Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Desarrollo de una aplicación móvil para la creación de una herramienta de control de asistencia mediante NFC

ESCUELA POLITECNICA SUPERIOR

Autor: Eduardo Graván Serrano

Tutor/es: Ana Castillo Martínez

2020

UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior

GRADO EN INGENIERÍA INFORMÁTICA

Trabajo Fin de Grado
Desarrollo de una aplicación móvil para la creación de una herramienta de control de asistencia mediante NFC

Autor: Eduardo Graván Serrano

Tutor/es: Ana Castillo Martínez

TRIBUNAL:

Presidente: << Nombre y Apellidos >>

Vocal 1º: << Nombre y Apellidos >>

Vocal 2º: << Nombre y Apellidos >>

FECHA: << Fecha de depósito >>

Agradecimientos

En primer lugar, me gustaría dedicar unas palabras de agradecimiento a mis padres. A lo largo de toda mi vida han luchado por mi educación y sé que mi graduación les llenará de orgullo tanto como a mí.

Me gustaría agradecer también a Ana Castillo, mi tutora, por haberme guiado a lo largo de los meses de desarrollo de todo el proyecto.

Gracias a todos los amigos y familiares que he usado de soporte a la hora de debugear el código a lo largo de todo el desarrollo, aunque en muchas ocasiones no entendiesen nada de lo que estaba hablando.

Por último, me gustaría dar las gracias a la comunidad de desarrolladores de Android y en especial a *StackOverflow* por ser un pilar fundamental a la hora de resolver dudas y aportar ejemplos para ayudarme a construir las bases del proyecto.

Índice

Resumen 12

Abstract 13

1. Introducción 14

2. Objetivo 16

3. Estado del arte 17

3.1. Estudio del uso de dispositivos móviles como herramienta de control de asistencia .. 17
3.2. Estudio de aplicaciones existentes en el mercado para el control de asistencia 17
3.3. Estudio de la tecnología NFC 19
3.4. Sistema Android y API de NFC 25
3.4.1 Visión general de la API de Android 26
3.4.2 API de Android para NFC 29

4. Desarrollo del sistema 33

4.1. Arquitectura del sistema 33
4.2. Base de datos 34
4.3. Servidor HTTP 36
4.4. Aplicación Android 44
4.4.1. Login, información de usuario y conexión con el servidor HTTP 45
4.4.2. Actividad y servicio de emulación de etiquetas 49
4.4.3. Actividad de lectura de etiquetas 57
4.5. Aplicación de escritorio para administradores 61

5. Conclusiones 66

6. Trabajo futuro 67

7. Coste del proyecto 68

8. Bibliografía 71

9. Anexo A - Manual de usuario 73

9.1. Aplicación Android 73
9.1.1. Empleado 74
9.1.2. Administrador 79
9.2. Panel de administración 83

10. Anexo B – Elementos adicionales entregables 93

Índice de Imágenes

Figura 1. Usuarios de Smartphone en España vs Internautas – Ditrendia [3]. 15
Figura 2. Sistema físico de identificación Odoo. 18
Figura 3. Interfaz de la aplicación de Android de Cloud TnA. 18
Figura 4. Logo de NFC. 19
Figura 5. Etiqueta NFC. 20
Figura 6. Ventas de dispositivos móviles con soporte para NFC – Bluebite [10]. 20
Figura 7. Estructura de un mensaje NDEF. 21
Figura 8. Logo de Android. 25
Figura 9. Arquitectura del sistema Android. 26
Figura 10. Ciclo de vida de una actividad Android. 27
Figura 11. Ciclo de vida de un servicio Android. 28
Figura 12. Esquema de funcionamiento de Android Beam. 30
Figura 13. Emulación de etiquetas NFC en Android. 30
Figura 14. Funcionamiento de Tag Dispatch System. 31
Figura 15. Visión general del sistema. 33
Figura 16. Modelo de datos. 34
Figura 17. Logo de SQLite. 36
Figura 18. Logos de Python3 y el framework Flask. 37
Figura 19. Visión general de la GUI de Swagger. 42
Figura 20. Ejemplo de llamadas a la API con la GUI de Swagger. 42
Figura 21. Ejemplo de llamada con la GUI de Swagger. 43
Figura 22. Ejemplo de respuesta con la GUI de Swagger. 43
Figura 23. Vista general de la API ReST Implementada. 44
Figura 24. Función checkLoginStatus. 46
Figura 25. Establecimiento de las SharedPreferences. 46
Figura 26. Permisos necesarios para el uso de conexiones de red. 46
Figura 27. Lanzamiento de una AsyncTask. 47
Figura 28. Código de ejemplo de una petición HTTP en una AsyncTask. 48
Figura 29. Ejemplo de función onPostExecute. 48
Figura 30. Ejemplo de actualización de la interfaz de usuario con la información recogida de la petición HTTP. 49
Figura 31. Inicialización del servicio de emulación de etiquetas. 50
Figura 32. Método encargado de parar el servicio de emulación de etiquetas. 50
Figura 33. Método onResume de la actividad de emulación de etiquetas. 50
Figura 34. Método onPause de la actividad de emulación de etiquetas. 51
Figura 35. Método encargado de recoger los broadcasts lanzados por el servicio de emulación de etiquetas. 51
Figura 36. Declaración de permisos de NFC. 52
Figura 37. Declaración de características necesarias (NFC y HCE). 52
Figura 38. Descriptor del servicio de emulación de etiquetas. 52
Figura 39. Archivo XML que recoge los AID asociados al servicio de emulación de etiquetas. 53
Figura 40. Método onStartCommand del servicio de emulación de etiquetas. 53
Figura 41. Método auxiliar createTextRecord del servicio de emulación de etiquetas. 54
Figura 42. Ejemplo de comandos APDU. 54
Figura 43. Ejemplo de respuestas APDU. 55
Figura 44. Comprobación de la inicialización del servicio de emulación de etiquetas. 55
Figura 45. Ejemplo de procesamiento de los C-APDU. 55
Figura 46. Creación de R-APDU con el mensaje NDEF. 56
Figura 47. Respuesta con R-APDU de error. 56

