      }
      if aux != "" {
         asc.StopMask = aux
      }

      //Construcción del token para la autorizacion
      {…}
      //w.WriteHeader(200)
      http.SetCookie(w, &http.Cookie{
         Name:    "token",
         Value:   tokenString,
         Expires: expirationTime,
      })
      json.NewEncoder(w).Encode(asc)

      fmt.Println("Llamada desde cabina")
      return
   }
}
```

Código 13: Comportamiento del servidor al recibir etiqueta de cabina.

- **[PUT] /sigma/rs/remotecall/{rae}/stops/{planta}/call.** Endpoint que recibirá las peticiones HTTPS de los usuarios cuando pulsen un botón en la botonera de la aplicación. Donde:
  - *{rae}* es el número RAE del ascensor al que se le va a realizar la llamada.
  - *{planta}* es el número de la planta pulsado por el usuario.

  Cuando el servidor reciba una petición HTTPS a este *endpoint* el servidor comprobará que dicho

ascensor ha sido registrado previamente y que el número de la planta es válido. En caso de que lo anterior sea correcto, el servidor realizará una llamada al ascensor a dicha planta. Este *endpoint* ha sido reutilizado casi por completo de la solución actual a excepción del cambio del ICC ID por el número RAE. Al igual que en la solución actual, el usuario deberá enviar el *token* de autorización que ha recibido a la hora de escanear la etiqueta NFC de la cabina. En caso de llamada exitosa el servidor devolverá un 204 *No Content*, 401 *Unauthorized* en caso de *token* incorrecto, 400 *Bad Request* en caso de RAE inválido o 500 *Internal Server Error* en caso de planta incorrecta.

```
func handlerLlamada(w http.ResponseWriter, r *http.Request) {

    // Obtenemos el token de la cabecera de la llamada
    c := r.Header.Get("token")
    if c == "" {
        // If the cookie is not set, return an unauthorized status
        w.WriteHeader(http.StatusUnauthorized)
        return
    }

    //Validamos el token
    {...}
    variables := mux.Vars(r)
    //Obtenemos rae y planta del path
    rae := variables["rae"]
    planta := variables["planta"]

    raeInt, err := strconv.Atoi(rae)
    if err != nil {
        w.WriteHeader(400)
        return
    }

    //Comprobamos que ascensor esté registrado y planta sea válida
    isOk := isPlantaOk(raeInt, planta)
    if !isOk {
        w.WriteHeader(500)
        return
    } else {
        w.WriteHeader(204)
        ////////////////////////Aquí se haría la llamada al
ascensor//////////////////////////////////////
        fmt.Println("Llamada a ascensor ", rae, " a planta ", planta)
        return
    }
}
```

Código 14: Función encargada de recibir las peticiones de llamadas a plantas.

- **[DELETE] /sigma/rs/remotecall/desvinculartag/{etiqueta}/{idOp}.** *Endpoint* que realizará la desvinculación de una etiqueta NFC de su acción vinculada en el servidor. Donde *{etiqueta}* es el identificador de la etiqueta e *{idOp}* el identificador de operario.

  Cuando el servidor reciba una llamada HTTPS a este *endpoint*, comprobará que el identificador de operario es correcto y acto seguido borrará de su base de datos toda referencia al identificador de etiqueta recibido.

```
func handlerDesvincularEtiqueta(w http.ResponseWriter, r *http.Request) {
   variables := mux.Vars(r)
   //Obtenemos rae, id de etiqueta e id de operador del path
   etiqueta := variables["etiqueta"]
   idOp := variables["idOp"]

   {…}
   //Comprobamos que el ID del operario está en nuestra base de datos
   if ¡existeOperario(idOpInt) {
      w.WriteHeader(405)
      return
   }

   _, err = bdGlobal.Exec("DELETE FROM cabinas WHERE etiqueta = ¿",
etiquetaInt)
   if err ¡= nil {
      w.WriteHeader(500)
      return
   }
   _, err = bdGlobal.Exec("DELETE FROM plantas WHERE etiqueta = ¿",
etiquetaInt)
   if err ¡= nil {
      w.WriteHeader(500)
      return
   }

   w.WriteHeader(204)
   fmt.Println("Desvinculada etiqueta: ", etiquetaInt)
   return
}
```

Código 15: Función encargada de desvincular etiquetas.

Base de datos

El otro punto clave en el diseño de la aplicación del servidor, es el diseño de la base de datos en la que guardaremos los datos de los ascensores, las etiquetas registradas y los operarios autorizados. Como lenguaje, utilizaremos SQLite por simple gusto personal y familiaridad. En la siguiente figura se muestra el diagrama de entidad-relación correspondiente a la base de datos del servidor:

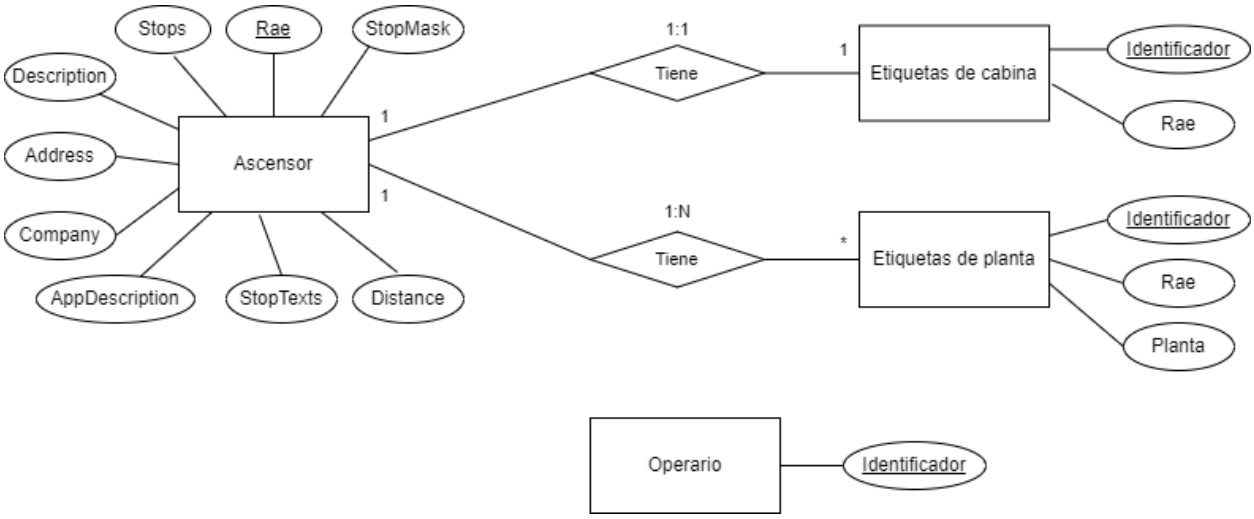


Figura 18: Diagrama E-R.

A continuación, se detallarán las tablas que he considerado necesarias para el funcionamiento del proyecto:

- **Ascensores**. Esta tabla guardará los datos de los ascensores. He considerado que los datos almacenados en esta tabla serán los mismos que los que se envíen en el *endpoint* /sigma/rs/remotecall/tag/{etiqueta} cuando se escanea una etiqueta de cabina. La tabla consta de 9 registros:
    - **rae**. Clave primaria de tipo *number.* Indica el número RAE del ascensor, el cual es un identificador único.
    - ***stops.*** De tipo *number.* Indica el número de paradas que tiene el ascensor.
    - ***description.*** De tipo *text*. Contiene una breve descripción del ascensor.
    - ***address.*** De tipo *text.* Indica la calle y el número donde está ubicado el ascensor.
    - ***company.*** De tipo *text.* Indica el nombre de la empresa propietaria del ascensor.
    - ***appDescription.*** De tipo *text*. Contiene una descripción de la *app*.
    - ***stopTexts.*** De tipo *text*. Contiene los textos de los botones de las paradas del ascensor separados por comas.
    - ***stopMask.*** De tipo *text*. Contiene los textos separados por comas de los botones que no estarán disponibles para ser llamados por la aplicación.
    - ***distance.*** De tipo *text*. Indica la distancia a la que está el ascensor. Registro que viene heredado de la situación actual y que no nos aporta ningún valor.
- **Plantas**. Esta tabla guardará las etiquetas que van colocadas en planta y que han registrado los operarios. La tabla consta de 3 registros: etiqueta, rae y planta; los dos primeros de tipo *number* y el último de tipo *text*. El registro etiqueta almacenará el identificador de la etiqueta y será clave primaria puesto que no debe haber etiquetas con identificadores repetidos. El registro rae guarda el número RAE del ascensor al que está vinculada la etiqueta y el registro planta guarda el texto de la planta en la que está instalada la etiqueta.
- **Cabinas**. Esta tabla guardará las etiquetas que van colocadas en cabina y que han sido registradas por los operarios. La tabla consta de 2 registros: etiqueta y rae; ambos de tipo *number*. Al igual que en la tabla anterior, los registros guardarán el identificador de etiqueta y el número RAE del ascensor asociado respectivamente.
- **Operarios**. Esta tabla guardará los identificadores de los operarios autorizados para la realización de registros de ascensores y etiquetas. La tabla consta de un solo registro de tipo *number*, id, en el que se guardará el identificador de los operarios.