Figura 48. Método encargado de enviar los broadcasts desde el servicio de emulación de etiquetas. 57
Figura 49. Método encargado de activar ReaderMode. 58
Figura 50. Creación del AlertDialog de lectura de etiquetas........ 59
Figura 51. Método encargado de desactivar ReaderMode. 59
Figura 52. Primera parte del método encargado de recoger las etiquetas leídas por el lector..... 60
Figura 53. Segunda parte del método encargado de recoger las etiquetas leídas por el lector.... 60
Figura 54. Análisis de la respuesta del servidor frente a la lectura de etiquetas y actualización de la interfaz de usuario. 61
Figura 55. Ejemplo de recogida y formateo de datos en la aplicación de administración. 63
Figura 56. Ejemplo de conexión con el servidor HTTP en la aplicación de administración. 63
Figura 57. Ejemplo de procesamiento de respuesta del servidor HTTP en la aplicación de administración. 63
Figura 58. Función encargada de rellenar un objeto de tipo jTable. 64
Figura 59. Ejemplos de métodos encargados de gestionar el evento de pulsar un botón en la interfaz........ 64
Figura 60. Procesamiento de la respuesta del servidor HTTP en el caso del login. 65
Figura 61. Capturas del panel de login en la aplicación Android. 73
Figura 62. Menú desplegable dentro de la pantalla principal de la aplicación Android. 74
Figura 63. Menú principal de la aplicación Android para usuarios no administradores. 74
Figura 64. Menú de emulación de etiquetas. 75
Figura 65. Emulación de etiquetas en proceso. 76
Figura 66. Posibles respuestas ante la emulación de etiquetas. 77
Figura 67. Menú de comprobar horarios para usuarios no administradores en la aplicación Android. 78
Figura 68. Posibles respuestas del menú de comprobar horario. 78
Figura 69. Menú principal de la aplicación Android para usuarios administradores. 79
Figura 70. Lectura de etiquetas en proceso. 80
Figura 71. Mensajes de fichaje correcto en la aplicación. 80
Figura 72. Mensajes de fichaje erróneo en la aplicación. 81
Figura 73. Menú de registro de nuevos empleados para usuarios administradores. 82
Figura 74. Posibles errores a la hora de rellenar el formulario de registro de nuevos empleados. 82
Figura 75. Posibles respuestas del servidor ante el intento de creación de un nuevo empleado a través de la aplicación Android. 83
Figura 76. Pantalla de login en la aplicación de escritorio. 84
Figura 77. Mensajes de error ante el inicio de sesión en la aplicación de escritorio. 84
Figura 78. Mensaje de inicio de sesión correcto en la aplicación de escritorio. 84
Figura 79. Pestañas del menú principal en la aplicación de escritorio para administradores...... 85
Figura 80. Menú de alta de empleado y posibles respuestas de la aplicación. 86
Figura 81. Menú de baja de empleado y respuesta correcta del servidor. 86
Figura 82. Menú de información de usuario y respuesta de la aplicación. 87
Figura 83. Menú de creación de horarios. 87
Figura 84. Posibles respuestas del servidor ante la creación de un nuevo horario para un empleado. 88
Figura 85. Menú de eliminación de un horario para un empleado. 88
Figura 86. Menú de consulta de información de horarios de empleados. 89
Figura 87. Menú de comprobación de asistencia de un empleado. 90
Figura 88. Menú de análisis de horas trabajas por un empleado. 91
Figura 89. Posibles respuestas después de rellenar el formulario de análisis de horas de empleados. 91
Figura 90. Respuesta de la consulta de horas trabajadas de un empleado en un mes con faltas. 92

Figura 91. Respuesta de la consulta de horas trabajadas de un empleado en un mes con horas extra. 92

Índice de Tablas

Tabla 1. Tipos de etiquetas NFC – ST25 NFC Guide [12]. 22
Tabla 2. Desglose de costes de mano de obra del proyecto. 68
Tabla 3. Desglose del coste hardware. 69
Tabla 4. Desglose del coste de licencias software.. 69
Tabla 5. Coste total de materiales. 70
Tabla 6. Gastos generales del proyecto. 70
Tabla 7. Desglose del coste global del proyecto. 70

Resumen

Se ha desarrollado una aplicación Android para el control de asistencia y horarios para los empleados de una organización. La aplicación está basada en el uso de las tecnologías NFC aprovechando su implantación en dispositivos móviles.

Para cubrir la necesidad de gestionar los datos generados por la aplicación, se ha creado una aplicación de escritorio para administradores, abstrayendo las consultas a la base de datos a través de una interfaz gráfica.

Ambas aplicaciones se apoyan en un servidor HTTP que implementa una API ReST para hacer de intermediario con la base de datos.

Palabras Clave

Android, NFC, ReST API, Attendance management, Schedule control for employees

Abstract

An Android application has been developed to control attendance and schedules data for the employees of an organization. The application is based on the use of NFC technologies, taking advantage of its implementation on mobile devices.

To cover the need to manage the data generated by the application, a desktop application for users with administrative privileges has been created, abstracting the queries to the database through a graphical interface.

Both applications rely on an HTTP server that implements a ReST API to act as an intermediary between the database and the rest of the system.

Key words

Android, NFC, ReST API, Attendance management, Schedule control for employees

1. Introducción

El control de la asistencia de empleados es una necesidad para las empresas de cara a conocer y hacer un seguimiento sobre las horas trabajadas por los empleados.

Esta necesidad de controlar la asistencia de los empleados de una empresa se ve consolidada a través del Real Decreto-ley 8/2019, de 8 de marzo, de medidas urgentes de protección social y de lucha contra la precariedad laboral en la jornada de trabajo [1]. Esta orden obliga a las empresas a mantener un control horario de todos sus trabajadores, independientemente de su jornada. Esto va enfocado no solo a conocer la duración de la jornada ordinaria, sino también para hacer un seguimiento completo de las horas extras que hagan los trabajadores.

Debido a este Decreto-Ley, muchas empresas están buscando nuevas formas de automatizar este control horario a través de distintos sistemas basados en software. Aquel en el que se centrará este proyecto es el control de horario/asistencia haciendo uso de tecnologías móviles y etiquetas de NFC.

La tecnología NFC, ya sea a través de chips integrados en dispositivos móviles, o cualquier otro tipo de etiqueta NFC (en pulseras, llaveros, etc.), es perfecta para el desarrollo de un sistema de control de asistencia gracias a la rapidez y eficiencia implicados en el intercambio de información inherentes a esta tecnología, sirviendo también como una forma de asegurar la presencialidad del empleado a la hora de fichar en su puesto de trabajo.

A parte de la posibilidad de tener etiquetas físicas que almacenen datos sobre los empleados para fichar en el puesto de trabajo, Android tiene una API bastante extensa encargada de explotar al máximo los chips NFC de los teléfonos móviles. A través del uso de esta API, se pueden crear sistemas software para crear sistemas de gestión de la asistencia, verificaciones de presencia, y creación de registros horarios.

Según un informe presentado por Deloitte [2] en 2017, la tasa de penetración de los dispositivos móviles inteligentes ha llegado a un 92% de la cuota de mercado en España, mientras que un estudio realizado por Ditrendria [3] en 2019, afirmaba que durante los últimos años la tasa de penetración había aumentado hasta un 96-97% de la población.

Usuarios de Smartphone en España vs. Internautas

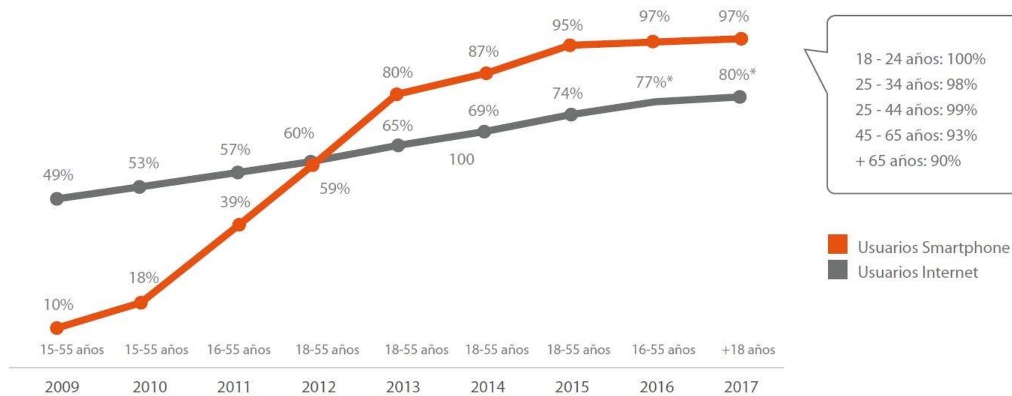


Gráfico elaborado por ditrendia a partir de datos de EGM

ditrendia digital marketing trends

Figura 1. Usuarios de Smartphone en España vs Internautas – Ditrendia [3].