```
func initRepo() {

   bd, err := sql.Open("sqlite3", "nfclift.sqlite")
   if err != nil {
      fmt.Println(err)
   }
   _, err = bd.Exec("create table if not exists ascensores(rae number primary
key, stops number, description text, address string, company text,
appDescription text, stopTexts text, stopMask text, distance text)")
   if err != nil {
      panic(err)
   }
   _, err = bd.Exec("create table if not exists plantas(etiqueta number
primary key, rae number, planta number)")
   if err != nil {
      panic(err)
   }
   _, err = bd.Exec("create table if not exists cabinas(etiqueta number
primary key, rae number)")
   if err != nil {
      panic(err)
   }
   _, err = bd.Exec("create table if not exists operarios(id number primary
key)")
   if err != nil {
      panic(err)
   }

   bdGlobal = bd

}
```

Código 16: Función que inicializa la base de datos.

3.4 Entornos de desarrollo y sus réplicas

3.4.1 Aplicaciones Android

Para el proceso de desarrollo de las aplicaciones Android he usado tres herramientas para conformar el entorno. La primera ha sido Android Studio, el IDE con el que he realizado la propia escritura del código de dichas aplicaciones. La segunda herramienta que he usado ha sido Git, un software de control de versiones que me ha ayudado a conseguir un desarrollo más seguro a la hora de modificar funcionalidades de las propias aplicaciones y no perder trabajo ya realizado debido a errores. Por último, he hecho uso de la página web GitHub, la cual me ha permitido alojar el proyecto en sus repositorios y realizar fácilmente archivos comprimidos con los códigos fuente.

En el caso de que el lector quiera replicar el entorno de desarrollo que yo mismo he usado y ser capaz de probar el código, mantenerlo o incluso mejorarlo, podrá hacerlo siguiendo las siguientes instrucciones:

1. En primer lugar, deberá descargarse los dos softwares arriba mencionados (Android Studio y Git) de sus correspondientes páginas oficiales (Ver apartado Enlaces de interés) e instalarlos.
2. A continuación, deberá descargarse el código fuente de las aplicaciones de los siguientes enlaces:

   Aplicación de operario:
   https://drive.google.com/file/d/1ABYGzeS8qrdFHVU4krvZMFp3DHXw_PTl/view?usp=sharing

   Aplicación de usuario:
   https://drive.google.com/file/d/1kguciV3n3tOpfSq1h4FiffY2S2TQcH4h/view?usp=sharing
3. Una vez con el código fuente en formato *zip,* deberá descomprimir cada archivo en una nueva carpeta y al abrir dicha carpeta con Android Studio ya tendremos el código disponible para ser inspeccionado

o incluso editado.

4. Para realizar un control de versiones sobre el código tan sólo se debe abrir una consola, situarse en la carpeta en la que se ha descomprimido el código y ejecutar el comando *git init*.

En este punto el lector habrá replicado el entorno de desarrollo con el que he trabajado a la hora de desarrollar las aplicaciones Android de esta solución.

3.4.2 Servidor

Al igual que para las aplicaciones Android, he usado Git para el control de versiones en el proceso de desarrollo y GitHub para el alojamiento del código. Por el contrario, para la escritura del código he usado GoLand como IDE puesto que es uno diseñado expresamente para codificar en Go.

A la hora de replicar el entorno de desarrollo, los pasos a seguir son exactamente los mismos que los descritos en el apartado anterior pero esta vez descargando e instalando GoLand en lugar de Android Studio. Aunque GoLand no es un IDE gratuito, tiene una versión de pruebas gratuita de 30 días.

Enlace de descarga del código fuente del servidor:
https://drive.google.com/file/d/1O5iFan85MBUzqBMclkbPmnhDe82niDRn/view?usp=sharing

4 PRUEBAS

En este capítulo detallaré todo el proceso de pruebas que he seguido para la comprobación de la consecución de los objetivos del proyecto. Estas pruebas también servirán para cerciorarnos del buen funcionamiento de las aplicaciones que hemos desarrollado y corregir algún error que pueda haber.

4.1 Materiales utilizados

Los materiales que he utilizado para la realización de las pruebas han sido:

- **Motorola Moto g6 plus.** Éste es el dispositivo con el que he realizado las pruebas en casi todo momento. Es un sistema Android con la versión 9.0 y con lector/escritor NFC. En este dispositivo se ha instalado tanto NFClift (aplicación de usuario) como NFClift Operator (aplicación de operario).
- **OnePlus Nord.** Dispositivo Android con versión 11.0 y también con lector/escritor NFC. Ha sido utilizado como apoyo al dispositivo anterior y también para probar las aplicaciones en versiones de Android distintas.
- **Etiquetas NTAG215 reescribibles.** Aunque con unas etiquetas NFC de categoría NTAG210 habría sido suficiente, por temas de comodidad y disponibilidad, decidí usar de categoría NTAG215. Por otro lado, en el momento del despliegue en campo y la puesta en producción la solución funcionará con etiquetas escritas de fábrica y *read only,* pero en el momento de las pruebas necesitamos que sean reescribibles para realizar múltiples intentos.
- **Ordenador con Windows 10, 16 GB de RAM y procesador x86 de 64 bits con 4 núcleos a 3'6 GHz.** Éste es el dispositivo que he usado como servidor durante las pruebas. Su velocidad y potencia supera ampliamente las mínimas necesarias para que se pueda ejecutar la aplicación de servidor, que, aunque no se ha realizado un estudio exhaustivo de las necesidades mínimas, podrá ejecutarse en ordenadores menos potentes.

4.2 Herramientas utilizadas

Para la realización de las pruebas he necesitado utilizar herramientas software de terceros: *NFC Tools, Docker, Postman* y *DB Browser for SQlite*. A continuación, detallaré la utilidad de todas esas herramientas y para qué las he necesitado en la realización de las pruebas de nuestras aplicaciones.

NFC Tools

NFC Tools es una aplicación para Android que permite la lectura y escritura de etiquetas NFC con nuestro dispositivo móvil. Además de realizar la lectura del contenido de los datos de las etiquetas, también nos permite ver el contenido de las cabeceras de éstas. Un ejemplo de lectura de etiqueta con esta aplicación lo podemos ver en la figura 11.

Figura 19: Icono de NFC Tools.

Docker

Docker es un proyecto de código libre que nos permite la automatización del despliegue de nuestras aplicaciones en contenedores aislados. Para las pruebas que realizaremos, nos apoyaremos en *Docker* para el despliegue de nuestro servidor en un contenedor que actúe como servicio, posibilitándonos el levantarlo o pararlo de manera sencilla.

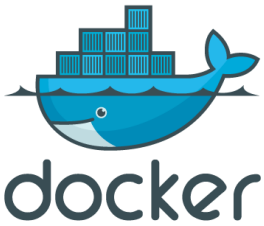


Figura 20: Logo de Docker.

Para poder desplegar nuestro servidor con Docker primero debemos crear un *Dockerfile* para su compilación y creación de imagen de Docker. Un *Dockerfile* es un archivo de texto en el que se incluyen distintos comandos que se ejecutarán en orden para la creación de la imagen Docker. En la figura siguiente se muestra el *Dockerfile* que he creado para la creación de una imagen Docker con el binario estático de nuestra aplicación de servidor.

```
1  FROM golang:1.14
2
3  COPY ./main.go .
4  COPY ./service.go .
5  COPY ./repo.go .
6  COPY ./nfclift.sqlite .
7  COPY ./key.pem .
8
9  RUN go get -t -d -v .
10
11 RUN go build -tags osusergo,netgo -o serverNFClift -a .
12
13 EXPOSE 80/tcp
14
15 ENTRYPOINT ./serverNFClift
```

Figura 21: Dockerfile usado para la imagen del servidor.

Los comandos que contiene el *Dockerfile* anterior copiarán el código del servidor, base de datos y certificado SSL; descargarán sus dependencias, compilarán el código, habilitará el puerto 80 (es el que estará escuchando nuestro servidor) y expondrá el binario al exterior. Como nuestro servidor no tiene implementada ninguna manera de añadir nuevos identificadores de operarios autorizados ni ascensores, debemos encapsular la base de datos en el contenedor Docker con los identificadores de operarios y ascensores ya insertados. Cuando ejecutemos el comando `docker build` se ejecutarán los comandos del *Dockerfile* y se creará la imagen de Docker que contendrá nuestro servidor. Aunque se podría desplegar el servidor simplemente ejecutando `docker run <nombre de nuestra imagen>`, en mi caso he preferido desplegarlo en *Docker Desktop*.

Docker Desktop es una aplicación de escritorio para Windows y Mac que permite desplegar los contenedores de Docker de una manera sencilla. Nos ayudará a levantar y parar nuestro servidor de manera sencilla y sin necesidades de ejecutar comandos de consola, todo por interfaz gráfica.

Postman

Postman es un programa que permite realizar llamadas HTTP y HTTPS para probar APIs fácilmente gracias a su interfaz gráfica. Para las pruebas, usaremos *Postman* para que nos facilite el comprobar el buen funcionamiento de los *endpoints* de NFClift Server.

Figura 22: Logo de Postman.

DB Browser for SQlite

DB Browser for SQlite es un programa que nos permite inspeccionar y modificar los datos de bases de datos *SQlite* de forma intuitiva a través de su interfaz gráfica. Lo usaremos en las pruebas sobre NFClift Server y comprobar que las llamadas a los *endpoints* están efectivamente modificando la base de datos de la forma que pretendemos.

4.3 Escenario de pruebas

El escenario en el que se han realizado las pruebas consta de los siguientes elementos:

- Dispositivo Android con lector NFC con NFClift y NFClift Operator instalados. La conexión a internet del dispositivo es a través de la red móvil para así estar en una subred distinta al servidor.
- Etiquetas NFC NTAG215 reescribibles.
- Ordenador con la imagen de docker en la que se ha encapsulado el servidor corriendo en Docker Desktop, escuchando en el puerto 80 y con conexión a internet. Android Studio está instalado para poder depurar las aplicaciones Android.

Figura 23: Docker Desktop con el contenedor del servidor corriendo.

Para que las aplicaciones NFClift y NFClift Operator puedan comunicarse con nuestro servidor de pruebas a través de peticiones HTTPS las aplicaciones deben confiar en el certificado usado por el servidor. Como esto es un entorno de pruebas, utilizaremos un certificado auto firmado. Esto significa que generaremos el certificado directamente en el ordenador que actuará de servidor y lo validaremos nosotros mismos. Esto hará que nuestras aplicaciones no confíen de dicho certificado puesto que al ser auto firmado no pueden validarlo con una entidad validadora externa. Para que nuestras aplicaciones puedan confiar en dicho certificado deberemos encapsularlo en nuestro propio código (carpeta res/raw) y añadir la siguiente línea al apartado *application* del Android Manifest:

```
android:networkSecurityConfig="@xml/network_security_config"
```

Código 17: Línea a añadir al Android Manifest.

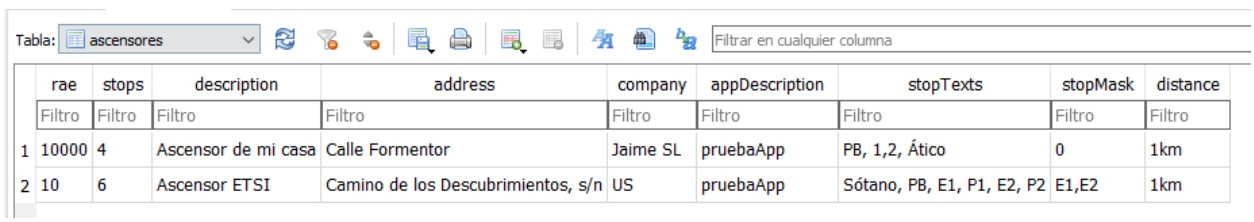
Por último, deberemos crear un nuevo archivo en la carpeta res/xml llamado *network_security_config.xml* con el siguiente contenido:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <base-config>
        <trust-anchors>
            <certificates src="@raw/cert"/>
        </trust-anchors>
    </base-config>
</network-security-config>
```

Código 18: Archivo network_security_config.xml

En este punto nuestras aplicaciones ya confían en el certificado auto firmado de nuestro servidor. Como es de esperar, esto solo lo haremos para funcionar en un entorno de pruebas. Cuando la solución fuera a desplegarse en un escenario real necesitaremos tener un certificado validado por entidades externas.