Apoyándonos en estudios e informes como estos, podemos afirmar que prácticamente toda la población tiene acceso a dispositivos móviles, y que la extensión de su uso sigue estando en crecimiento. Esto cualifica a los dispositivos móviles perfectamente para ser utilizados como una herramienta para el control de asistencia y presencia.

Por todo esto, se ha desarrollado un sistema software que facilite y automatice el control de asistencia para una institución. Este sistema hace uso de la tecnología NFC aprovechando su implantación en dispositivos móviles a través de una aplicación para el sistema operativo Android.

Aun así, se entiende que no todos los teléfonos móviles Android cuentan con esta tecnología, por lo que se ha hecho un análisis de las posibles soluciones en busca de una solución que satisfaga la necesidad de crear un sistema de estas características apoyado en tecnologías móviles y NFC, pero sin dejar fuera a aquellos usuarios que no cuenten con un dispositivo con este chip.

La memoria recoge el proceso de estudio de las diferentes alternativas posibles a la hora de crear el sistema, explicando en detalle la solución tomada finalmente. Se presenta la documentación del sistema completo, haciendo hincapié en sus partes más importantes, así como un manual de usuario que explica el funcionamiento de las interfaces de usuario creadas.

2. Objetivo

El objetivo de este trabajo es el desarrollo de una aplicación que permita a una empresa/institución el control de los horarios de sus empleados, haciendo uso de tecnologías móviles.

Para conseguir esto, se ha hecho uso de la API de NFC para Android, aprovechando que la tecnología NFC está implantada en muchos dispositivos móviles actuales. Para hacer uso de la tecnología NFC desde Android, se ha creado una aplicación Android basada en NFC que cuenta con las capacidades de emular etiquetas NFC y leer etiquetas NFC.

Haciendo uso de esta aplicación, el trabajador podrá usar su teléfono móvil para fichar al entrar a su puesto de trabajo. Simplemente haciendo lo mismo a la hora de salir, se podrá realizar un control horario total sobre el trabajador, haciendo que el proceso de fichar no tome más de unos segundos y sea totalmente automático, además de permitir a los empleadores saber a ciencia cierta que el empleado se encontraba en su puesto de trabajo a la hora de fichar, debido a la presencialidad inherente a la tecnología NFC.

Los administradores deberán ser capaces de acceder y gestionar todos los datos que almacene la aplicación. Esto es, se deberá crear una base de datos que guarde registros sobre los siguientes datos:

- Información sobre empleados
- Información sobre horarios de empleados
- Información sobre datos de asistencia de empleados

Para facilitar el acceso y gestión de todos los datos generados por la aplicación, se desarrollará una aplicación de escritorio con interfaz gráfica disponible para los usuarios administradores. Esta aplicación hará de intermediario entre el *back-end* del sistema y los usuarios administradores, eliminando la necesidad de sus usuarios de tener conocimientos sobre bases de datos o programación.

Con estas partes, se ha creado un sistema completo y autosuficiente capaz de gestionar todos los registros de asistencia y horarios de los empleados de una organización.

3. Estado del arte

En este apartado se hará un estudio de algunas soluciones existentes en el mercado hoy en día para el problema planteado, así como un estudio de las tecnologías disponibles para crear estas soluciones.

3.1. Estudio del uso de dispositivos móviles como herramienta de control de asistencia

Haciendo un estudio del estado del arte en el campo de la gestión de asistencia de empleados a través del uso de tecnologías NFC se han encontrado varios documentos que teorizan sobre el uso de NFC y Android para suplir la necesidad del control de asistencia de empleados.

Entre estos documentos se encuentran *Cloud-based web application with NFC for employee attendance management system* [4] y *Near Field Communication (NFC) based Mobile Phone Attendance System for Employees* [5].

El primer artículo [4] propone, de forma teórica, la creación de un sistema por el cual los empleados de una organización utilizarían etiquetas NFC físicas para fichar en un dispositivo Android con chip NFC puesto en modo lectura. Al fichar, el dispositivo Android que está haciendo de lector de etiquetas, usaría la cámara para hacer un reconocimiento facial del empleado y enviaría los datos a una base de datos alojada en la nube.

El segundo artículo [5] es más antiguo y propone de forma teórica la combinación de la tecnología NFC con dispositivos móviles para crear sistemas de control de asistencia y horarios de empleados para una organización. Se hace una comparativa con la tecnología Bluetooth y después se plantea un sistema teórico en el cual los empleados ficharían con tarjetas físicas en un dispositivo móvil con NFC que se encargaría de enviarle los datos a una base de datos remota.

3.2. Estudio de aplicaciones existentes en el mercado para el control de asistencia

En cuanto a las aplicaciones empresariales que utilizan NFC para el control de asistencia de empleados, se han estudiado las siguientes:

En primer lugar, *landoo* [6] es una empresa española que, entre otros productos, vende un sistema para el control de asistencia de empleados que proporciona una gran cantidad de posibilidades para los empleados a la hora de fichar, el producto se llama *Odoo*. Una de estas soluciones es el uso de etiquetas NFC. A diferencia del proyecto descrito en esta memoria, las etiquetas NFC son etiquetas físicas que fichan en un lector de etiquetas específico, es decir, no hacen uso de la tecnología Android como forma de abstracción de NFC.

El sistema cuenta con un lector de etiquetas NFC conectado a una Raspberry Pi con pantalla. Cuando el usuario ficha se le muestra por la pantalla de la Raspberry el resultado del fichaje. En la siguiente figura se pueden ver estos componentes en un entorno real.

Figura 2. Sistema físico de identificación Odoo.

En segundo lugar, se ha encontrado un sistema llamado *Cloud TnA* [7]. La empresa responsable de este sistema vende la implantación de un sistema de control de asistencia, alojado en la nube. A la hora de generar los registros de control de asistencia, el sistema se apoya en la tecnología NFC implantada a través de una aplicación de Android. Esta aplicación recoge la información del usuario, la fecha y hora del registro horario, y otros datos como la ubicación del usuario en el momento en que se ficha, y se los envía al servidor alojado en la nube. En este caso, también se depende de etiquetas NFC físicas.

Figura 3. Interfaz de la aplicación de Android de Cloud TnA.

En todos los casos estudiados, la tecnología Android es simplemente utilizada como lector de etiquetas NFC, no se ha encontrado ningún caso en el que el dispositivo Android se use como emulador de tarjetas NFC. La posibilidad de emular etiquetas NFC con Android (HCE) existe desde Android 4.4 y será utilizada para el desarrollo del sistema descrito en la memoria de este proyecto, ya que permite eliminar la necesidad del empleado de tener una etiqueta NFC física que deba llevar todos los días a su puesto de trabajo.

3.3. Estudio de la tecnología NFC

El término **NFC** [8] [9] (Near Field Communication) hace referencia a un protocolo de transmisión de datos de corto alcance basado en la tecnología de radiofrecuencia **RFID**. Como indica su nombre, la tecnología NFC tiene muy poco alcance operativo; dependiendo de la implementación, se puede tener una distancia máxima de entre 5 y 10 centímetros.

Figura 4. Logo de NFC.

Debido a la facilidad con la que se puede implementar, la implantación de esta tecnología en dispositivos móviles se ha incrementado en gran medida durante los últimos años, pasando a ser prácticamente un estándar. Esto permite que se puedan desarrollar aplicaciones para estos terminales con los cuales explotar la tecnología al máximo, eliminando en gran medida la necesidad de etiquetas físicas que porten las etiquetas NFC.

Los dispositivos que cuenten con tecnología NFC pueden actuar en distintos modos de operación:

- **Emulación de etiqueta:** el dispositivo móvil emula la funcionalidad de una etiqueta NFC, compartiendo la información para que otros dispositivos que estén dentro de su rango puedan leer la información de la etiqueta NFC virtual.

- **Modo lectura/escritura:** El modo lectura permite al dispositivo ponerse en modo de escucha esperando que etiquetas NFC entren dentro de su rango de operación para leer su información. Por otro lado, el modo escritura nos permite escribir información a la etiqueta que entra dentro del rango de operación. Para poder escribir sobre estas etiquetas, se necesita de software especial capacitado para hacerlo.

- **Modo peer-to-peer:** se crea una red entre los dos dispositivos conectados por NFC. Esto permite establecer un "handshake" entre ambos dispositivos, posibilitando la compartición de datos de cualquier tipo, así como conexión Wi-Fi, bluetooth, etc.

Las etiquetas NFC son dispositivos pasivos que cuentan con una pequeña memoria en la cual almacenan la información que será leída por otros dispositivos activos, cuentan también con una pequeña CPU y una antena. Los elementos activos en la conexión NFC se encargan de dar la corriente eléctrica necesaria a las etiquetas para que se activen y poder leer sus contenidos de esta forma. Esto permite que se pueden almacenar etiquetas NFC en elementos como pulseras, tarjetas, colgantes, etc. sin necesidad de una conexión permanente o baterías.

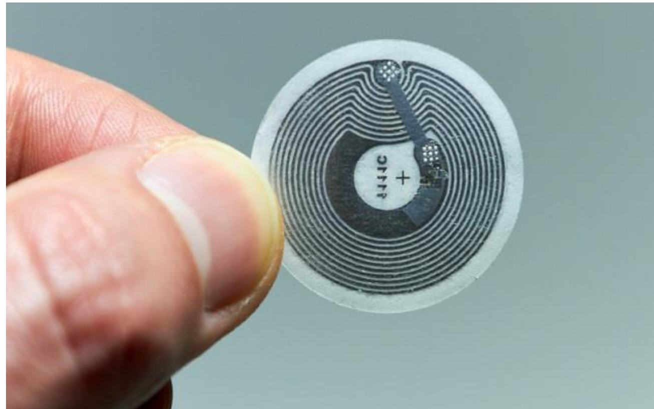


Figura 5. Etiqueta NFC.

En principio las etiquetas se encuentran en modo de solo lectura, pero pueden ser escritas para almacenar la información. Si así se desea, se puede configurar una etiqueta para que solo pueda ser escrita una vez, haciendo que sea imposible sobrescribir los datos si ya ha sido escrita anteriormente. Las etiquetas pueden almacenar todo tipo de información, y cuentan con un almacenamiento de entre 48 bytes y 1 Megabyte de memoria.