Como hemos comentado anteriormente, los datos de los ascensores y los identificadores de los operarios autorizados deben estar presentes en la base de datos desde un primer momento. Para las pruebas, he creado la imagen de Docker del servidor con la base de datos encapsulada en el siguiente estado:



	rae	stops	description	address	company	appDescription	stopTexts	stopMask	distance
	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro
1	10000	4	Ascensor de mi casa	Calle Formentor	Jaime SL	pruebaApp	PB, 1,2, Ático	0	1km
2	10	6	Ascensor ETSI	Camino de los Descubrimientos, s/n	US	pruebaApp	Sótano, PB, E1, P1, E2, P2	E1,E2	1km

Figura 24: Datos en la tabla ascensores.

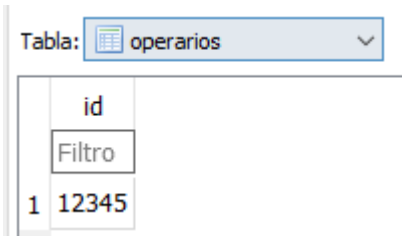


Figura 25: Datos en la tabla operarios.

Las etiquetas NFC que usaremos durante las pruebas han sido escritas previamente con *NFC Tools* con números aleatorios y con tipo *nfc/lift*.

En la siguiente figura se muestra el diagrama de red correspondiente al escenario con el que vamos a realizar las pruebas.

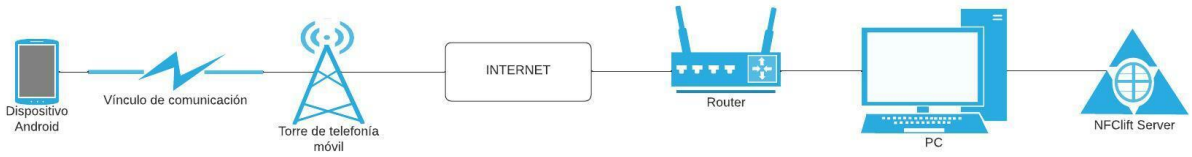


Figura 26: Diagrama de red del escenario de pruebas.

4.4 Plan de pruebas

Las primeras pruebas que le realizaremos a las aplicaciones serán pruebas que cubrirán funcionalidades concretas de cada una de ellas. Una vez hayamos comprobado que todas ellas funcionan correctamente de forma aislada, realizaremos una prueba final que consistirá en simular una posible situación real en un despliegue de nuestra solución en un ascensor.

4.4.1 Pruebas NFClift Operator

Las funcionalidades de NFClift Operator que cubriremos con las pruebas de este apartado serán las siguientes:

- Lectura de etiquetas NFC.
- Registro de etiquetas en el servidor.

Para la realización de las pruebas deberemos tener conectado nuestro dispositivo Android al ordenador con la depuración USB habilitada, lanzar la aplicación desde Android Studio y señalar como dispositivo el nuestro físico.

Lectura y registro de etiquetas NFC

Puesto que la lectura y el registro de las etiquetas NFC se realizan en el mismo instante, las pruebas de estas dos funcionalidades las realizaremos a la vez.

Empezaremos con las pruebas para etiquetas que vayan colocadas en planta. La metodología a seguir será colocar *logs* en el código y cerciorarnos de que actúa como hemos diseñado.

La primera prueba consistirá en una lectura y registro correctos de la etiqueta NFC que va a ir colocada en planta. En la siguiente figura podemos ver los datos con los que se intentará registrar la etiqueta:

Figura 27: Prueba de éxito de lectura y registro de etiqueta en planta.

Para que el registro sea satisfactorio, deberemos haber acercado el dispositivo móvil a la etiqueta NFC antes de pulsar el botón. En la figura siguiente podemos ver los *logs* correspondientes a la prueba:

```
D/I: ID de la etiqueta: 4249
D/I: Request: Request{method=PUT, url=https://85.136.224.37:80/sigma/rs/remotecall/registrartag/planta/10000/%C3%81tico/4249/12345}
```

Figura 28: Logs de prueba de éxito de lectura y registro de etiqueta en planta.

Los *logs* que podemos observar en la figura anterior han sido colocados en el momento de la lectura correcta de la etiqueta y en la construcción de la petición HTTPS de registro. Analizando dichos *logs* podemos comprobar que la petición HTTPS se ajusta a los datos introducidos en la Figura 27 y a lo esperado en la API del servidor (apartado 3.3.3).

Por último, realizaremos una prueba para comprobar que no se realiza ningún registro en el caso de que se introduzcan datos inverosímiles. En las figuras que se muestran a continuación vemos los datos con los que se intentará registrar la etiqueta y los *logs* resultantes.

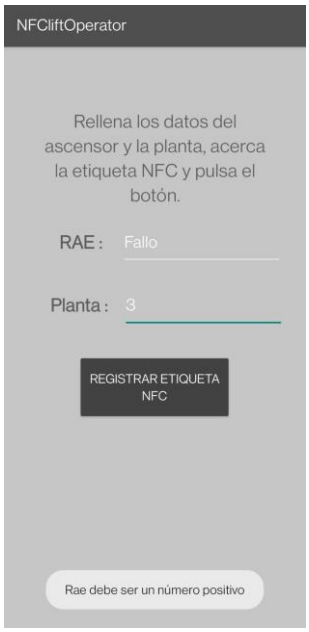


Figura 29: Prueba de error de registro de etiqueta de planta.

Podemos observar que, como era de esperar, el número RAE introducido no es válido y, por lo tanto, el registro no se ha realizado. El *toast* nos indica el error.

Puesto que la lectura y registro de las etiquetas NFC colocadas en cabina siguen un proceso idéntico al de las colocadas en planta, no realizaremos pruebas particulares para dicho caso, sino que lo comprobaremos en las pruebas de la solución completa.

Desvinculación de etiquetas

Figura 30: Interfaz de desvinculación de etiqueta.

En este apartado realizaremos la prueba de desvinculación de una etiqueta NFC de su acción. Al igual que en el apartado anterior, para cerciorarnos de que la petición de desvinculación llega bien al servidor, colocaremos *logs* en el código.

Cuando pulsemos el botón que aparece en la pantalla se enviará una petición al servidor que borrará la etiqueta de su base de datos. Para la prueba intentaremos desvincular una de las etiquetas que tenemos y veremos si la petición que se envía al servidor es correcta.

```
Contenido de la etiqueta a desvincular: 9351
D/I: https://85.136.224.37:80/sigma/rs/remotecall/desvinculartag/9351/12345
```

Figura 31: Logs generados al pulsar el botón desvincular.

Como podemos ver en la figura anterior, la petición realizada al servidor es correcta y la etiqueta que vamos a desvincular también.

4.4.2 Pruebas NFClift

Las funcionalidades de NFClift que cubriremos con las pruebas de este apartado son:

- Lectura de etiqueta NFC.
- Lanzamiento automático de la aplicación al escanear una etiqueta NFC de tipo *nfc/lift*.

Puesto que en este punto nuestro servidor aún no ha sido probado, no podemos comprobar la funcionalidad referente al mostrado de la botonera del ascensor. Para la prueba de esta funcionalidad necesitamos que el servidor conteste a nuestras peticiones HTTPS y por lo tanto la realizaremos en las pruebas de la solución completa.

Lectura de etiqueta NFC y lanzamiento automático de la aplicación

Como es lógico, para que se den las condiciones de probar que la aplicación se lance automáticamente al leer una etiqueta de tipo *nfc/lift*, primero la aplicación debe leer bien las etiquetas NFC. Por lo tanto, probaremos ambas funcionalidades a la vez.

Lo primero que haremos será conseguir una etiqueta NFC de tipo *nfc/lift* y con un número escrito en ella simulando un identificador de etiqueta. Como ya hemos hecho anteriormente, utilizaremos *NFC Tools* para escribir una etiqueta con el formato deseado para probar NFClift.

Con la etiqueta ya escrita, cogeremos nuestro dispositivo Android con NFClift corriendo a través de Android Studio y lo acercaremos a la etiqueta. He colocado varios logs en puntos críticos del proceso de la aplicación; en el momento en el que se ha recibido una etiqueta del tipo *nfc/lift* dentro de un *intent*, en el momento en el que se lee el contenido de la etiqueta y cuando se envía la petición HTTPS al servidor. En la siguiente figura podemos observar los *logs* correspondientes a la prueba.

```
D/I: Etiqueta NFC de tipo nfc/lift leida
D/I: Contenido de la etiqueta: 4249
D/I: Request{method=GET, url=https://85.136.224.37:80/sigma/rs/remotecall/tag/4249}
```

Figura 32: Prueba de éxito de lectura de etiqueta NFC con NFClift.

Si ahora leemos el contenido de la etiqueta con la aplicación *NFC Tools*, podemos observar que su contenido

es el mismo que el leído por NFClift. Por lo tanto, podemos concluir que NFClift lee correctamente las etiquetas.

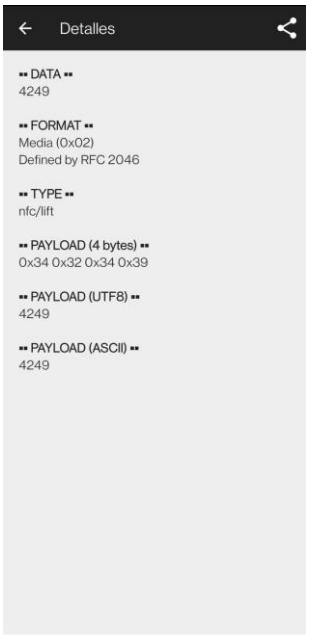


Figura 33: Contenido de la etiqueta leído por NFC Tools.

También se ha realizado la prueba de acercar una etiqueta NFC de un tipo distinto a *nfc/lift* para cerciorarnos de que nuestra aplicación no la captura. Al acercar la etiqueta no se aprecia ninguna traza de *log* de las anteriores y, por lo tanto, está funcionando correctamente.

Por otra parte, el lanzamiento automático de la aplicación al acercar una etiqueta del tipo *nfc/lift* también ha sido comprobado. Con NFClift instalada en el dispositivo Android, pero sin estar ejecutándose, hemos acercado la etiqueta a nuestro dispositivo y hemos obtenido un *toast* que nos indica el lanzado de la petición HTTPS al servidor, por lo que podemos dar por satisfactoria la prueba. Como esta prueba era dinámica por definición, resulta imposible el poder mostrarla gráficamente en un documento como éste.

4.4.3 Pruebas NFClift Server

NFClift Server no tiene ninguna funcionalidad para el usuario más allá del buen funcionamiento de los *endpoints* de la API y su relación con la base de datos. Por lo tanto, esto es precisamente lo que probaremos en este apartado gracias a las herramientas de *Postman* y *DB Browser for SQlite*.