En los últimos años se ha visto un gran incremento en el número de empresas que han decidido adoptar e implementar la tecnología NFC en sus dispositivos. Esto se puede ver reflejado en la siguiente figura:

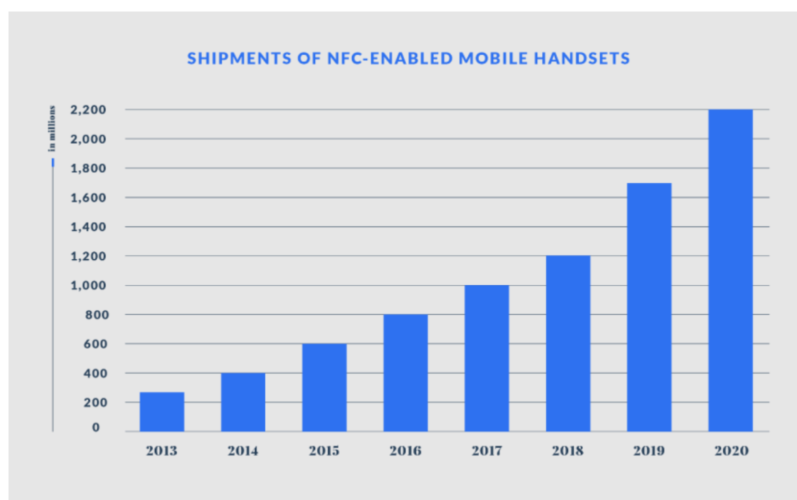


Figura 6. Ventas de dispositivos móviles con soporte para NFC – Bluebite [10].

Los principales usos de la tecnología NFC son los siguientes:

- **Pago con NFC:** muchos servicios de pago como Google Wallet o Apple Pay hacen uso de la tecnología NFC para permitir a sus usuarios pagar directamente a través de sus dispositivos móviles. Esta es sin lugar a duda el uso más extendido de la tecnología NFC.

- **Identificación:** la tecnología NFC puede ser utilizada para la autenticación e identificación de usuarios a través de estas etiquetas, pudiendo reemplazar otros sistemas como contraseñas, pines, etc.

- **Seguimiento e identificación de productos:** se le pueden asignar etiquetas NFC a productos importantes para almacenar información referente al producto y que pueda ser identificado en todo momento.

- **Control de asistencia:** se puede tomar provecho de esta tecnología para crear etiquetas identificadoras para cada usuario de cierto sistema, pudiendo crear aplicaciones para el control de asistencia, asegurando la presencialidad del usuario.

Debido al espacio limitado de memoria en estos chips, las etiquetas NFC suelen servir URLs con más información o registros que tienen solamente texto. Para asegurar la interoperabilidad entre las distintas implementaciones de la tecnología NFC, así como entre distintos softwares con estas etiquetas, la tecnología NFC cuenta con un estándar que indica el formato del texto almacenado en las etiquetas. Este estándar se conoce como NDEF (NFC Data Exchange Format).

El formato NDEF (NFC Data Exchange Format) es un formato de datos estandarizado para compartir información entre un dispositivo NFC y otro dispositivo NFC o etiqueta NFC compatibles. Para el uso del formato NDEF se tienen que usar protocolos y etiquetas NFC estandarizadas.

La comunicación a través de NDEF se divide en:

- **Mensajes NDEF:** son la unidad base de la comunicación basada en NDEF. Cada mensaje de NDEF puede contener uno o más registros NDEF.

- **Registros NDEF:** los registros NDEF trabajan a nivel de byte y tienen una cabecera en la cual se especifican datos como la longitud del paquete, el tipo de datos que contiene el paquete, etc. y el propio "payload" o "mensaje" que se quiere servir a través de la comunicación.

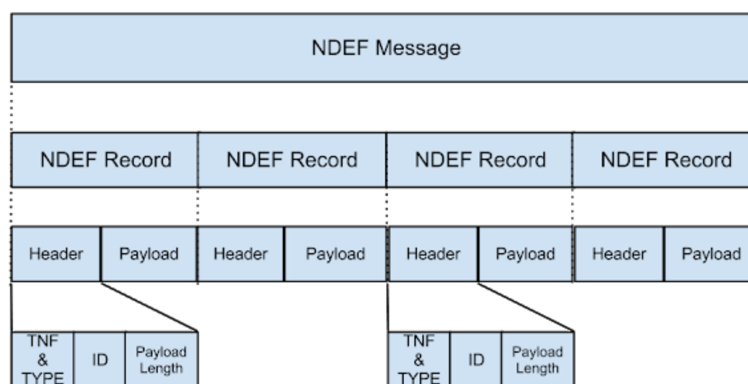


Figura 7. Estructura de un mensaje NDEF.