API y su relación con la base de datos

Para estas pruebas, lanzaremos el servidor en local, fuera del contenedor de Docker para poder incluir *logs* y observarlos de manera más sencilla. Además, de esta forma tendremos la base de dato más accesible para ver sus datos. En Go es sencillo compilar el código y con un par de comandos ya obtendríamos el binario estático del servidor. A pesar de ello, dado que *GoLand* permite lanzar los programas desde el propio IDE, eso es precisamente lo que haremos para que el cambio en el código para incluir o quitar *logs* sea más sencillo y no tener que recompilar cada vez que realicemos uno.

Las pruebas a los *endpoints* que realizaremos a continuación parten de una base de datos totalmente vacía a

excepción de la tabla que contiene los identificadores de los operarios autorizados a realizar cambios, la cual contendrá el identificador "12345", y la tabla de los ascensores. En exclusiva para facilitar las pruebas, se ha añadido un *endpoint* que vacía toda la base de datos a excepción de las tablas mencionadas.

A continuación, para cada *endpoint* de la API del servidor, se realizará una prueba de éxito y otra de fallo, mostrando además cómo afecta cada una al contenido de la base de datos. Las llamadas a los *endpoints* se realizarán todas a través de *Postman* y los cambios en la base de datos se observarán con *DB Browser for SQlite*.

- **[PUT] /sigma/rs/remotecall/registrartag/planta/{rae}/{planta}/{etiqueta}/{idOp}:** En primer lugar, realizaremos una prueba de éxito de registro de etiqueta en planta. En las siguientes figuras se muestra la petición HTTPS, la respuesta y el estado en el que se queda la base de datos:

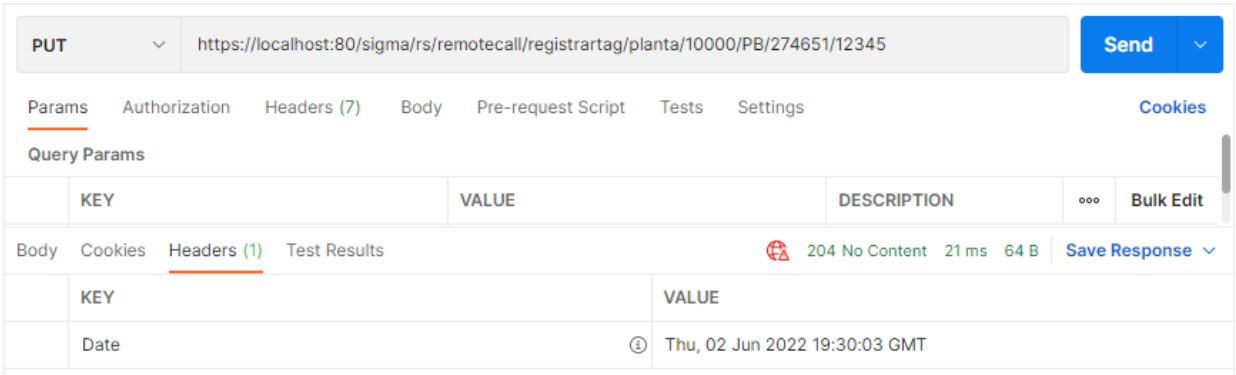


Figura 34: Petición y respuesta de caso de éxito de registro de etiqueta en planta.

Figura 35: Estado de la base de datos después de la prueba de éxito.

Se puede observar en las figuras anteriores que la respuesta del servidor ha sido un 204 *No Content* y que en la base de datos aparece registrada la etiqueta. Por lo tanto, la prueba ha tenido los resultados esperados.

Para la prueba de fallo intentaremos realizar el registro de una etiqueta vinculándola a un número RAE que no se corresponde con ninguno de los ascensores registrados en la base de datos. A continuación, los resultados:

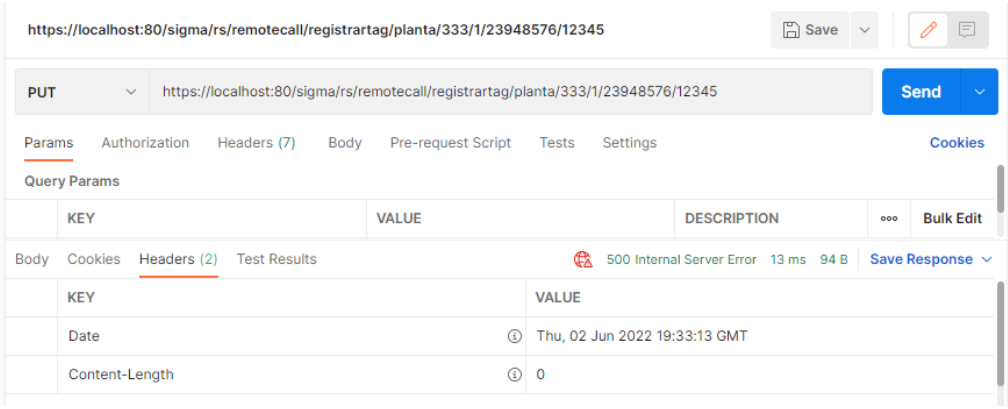


Figura 36: Petición y respuesta de caso de fallo de registro de etiqueta en planta.

Figura 37: Estado de la base de datos después de la prueba de fallo.

Podemos ver que la respuesta del servidor ha sido de código 500 *Internal Server Error* y que la nueva etiqueta no se encuentra registrada en la base de datos. La prueba ha resultado exitosa.

- **[PUT] /sigma/rs/remotecall/registrartag/cabina/{rae}/{etiqueta}/{idOp}:** Para la realización de las pruebas de este *endpoint* partiremos de la base de datos en el estado que la dejamos en la prueba anterior.

  Empezaremos realizando la prueba de éxito registrando la etiqueta de cabina con el número RAE del ascensor que está registrado en la base de datos. Las siguientes figuras muestran el resultado de la petición y el estado de la base de datos.

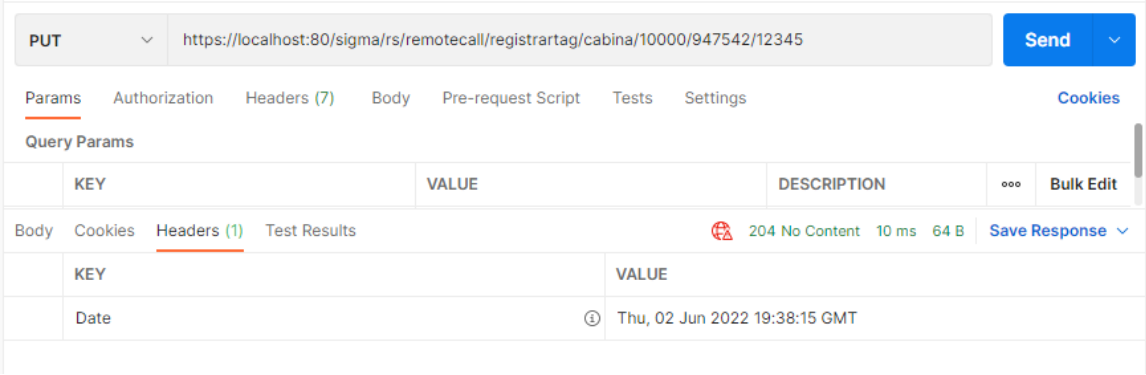


Figura 38: Petición y respuesta de registro de etiqueta de cabina para prueba de éxito.

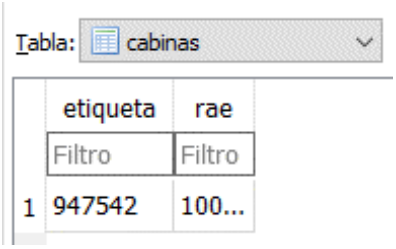


Figura 39: Estado de la base de datos después de la prueba.

Podemos observar que la respuesta a la petición es satisfactoria y que la etiqueta ha quedado registrada correctamente en la base de datos. Por lo tanto, la prueba ha tenido los resultados esperados.

La prueba de fallo, al igual que en el caso de las etiquetas de planta, se realizará intentando registrar una etiqueta con un número RAE no registrado anteriormente. En las siguientes figuras se muestra el resultado de la prueba.

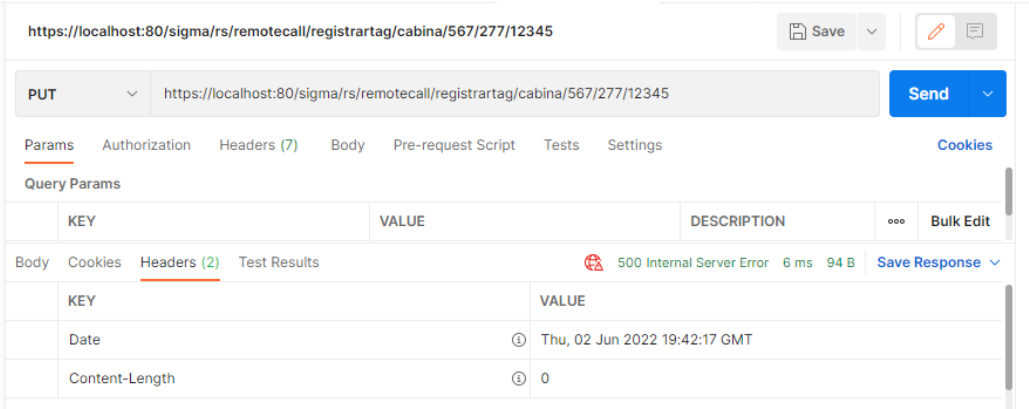


Figura 40: Petición y respuesta de registro de etiqueta de cabina para prueba de fallo.

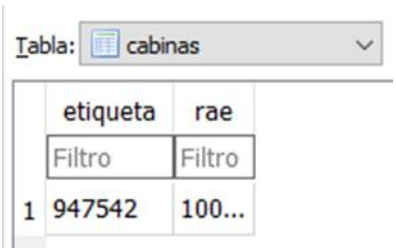


Figura 41: Estado de la base de datos después de la prueba.

Como podemos ver, el resultado ha sido el esperado; respuesta 500 *Internal Server Error* y etiqueta no registrada en la base de datos.

- **[GET] /sigma/rs/remotecall/tag/{etiqueta}.** Este *endpoint* se comportará de una manera distinta dependiendo de si el identificador de etiqueta enviado pertenece a una etiqueta de cabina o de planta. Por lo tanto, deberemos realizar una prueba de éxito para ambos casos. Para ello, usaremos los identificadores de las etiquetas registradas en las pruebas anteriores.

  La primera prueba que realizaremos será la de mandar una petición con el identificador de la etiqueta colocada en planta. En la siguiente figura podemos ver la petición y su respuesta.

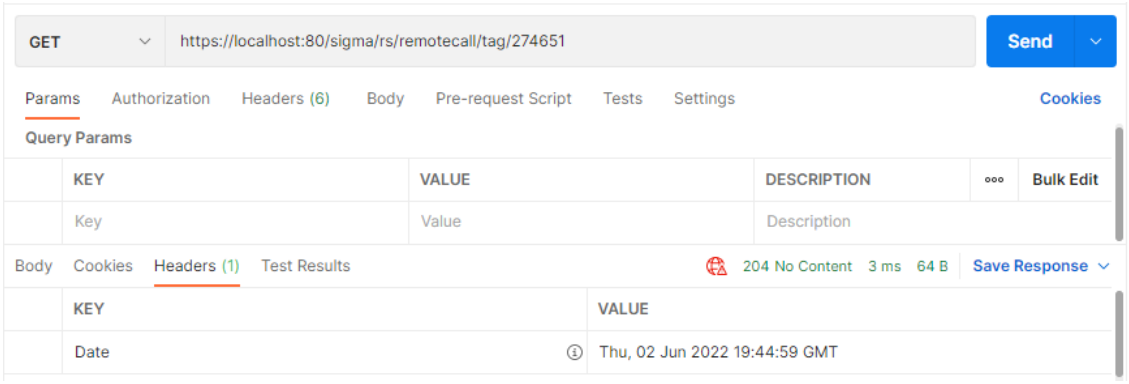


Figura 42: Petición y respuesta de llamada con etiqueta ubicada en planta para prueba de éxito.

En la imagen anterior podemos ver que la petición ha resultado exitosa puesto que la respuesta ha sido 204 *No Content*. En este caso no comprobaremos la base de datos porque este *endpoint* no debe modificarla en ningún momento. En el caso de una etiqueta de planta, el servidor realizará una llamada al ascensor de manera interna. Por lo tanto, con la respuesta positiva podemos dar por hecho el buen funcionamiento del *endpoint*.

A continuación, realizamos la prueba de mandar una petición con el identificador de una etiqueta colocada en cabina. En este caso, la respuesta del servidor deberá ser positiva, contener un objeto *json* con los datos del ascensor necesarios para poder montar la botonera de éste y un *token* de autenticación. En la siguiente figura podemos observar dicha petición y respuesta.

Figura 43: Petición y respuesta de llamada con etiqueta ubicada en cabina para prueba de éxito.

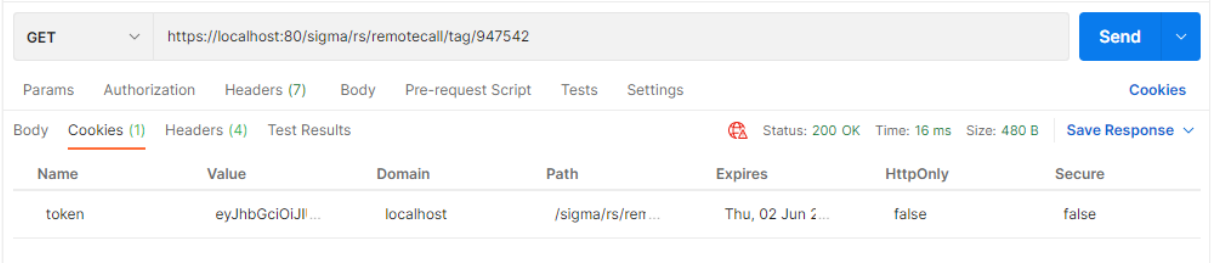


Figura 44: *Token* recibido con la respuesta.

En la imagen podemos ver que la respuesta ha sido correcta y que, efectivamente, nos ha devuelto un objeto *json* con los datos del ascensor y un *token* de autorización. Además, estos datos encajan con los que se registró la etiqueta en la prueba anterior. La prueba ha resultado satisfactoria.

Para la prueba de fallo simplemente realizaremos una petición con un identificador que no esté presente en la base de datos. En la siguiente figura se puede observar la petición y su respuesta y

constatar el buen funcionamiento.

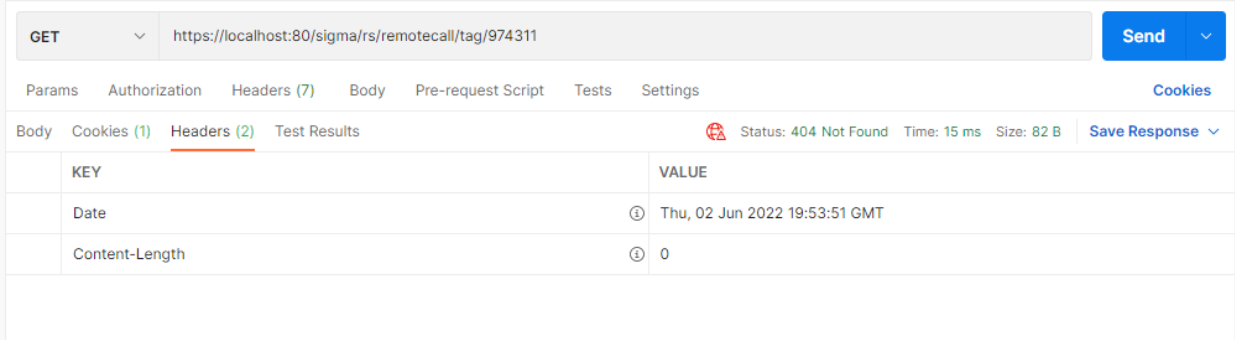


Figura 45: Petición y respuesta de llamada con etiqueta inexistente.

- **[PUT] /sigma/rs/remotecall/{rae}/stops/{planta}/call .** Para probar este *endpoint* usaremos la base de datos en el estado en el que la dejaron las anteriores pruebas.

  En primer lugar, realizaremos una llamada exitosa a una planta disponible del ascensor que está registrado en el servidor. En la siguiente figura podemos ver la petición y su respuesta. Para ello deberemos enviar el *token* recibido en el *endpoint* de llamada a etiqueta de cabina.

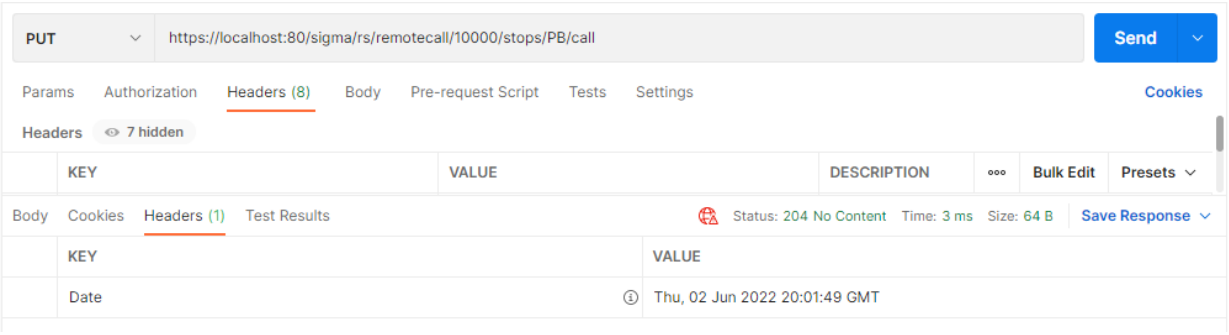


Figura 46: Petición y respuesta exitosa de llamada a planta de un ascensor.

La respuesta HTTPS se trata de un 204 *No Content* y por tanto podemos afirmar que la llamada a la planta ha sido exitosa y que el *endpoint* ha funcionado como esperábamos.

Por otra parte, en la prueba de fallo, comprobaremos que el servidor no nos permite llamar a una planta que se salga de las disponibles del ascensor registrado. En la siguiente imagen se ve la petición con una planta que se sale del rango de paradas del ascensor y su respuesta.

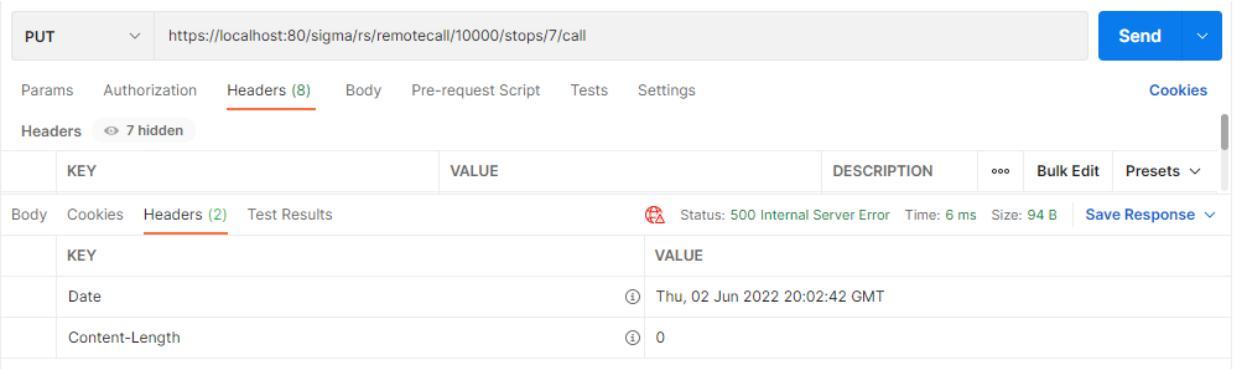


Figura 47: Petición y respuesta de una llamada a una planta no disponible.

Podemos observar que la respuesta del servidor ha sido un 500 *Internal Server Error* y que, por tanto, nuestra prueba ha resultado tal y como esperábamos.

Llegados a este punto, se han realizado pruebas sobre todos los *endpoints* de la API de NFClift Server; resultando satisfactorias todas y cada una de ellas. También se ha comprobado que dichos *endpoints* manejan de forma correcta la base de datos y hacen los registros y lecturas de la manera diseñada. Es por todo ello por lo que podemos afirmar, sin temor a equivocarnos, que NFClift Server y su API funcionan de la manera esperada.

4.4.4 Pruebas de la solución completa

Una vez que hemos comprobado que el funcionamiento de nuestras aplicaciones es el esperado tras realizarles pruebas aisladas, es el momento de probar la solución completa. Dicha prueba constará de dos fases:

- **Despliegue de la solución**. En esta fase simularemos el proceso que un operario tendría que seguir para desplegar NFClift en un ascensor. Para ello tendremos que registrar una etiqueta para la cabina del ascensor y, al menos, una etiqueta para una de las plantas. En esta fase solo usaremos NFClift Operator y NFClift Server.
- **Prueba de usuario**. En esta fase simularemos la experiencia que tendría un usuario que quisiera utilizar el ascensor con nuestra solución. El flujo sería: llamada a planta escaneando la etiqueta ubicada en la planta, escaneo de la etiqueta colocada en cabina y llamada a una planta a través de la botonera virtual. En esta fase solo usaremos la aplicación Android NFClift y NFClift Server.