Hay cinco tipos de etiqueta NFC estandarizadas que pueden implementar el formato NDEF para compartir datos. Los estándares para estos tipos de etiqueta están definidos por el foro NFC y sus principales características son las siguientes [11]:

- **NFC Forum Type 1 Tag:** este tipo de etiquetas son las más simples. Gracias a esta simplicidad tienen una mayor capacidad de memoria que las de tipo 2. Suelen ser utilizadas para tarjetas de un solo uso, enlaces con aplicaciones bluetooth, tarjetas de negocios en formato NFC.

- **NFC Forum Type 2 Tag:** es el tipo de etiqueta más barato. Son más rápidas que las de tipo 1 y suelen utilizarse para almacenar URLs, transacciones de poco valor, etc.

- **NFC Forum Type 3 Tag:** las etiquetas de tipo 3 siguen un estándar distinto que el resto de las etiquetas. Son muy populares en Asia debido a la amplia funcionalidad que proporcionan. Son usadas principalmente para almacenar dinero electrónico, identificación electrónica, tarjetas de membresía, etc.

- **NFC Forum Type 4 Tag:** son las etiquetas más flexibles y con mayor capacidad de memoria de entre todas las etiquetas. Es la única etiqueta que soporta el estándar de seguridad ISO 7816, permitiendo una autenticación total del usuario. Permite también que las etiquetas NFC auto modifiquen sus mensajes NDEF. Debido a la gran capacidad, se suelen usar para crear aplicaciones de compra de tickets de tránsito (pagos en autobús, metro, tren, etc.).

- **NFC Forum Type 5 Tag:** es el tipo de etiqueta más reciente. Soporta el estándar ISO/IEC 15693. Permite el uso de comunicación en modo activo, esto es, que tanto el emisor como el receptor tengan fuentes de corrientes propias. Se suelen utilizar para libros en bibliotecas, tickets electrónicos y paquetería.

La siguiente tabla recoge las principales características técnicas de los tipos de etiquetas a modo de resumen.

Tabla 1. Tipos de etiquetas NFC – ST25 NFC Guide [12].

Propiedad	Type 1	Type 2	Type 3	Type 4	Type 5
Estándar	ISO/IEC 14443A	ISO/IEC 14443A	ISO/IEC 18092, JIS X 6319-4, FELICA	ISO/IEC 14443A, ISO/IEC 14443B	ISO/IEC 15693
Memoria	96 bytes a 2 Kbytes	48 bytes a 2 Kbytes	2 Kbytes	32 Kbytes	64 Kbytes

Ratio de transferencia	106 kbit/s	106 kbit/s	212 kbit/s, 424 kbit/s	106 kbit/s, 212 kbit/s, 424 kbit/s	26.48 kbit/s
Funcionalidad	Read Re-write Read-only	Read Re-write Read-only	Read Re-write Read-only	Read Re-write Read-only Factory-configured	Read Re-write Read-only
Anticolisiones	No	Sí	Sí	Sí	Sí
Notas	Simples, baratas	-	Mayor coste, aplicaciones complejas	-	Área cercana

Android tiene soporte para leer y manejar los primeros cuatro tipos de etiquetas especificados anteriormente, pero su servicio de emulación de etiquetas (HCE) solo permite la emulación de etiquetas de tipo 4.

Para el desarrollo de la aplicación Android se ha escogido implementar un emulador de etiquetas de tipo 4 basado en la especificación *NFC Forum Type 4 Tag Operation Specification 2.0* [13].

Una etiqueta de tipo 4 basada en esta especificación debe contener una aplicación de etiqueta NDEF. Esta aplicación es un sistema de archivos que contiene al menos estos dos archivos:

- **Capability Container (CC):** archivo de solo lectura que contiene información sobre la versión de la especificación implementada, los parámetros de comunicación de la etiqueta, e información sobre el resto de los archivos en la etiqueta de tipo 4.
- **NDEF File:** el archivo NDEF es el que contiene el mensaje NDEF. Este mensaje puede ser leído o reescrito dependiendo de las propiedades definidas en el archivo CC. A su vez, el archivo NDEF consta de dos campos:
    - **NLEN**: 2 bytes que especifican la longitud del mensaje NDEF en formato big-endian.
    - **Mensaje NDEF:** el propio mensaje NDEF. Su longitud está definida por el campo NLEN.

Una vez tenemos una etiqueta o emulador de etiquetas de tipo 4 totalmente operativo, se puede iniciar la comunicación entre la etiqueta y un dispositivo de lectura. Para poder captar las etiquetas de tipo 4, el lector debe estar configurado para buscar etiquetas a través del protocolo NFC-A (basado en ISO/IEC 14443A).

La comunicación entre la etiqueta y el lector se hace a través de “application protocol data units” (APDUs). El lector de etiquetas estará sondeando para buscar etiquetas NFC-A a su alrededor,

una vez encuentre una empezará una serie de mensajes para descubrir qué tipo de etiqueta es. El que nos interesa es el protocolo de comunicación entre el lector NFC-A y nuestra etiqueta NFC de tipo 4 con NDEF. En él, el lector mandará una serie de C-APDUs (Command APDUs), a los cuales la etiqueta deberá responder con R-APDUs (Response APDUs).