En ambas fases el escenario de prueba será el detallado en el apartado 4.3 de este mismo documento y el diagrama de red será el representado por la figura 26. En el servidor partiremos de una base de datos totalmente vacía a excepción de la tabla ascensores y la tabla operarios, la cual contendrá un solo identificador autorizado; el "12345".

En mi caso, para que el servidor que está corriendo en el ordenador pueda recibir peticiones HTTPS desde el exterior de mi subred, he tenido que realizar un mapeo de direcciones en la configuración del rúter que le da acceso a internet.

Despliegue de la solución

Desplegaremos la solución en el ascensor con RAE igual a 10, presente en la tabla ascensores de la base de datos de NFClift Server. Para ello, en NFClift Operator introduciremos "12345" como identificador de operador, pulsamos el botón "Registrar etiqueta NFC" y a continuación pulsaremos el botón "Cabina" para registrar la etiqueta que irá colocada en cabina. Para realizar el registro de la etiqueta tan solo tendremos que introducir el número RAE del ascensor en el que va a ir colocada. Introducimos dicho número, acercamos la etiqueta que queremos registrar al dispositivo móvil y pulsamos el botón.

```
Registrada etiqueta en cabina:  4719 10
```

Figura 48: Log en el servidor del registro de la etiqueta de cabina.

En la figura anterior podemos ver el *log* generado en el servidor cuando la etiqueta se ha registrado. Podemos ver el número RAE del ascensor asociado (10) y el identificador de la etiqueta (4719).

Lo único que nos faltaría para terminar de tener desplegada la solución en el ascensor sería registrar una etiqueta NFC por cada planta para colocarlas ahí y que los usuarios puedan llamar al ascensor a dichas plantas

escaneándolas. En una situación real habría que registrar y escribir 4 etiquetas, pero como esto es solo una prueba, registraré tan solo 2 etiquetas puesto que el proceso es siempre el mismo.

Para registrar una etiqueta de planta realizaremos el mismo proceso que el anterior descrito para cabina, pero esta vez seleccionando que la etiqueta irá colocada en planta. Ahora para realizar el registro deberemos introducir el número RAE del ascensor y, además, la planta en la que irá colocada la etiqueta. Para la prueba registraremos y escribiremos las etiquetas que irían colocada en la planta "Sótano" y en la "P2". Para realizarlo, haremos lo mismo que antes; introducimos los datos, acercamos la pegatina al dispositivo móvil y pulsamos el botón. Realizaremos este proceso por cada una de las etiquetas.

```
Registrada etiqueta en planta:  123 10 Sótano
Registrada etiqueta en planta:  4249 10 P2
```

Figura 49: Logs en el servidor de los registros de etiquetas de planta.

En la figura anterior aparecen los *logs* correspondientes al registro que hemos realizado de las dos etiquetas. En estos *logs* se pueden ver el identificador de la etiqueta, el número RAE del ascensor asociado a la etiqueta y la planta en la que irán colocadas.

En este punto podemos dar por concluida la fase de despliegue de la solución en el ascensor. Dispondremos de la etiqueta de cabina y las de planta registradas correctamente.

Prueba de usuario

La prueba de usuario simulará la experiencia que un usuario tendría al usar un ascensor que tenga desplegada nuestra solución. En general, la intención del usuario será llamar al ascensor a la planta en la que esté para luego dirigirse a otra planta. Para realizarlo con nuestra solución desplegada tendrá que seguir los siguientes pasos:

1. Escanear la etiqueta NFC colocada en la planta para que el ascensor sea llamado a dicha planta.
2. Subirse a la cabina y escanear la etiqueta allí colocada para obtener la botonera virtual en el dispositivo móvil.
3. Pulsar en la botonera virtual el botón de la planta a la que se quiere dirigir.

Para la prueba supondremos que el usuario quiere coger el ascensor en la planta "P2" y quiere dirigirse a la planta "Sótano" del edificio. Para simular el primer paso, cogeremos nuestro dispositivo Android, en el que está instalado NFClift, y lo acercaremos a la etiqueta NFC que fue registrada en el despliegue para colocarse en la planta "P2". Recordemos que no es necesario tener abierta la aplicación, al acercar la etiqueta se iniciaría automáticamente.

```
Llamada desde planta
```

Figura 50: Log generado en el servidor al escanear la etiqueta de planta.

Figura 51: Resultado en NFClift al escanear etiqueta de planta.

En las figuras 50 y 51 podemos ver, respectivamente, el *log* generado en el servidor al escanear la etiqueta y el *toast* que muestra NFClift. El primer paso ha sido completado con éxito.

Para simular el segundo paso escanearemos la etiqueta que registramos anteriormente para ser colocada en la cabina. Para ello tan solo tendremos que acercar nuestro dispositivo Android a dicha etiqueta.

```
Llamada desde cabina
```

Figura 52: Log generado en el servidor al escanear la etiqueta de cabina.

Figura 53: Botonera virtual del ascensor.

En las figuras anteriores podemos ver el *log* generado en el servidor y parte de la botonera virtual que se ha

generado en NFClift. Si nos fijamos, podemos ver que los botones que no queremos que sean visibles (los presentes en *stopMask*) no aparecen en la botonera. Observamos que todo ha ido según lo esperado y por tanto el segundo paso está probado con resultado exitoso.

El último paso que queda para terminar de probar la solución es llamar a una planta concreta del ascensor desde la botonera virtual. Como hemos expuesto antes, el supuesto usuario querrá ir a la planta "Sótano" y por tanto pulsará ese botón. Procedemos a realizarlo.

```
Llamada a ascensor  10  a planta  Sótano
```

Figura 54: Log generado en el servidor al llamar a una planta desde la botonera.

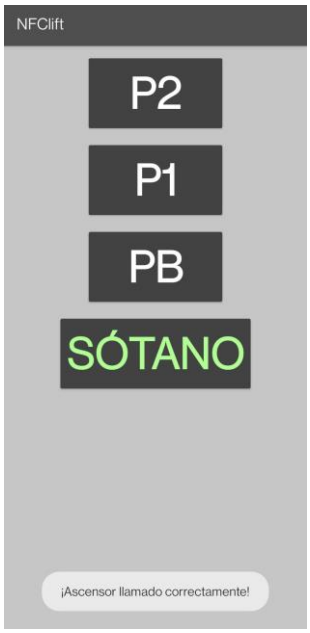


Figura 55: Resultado de NFClift al llamar desde botonera.

Si observamos las figuras anteriores podemos ver que la llamada ha llegado correctamente al servidor y en NFClift nos está dando el *feedback* de que la llamada ha sido realizada correctamente. Esto significa que el tercer paso también ha pasado la prueba satisfactoriamente.

Los tres pasos han sido simulados y todos han dado un resultado exitoso. Esto quiere decir que un usuario real que quisiera usar un ascensor que tuviera nuestra solución desplegada podría usarlo sin ningún problema. Podemos concluir que todas las pruebas sobre nuestra solución han resultado en el comportamiento que diseñamos en un principio.

4.5 Réplica del entorno de pruebas

En este apartado me pararé a detallar el proceso que he seguido a la hora de montar el entorno que he utilizado en la realización de las pruebas de solución completa del apartado anterior. Describiré los pasos que he realizado lo suficientemente claros para que un lector pueda replicar ese mismo entorno y poder probar la aplicación de la misma forma que yo lo he hecho.

Como punto de partida, consideraré que el lector ha seguido los pasos anteriormente descritos en el apartado 3.4 de este mismo documento y que ya dispone del entorno de desarrollo tanto de las aplicaciones Android como del servidor.

Dado que las comunicaciones entre las aplicaciones y el servidor se realizan mediante llamadas HTTPS, cifradas por TLS, nuestras aplicaciones deben fiarse del certificado usado por el servidor. En el apartado 4.3 ya he explicado esta necesidad y su solución con un certificado auto firmado. Para que el lector pueda replicar esto y desplegar el servidor en su red doméstica de manera funcional, deberá generar su propio certificado auto firmado. Esto es debido a que los certificados van asociados a una dirección IP y, por lo tanto, el que yo uso no servirá para una dirección IP distinta a la de mi servidor.

Como mi certificado no va a resultar de utilidad, lo que haremos será borrarlo. Debemos eliminar los archivos *cert.pem* y *key.pem* de la carpeta raíz del código del servidor. Hay muchas formas de crear certificados auto firmados para servidores HTTPS, pero en el caso de NFClift Server está diseñado para generar automáticamente uno si a la hora de iniciarse no encuentra ninguno. Para que el certificado que genere esté vinculado a la IP que queremos debemos cambiar en el código la línea 79 del archivo *service.go* y poner la dirección IP que va a usar nuestro servidor en el sitio que se indica en la siguiente imagen.

```
err := httpscerts.Check( certPath: "cert.pem",  keyPath: "key.pem")
if err != nil {
    err = httpscerts.Generate( certPath: "cert.pem",  keyPath: "key.pem",  host: "85.136.224.37")
    if err != nil {
        log.Fatal( v...: "Error: Couldn't create https certs.")
    }
    //panic("No se encuentra cert.pem junto con su key.pem")
}
log.Fatal(servidor.ListenAndServeTLS( certFile: "cert.pem",  keyFile: "key.pem"))
```

Figura 56: Lugar donde insertar la IP del servidor.

El lector debe tener en cuenta que el objetivo de la prueba es que nuestro servidor sea accesible desde cualquier red y, por lo tanto, necesitamos que nuestro servidor se muestre en una dirección IP pública. En mi caso el servidor lo he desplegado en un ordenador de mi red doméstica, la cual accede a internet a través de una NAT. Para que la comunicación se pueda realizar con el servidor he tenido que realizar un mapeo de direcciones desde la configuración de mi rúter. Es decir, los mensajes con dirección al puerto 80 de la IP pública de mi rúter serán reenviadas a la dirección IP del ordenador dentro de mi subred. De esta forma a ojos de los dispositivos de fuera de mi subred el servidor está desplegado en la IP pública de mi rúter en el puerto 80. Es por ello que la dirección IP a la que está vinculada mi certificado es la dirección IP pública de mi rúter. Con todo esto, tan solo deberemos lanzar la aplicación desde el IDE, generará el certificado y ya podremos pararla.

El siguiente paso es la creación de una imagen de *Docker* con nuestra aplicación encapsulada y sus archivos necesarios para su correcto funcionamiento. Para generar dicha imagen deberemos ejecutar el comando `docker build .` en la consola de comandos una vez ubicados en la carpeta raíz del código del servidor.

El último paso para levantar el servidor será entrar en *Docker Desktop*, acceder a la pestaña de *Images*, darle al botón *Run* al lado de la imagen que hemos generado, configurar el puerto que queremos que use el servidor y volver a darle a *Run*.

En este punto tenemos el servidor totalmente desplegado para el entorno de pruebas. El siguiente paso será entonces desplegar las aplicaciones en el dispositivo Android. Para que las aplicaciones puedan comunicarse con el servidor que hemos desplegado debemos encapsular en las aplicaciones el certificado que hemos creado antes. Lo único que tendremos que hacer será copiar el archivo *cert.pem* generado antes y pegarlo en la carpeta app/src/main/res/raw de cada aplicación. Además, deberemos modificar en el código de ambas aplicaciones la dirección IP donde estará el servidor desplegado. Esta dirección se encuentra en los archivos *LiftInfo.java* y *Utils.java* en NFClift y NFClift Server respectivamente.

```
public static final String urlBaseApiMock = "https://85.136.224.37:80/sigma/rs/remotecall";
```

Figura 57: Lugar donde colocar la IP del servidor al que queremos apuntar.

Lo siguiente será instalar con archivos *apk* las aplicaciones Android en un dispositivo con lector NFC. Los archivos *apk* son paquetes para Android que se usan para la instalación de aplicaciones en este sistema operativo. Lo primero que haremos será generar estos paquetes del código de nuestras aplicaciones. Para ello deberemos abrir el código de la aplicación con Android Studio y seleccionaremos *Build, Bundle(s)/APK(s)* y *Build APK(s)*.

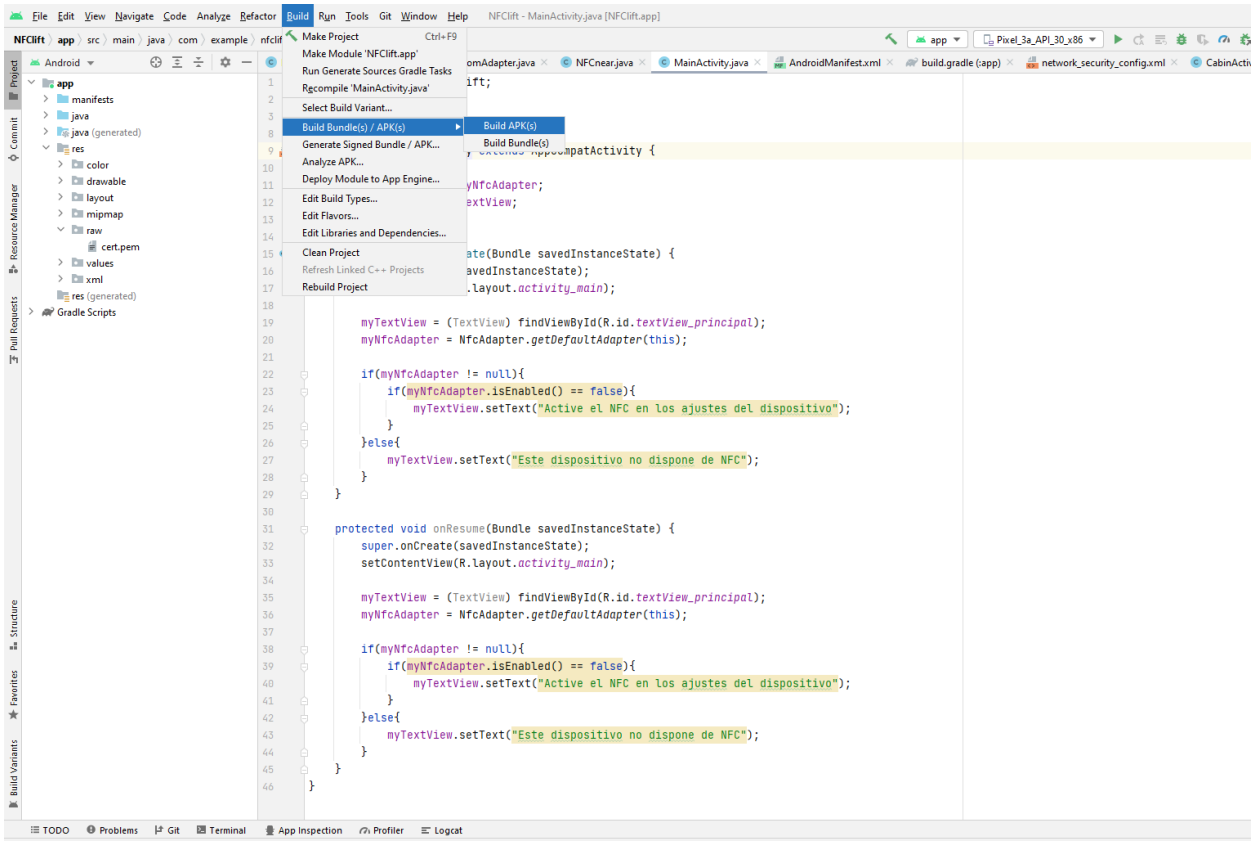


Figura 58: Opción para generar una apk.

El proceso de generado tardará unos minutos y cuando termine se habrá generado un archivo llamado *app-debug.apk* en la ruta por defecto app/build/outputs/apk/debug/. Tendremos que realizar este proceso para ambas aplicaciones para obtener el instalador de cada una. Estos archivos deberemos transferirlos a nuestro dispositivo Android e instalarlos.

En este punto el lector ya tendrá el entorno de pruebas que he utilizado replicado y listo para realizar las pruebas que estime necesarias.

5 CONCLUSIONES

Llegados hasta aquí, es factible decir que los objetivos marcados antes del desarrollo de esta solución han sido cumplidos. Los 3 programas que han sido desarrollados cumplen sus objetivos concretos tanto con su diseño como con su funcionamiento. Repasemos los objetivos cumplidos:

- En el caso de la aplicación NFClift, se ha conseguido crear una aplicación de usuario que permite controlar los ascensores de forma eficaz, sin tener que establecer contacto físico con ellos. Además, al haberse diseñado para que la aplicación se inicie automáticamente al acercarse una etiqueta NFC adecuada, hemos conseguido que el proceso sea muy fácil e intuitivo para el usuario. Esto significa que su introducción en la rutina de los usuarios sería menos traumática y, por lo tanto, más sencilla.
- Con NFClift Operator se ha conseguido una aplicación que nos permite de manera eficaz el despliegue de nuestra solución en los ascensores por parte de los operarios. Realiza el registro de las etiquetas NFC en el servidor de manera efectiva. Además, permite la identificación de los operarios para darle una capa de seguridad extra a nuestra solución.
- NFClift Server no es un software totalmente funcional, pero se ha desarrollado una API y una lógica de base de datos que las otras dos aplicaciones pueden utilizar para el despliegue por parte del operario y para el uso de la solución por parte del usuario. Esta API y base de datos podrían ser implementados en el servidor de la empresa de ascensores para el uso de esta solución en sus ascensores.

Desde un punto de vista más general, podemos decir que nuestra solución ha conseguido paliar las desventajas presentes en la solución actual, detalladas en el apartado 1.3 de este mismo documento. Con nuestra solución se puede identificar inequívocamente el ascensor que el usuario tiene la intención de usar. Además, NFClift no depende del servicio de ubicación para ser funcional. Por otro lado, aunque NFClift Server no sea una aplicación 100% funcional ni esté preparada para la puesta en producción, es un muy buen punto de partida para construir el programa real que operaría en los servidores de la empresa de ascensores.

En la parte personal, el desarrollo de este proyecto me ha hecho conocer el cómo desarrollar aplicaciones Android que hagan uso de una nueva tecnología como es el NFC. Por otra parte, el uso de Go como lenguaje en el servidor y su despliegue en Docker hacen mucho más atractiva la solución puesto que son tecnologías en pleno auge. En relación con esto último, en mi caso ya había usado antes estas tecnologías, pero uso de ellas en este proyecto me ha hecho afianzar conceptos y masterizarlas.

5.1 Líneas de continuación

En este último apartado, listaré los puntos en los que personalmente creo que nuestra solución podría incluir mejoras y que por cuestiones de tiempo o de alcance no se han llegado a realizar.

- **Registro de nuevos operarios autorizados**. Como ya se ha comentado antes, NFClift Server no es un programa que esté preparado para la puesta en producción. Una de las razones de esto, es que no hay implementado un método efectivo para añadir nuevos identificadores de operarios autorizados en la base de datos. Si quisiéramos seguir desarrollando NFClift Server, la primera línea de trabajo a seguir sería la de implementar una forma segura y sencilla de añadir estos identificadores.
- **Migración de NFClift y NFClift Operator a iOS.** Aunque Android es el sistema operativo más extendido para dispositivos móviles, una línea de trabajo interesante sería el migrar nuestras aplicaciones para que fuesen compatibles con dispositivos iOS. Esto permitiría llegar a un número de posibles usuarios incluso mayor.

- **Registro de nuevos ascensores.** NFClift Server no tiene ningún método efectivo de agregar nuevos ascensores a la base de datos a parte de su escritura directa en ella. Un buen punto de partida para convertirlo en un servidor completamente funcional sería éste.

ANEXO A: GUÍA DE INSTALACIÓN Y USO DE NFCLIFT

Guía de instalación

Para instalar NFClift en nuestro dispositivo lo primero que deberemos hacer será descargarnos la *apk* con el siguiente enlace desde el dispositivo en el que queremos instalarlo:

https://drive.google.com/file/d/1kguciV3n3tOpfSq1h4FiffY2S2TQcH4h/view?usp=sharing

Cuando se complete la descarga de dicho archivo, se nos mostrará la opción de abrirlo con el instalador de paquetes del sistema. Según como se tenga configurado el sistema Android, es posible que se tenga que conceder el permiso de instalar *apks* a la aplicación de *Google Drive.* En ese caso, el propio sistema te redirigirá a donde se le puede conceder dicho permiso o, si no, tendremos que buscarlo en el apartado Ajustes del sistema.

Con el permiso de instalación concedidos podemos proceder a instalar la *apk* con el instalador de paquetes del sistema. Nos aparecerá un cuadro como el de la figura a continuación y le daremos a instalar para tener la aplicación en nuestro sistema.

Figura: Cuadro de diálogo para la instalación de NFClift.

Guía de uso

NFClift es una aplicación que está diseñada para que sea muy intuitiva y su uso no sea nada complicado.

Una vez iniciamos la aplicación sólo veremos un texto que nos indicará que tenemos que acercar el dispositivo a la etiqueta NFC del ascensor.

Cuando acerquemos el dispositivo a la etiqueta NFC de la planta del ascensor, el ascensor será llamado a dicha planta y la aplicación mostrará un *toast* indicando el éxito de la llamada y pidiéndonos que escaneemos la etiqueta de dentro de la cabina.

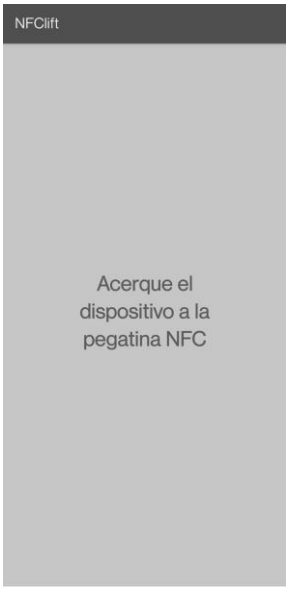


Figura: Interfaz principal de NFClift.

Cuando escaneemos la etiqueta de dentro de la cabina obtendremos una botonera como la siguiente:

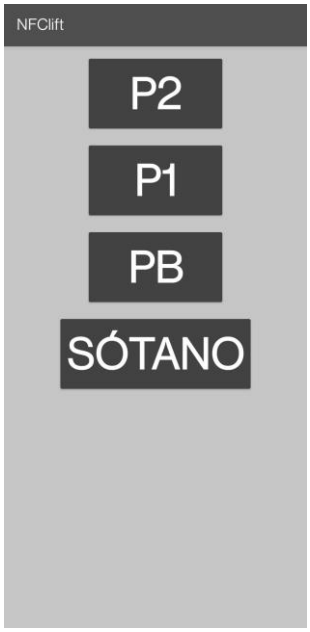


Figura: Botonera virtual del ascensor.

Estos botones corresponden a la botonera real del ascensor. Para que el ascensor se mueva a una planta tan sólo tendremos que pulsar su botón. Dicho botón se pondrá en verde si la acción se ha realizado con éxito.

IMPORTANTE: NFClift apunta a un servidor personal alojado detrás de una NAT y por lo tanto rara vez estará visible al exterior. Esto quiere decir que las llamadas a etiquetas y a las plantas darán error. En el caso de querer probar la aplicación enviar un correo a la dirección palomosivianesjaime@gmail.com indicando una fecha y franja horaria para mantener el servidor accesible.

ANEXO B: GUÍA DE INSTALACIÓN Y USO DE NFCLIFT OPERATOR

Guía de instalación

Para instalar NFClift Operator en nuestro dispositivo lo primero que deberemos hacer será descargarnos la *apk* con el siguiente enlace desde el dispositivo en el que queremos instalarlo:

https://drive.google.com/file/d/1ABYGzeS8qrdFHVU4krvZMFp3DHXw_PTl/view?usp=sharing

Cuando se complete la descarga de dicho archivo, se nos mostrará la opción de abrirlo con el instalador de paquetes del sistema. Según como se tenga configurado el sistema Android, es posible que se tenga que conceder el permiso de instalar *apks* a la aplicación de *Google Drive.* En ese caso, el propio sistema te redirigirá a donde se le puede conceder dicho permiso o, si no, tendremos que buscarlo en el apartado Ajustes del sistema.

Con el permiso de instalación concedidos podemos proceder a instalar la *apk* con el instalador de paquetes del sistema. Nos aparecerá un cuadro como el de la figura a continuación y le daremos a instalar para tener la aplicación en nuestro sistema.

Figura: Cuadro de diálogo para la instalación de NFClift.

Guía de uso

Lo primero que nos aparecerá al iniciar la aplicación es una línea para introducir el identificador de operario con el que querremos registrar las etiquetas. Cuando lo introduzcamos deberemos pulsar el botón.

Figura: Requerimiento de ID del operario.

A continuación, nos aparecerá una pantalla en la que nos preguntará qué acción queremos realizar. Supondremos que hemos pulsado el botón de "Registrar etiqueta". Nos preguntará dónde irá colocada dicha etiqueta: en planta o en cabina.

Figura: Elección de dónde irá la etiqueta.

Según el botón que pulsemos la aplicación nos llevará a un proceso u a otro. En el caso de pulsar el botón cabina la interfaz que encontraremos será la siguiente:

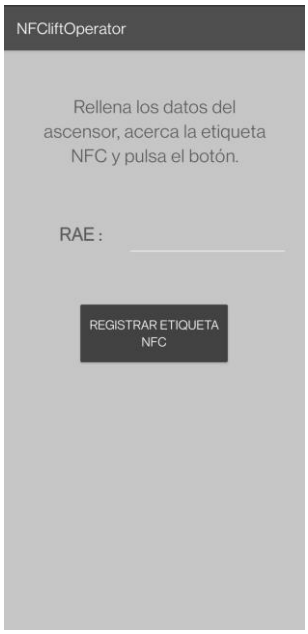


Figura: Registro de etiqueta ubicada en cabina.

Para registrar la etiqueta que queremos colocar en la cabina del ascensor solo tendremos que rellenar el número RAE del ascensor, acercar el dispositivo móvil a dicha etiqueta hasta que notemos una pequeña vibración y pulsar el botón de registrar. La aplicación nos avisará del éxito o el fracaso del registro a través de un *toast.*

En el caso de que hubiéramos pulsado el botón de planta la interfaz que nos encontraríamos sería la siguiente:

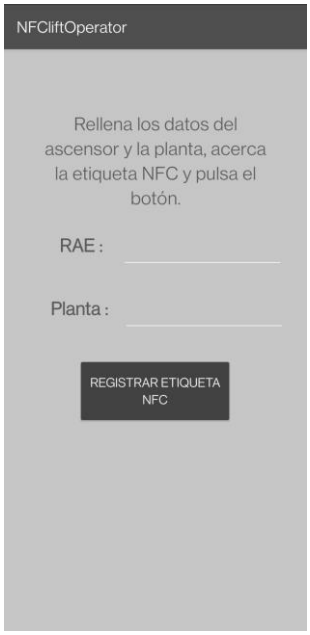


Figura: Registro de etiqueta ubicada en planta.

Para registrar la etiqueta que queremos colocar en la cabina del ascensor solo tendremos que rellenar el número RAE del ascensor y el texto de la planta en la que irá la etiqueta; acercar el dispositivo móvil a dicha etiqueta hasta que notemos una pequeña vibración y pulsar el botón de registrar. La aplicación nos avisará del éxito o el fracaso del registro a través de un *toast.*

Si al inicio de la aplicación pulsamos el botón de “Desvincular etiqueta NFC” la aplicación nos llevará a una nueva pantalla con un nuevo botón con el mismo texto y unas pequeñas instrucciones de lo que hacer. Para desvincular una etiqueta de su acción tan solo tendremos que acercar el dispositivo a dicha etiqueta y pulsar el botón. La aplicación nos avisará del éxito o el fracaso del proceso a través de un *toast.*

IMPORTANTE: NFClift Operator apunta a un servidor personal alojado detrás de una NAT y por lo tanto rara vez estará visible al exterior. Esto quiere decir que el registro y la desvinculación de etiquetas darán error. En el caso de querer probar la aplicación enviar un correo a la dirección palomosivianesjaime@gmail.com indicando una fecha y franja horaria para mantener el servidor accesible.

ANEXO C: GUÍA DE INSTALACIÓN Y USO DE NFCLIFT SERVER

Guía de instalación

Para instalar NFClift Server en el dispositivo que quiera actuar como servidor deberemos descargar el código desde la siguiente dirección:

https://drive.google.com/file/d/1O5iFan85MBUzqBMclkbPmnhDe82niDRn/view?usp=sharing

Una vez descargado el archivo anterior tendremos que descomprimirlo en una carpeta. Lo siguiente que haremos será crear la imagen de Docker. Para ello nuestro sistema tendrá que tener instalado Docker. En caso de que no sea así, descargar desde el siguiente enlace:

https://www.docker.com/products/personal/

Con Docker instalado en el dispositivo, abrir la consola de comandos, situarse en la carpeta donde hemos descomprimido el código y ejecutar el siguiente comando:

```
docker build -t nfcliftserver .
```

En este punto ya tendremos el contenedor de Docker con nuestro servidor encapsulado.

Guía de uso

Para lanzar el servidor tan solo tendremos que abrir la consola de comandos y ejecutar el siguiente comando:

```
docker run nfcliftserver -p 80
```

En este punto el servidor ya estará levantado y escuchando peticiones en el puerto 80 del sistema.

REFERENCIAS

[1] Xataka. NFC: qué es y para qué sirve:

https://www.xataka.com/moviles/nfc-que-es-y-para-que-sirve

[2] El Español. Los códigos QR y el pago NFC se asientan en la vida diaria de los españoles tras la covid-19:

https://www.elespanol.com/invertia/disruptores-innovadores/politica-digital/espana/20210707/codigos-qr-nfc-asientan-diaria-espanoles-covid-19/594440865_0.html

[3] Distribuciones y arquitecturas compatibles con Android Studio:

https://developer.android.com/studio#downloads

ENLACES DE INTERÉS

- Programa de higiene de manos del SNS

  https://seguridaddelpaciente.es/es/practicas-seguras/programa-higiene-manos/
- ISO 18092:2013

  https://www.iso.org/standard/56692.html
- Web oficial de Go:

  https://go.dev
- ¿Qué es una API REST?:

  https://www.ibm.com/es-es/cloud/learn/rest-apis
- NFC Tools en la Play Store:

  https://play.google.com/store/apps/details?id=com.wakdev.wdnfc&hl=es&gl=US
- Web oficial de Postman:

  https://www.postman.com
- Página oficial de DB Browser for SQlite:

  https://sqlitebrowser.org
- Herramienta online usada para la creación de los diagramas:

  https://www.lucidchart.com/pages/es
- Cómo usar etiquetas NFC en Android:

  https://code.tutsplus.com/es/tutorials/reading-nfc-tags-with-android--mobile-17278
- Cómo usar API REST en Android:

  https://code.tutsplus.com/tutorials/android-from-scratch-using-rest-apis--cms-27117
- Cómo crear un *toast* dentro de un *worker thread*:

  https://stackoverflow.com/questions/3875184/cant-create-handler-inside-thread-that-has-not-called-looper-prepare
- Cómo usar OkHttp para peticiones HTTP:

  https://www.youtube.com/watch?v=oGWJ8xD2W6k
- Editor de iconos para Android:

  https://easyappicon.com
- Cómo crear un servidor web en Go:

  https://parzibyte.me/blog/2018/12/03/servidor-web-go/

  https://medium.com/@evers.rivero/https-y-golang-2db1d2739e7e
- Cómo usar la librería mux en Go:

  https://parzibyte.me/blog/2019/05/30/enrutador-middleware-go-gorilla-mux/#Instalar_Go_y_Gorilla_mux

- Como implementar autenticacion por tokens JWT en API REST en Golang: https://www.sohamkamani.com/golang/jwt-authentication/
- Enlace de descarga de Android Studio: https://developer.android.com/studio
- Enlace de descarga de Git: https://git-scm.com/downloads
- Enlace de descarga de GoLand: https://www.jetbrains.com/es-es/go/download/#section=windows
- Librería gson: https://github.com/google/gson
- Librería OkHttp: https://square.github.io/okhttp/