El flujo que sigue el protocolo de comunicación para leer la etiqueta es el siguiente:

- Lector manda C-APDU: **NDEF Tag Application Select**. El lector le indica a la etiqueta que está buscando una aplicación con mensaje NDEF. El comando está buscando por la ID **D2760000850101h**, que representa este tipo de etiquetas NFC con mensajes NDEF.
- Etiqueta manda R-APDU: el R-APDU confirma si la lectura es correcta o si la etiqueta rechaza la conexión.
- Lector manda C-APDU: **Capability Container (CC) Select**. El lector le pide "permiso" a la etiqueta para acceder a su fichero CC.
- Etiqueta manda R-APDU: el R-APDU confirma si la lectura es correcta o si la etiqueta rechaza la conexión.
- Lector manda C-APDU: **Capability Container Read**. El lector pide leer el archivo CC de la etiqueta.
- Etiqueta manda R-APDU: el R-APDU contiene el archivo CC.
- Lector manda C-APDU: **NDEF Select**. El lector le pide "permiso" a la etiqueta para acceder a su archivo NDEF.
- Etiqueta manda R-APDU: el R-APDU confirma si la lectura es correcta o si la etiqueta rechaza la conexión.
- Lector manda C-APDU: **NDEF Read**. El lector le pide a la etiqueta el archivo NDEF.
- Etiqueta manda R-APDU: el R-APDU contiene el campo **NLEN** del archivo NDEF, especificando la longitud del mensaje.

Una vez el lector conoce la longitud del mensaje NDEF (NLEN), continúa lanzando C-APDU pidiendo el archivo NDEF.

La etiqueta contesta con el mensaje NDEF en su R-APDU.

Una vez se ha mandado este último R-APDU, el mensaje NDEF ha llegado correctamente al lector de etiquetas y por lo tanto la comunicación ha terminado correctamente.

3.4. Sistema Android y API de NFC

Android es un sistema operativo para dispositivos móviles inteligentes. Está basado en el kernel de Linux y en la actualidad es desarrollado por la empresa Google.

El sistema operativo Android está pensado para ser utilizado en dispositivos móviles con una pantalla táctil, principalmente smartphones y tablets.

Android es el sistema operativo móvil más extendido del mundo, teniendo una cuota de mercado superior al 80% en el año 2017 [14], seguido por el sistema operativo de Apple, IOS.

Figura 8. Logo de Android.

El sistema operativo Android tiene una arquitectura en forma de pila, compuesta principalmente por cuatro capas, habiendo una pequeña subsección especial:

- **Kernel de linux:** es la capa de más bajo nivel. Proporciona una capa de abstracción entre los componentes hardware y las capas de más alto nivel. Cuenta con todos los drivers de los componentes hardware como la cámara, pantalla, interfaz de red, etc.

- **Librerías:** es una capa compuesta por librerías de código que pueden ser utilizadas por capas de mayor nivel. En esta categoría entrarían librerías como SSL, SQLite, y otras librerías de Java específicas para Android, como Android.os (llamadas al sistema operativo), Android.opengl (llamadas a la interfaz de OpenGL para renderizar gráficos), etc.

- **Android runtime:** se encuentra en la misma capa que las librerías. En esta capa se implementa la máquina virtual de **Dalvik**, que es una máquina virtual derivada de la JVM diseñada y optimizada especialmente para Android. En esta máquina virtual se ejecuta todo el código de las aplicaciones Android. Android runtime proporciona una serie de librerías base que permiten a los desarrolladores escribir código Java nativo y ejecutarlo sobre Android.

- **Framework de aplicaciones:** en esta capa se implementan muchos servicios de alto nivel, implementados como clases Java. Los desarrolladores pueden acceder a estos servicios en sus aplicaciones a través del uso de estas clases. Los principales servicios son:

- **Gestor de actividades:** controla el ciclo de vida de las aplicaciones y la pila de actividades.
- **Proveedores de contenido:** permite a las aplicaciones compartir datos con otras aplicaciones.
- **Gestor de recursos:** gestiona datos referentes a las interfaces de usuario como colores, layouts de interfaz, etc.
- **Gestor de notificaciones:** gestionan las notificaciones que le serán enviadas al usuario.
- **Sistema de vistas:** gestiona las "vistas", o interfaces de usuario.

- **Aplicaciones:** son las aplicaciones accesibles por el usuario desde la interfaz de Android. Son la capa más externa de Android.

Figura 9. Arquitectura del sistema Android.

3.4.1 Visión general de la API de Android

En este apartado se van a hablar de las principales características de la API de Android y su interacción entre ellas.

Las aplicaciones desarrolladas en Android cuentan principalmente con los siguientes componentes:

- **Manifiesto:** es un archivo XML que debe declarar todos los componentes, permisos y configuraciones especiales que tenga la aplicación.

- **Actividades**: las actividades son pantallas de la interfaz de usuario de una aplicación Android. Al lanzar una aplicación Android, se crea la actividad marcada como principal en el manifiesto. Son archivos de código Java o Kotlin que normalmente recogen el código principal de la pantalla. Se enlazan los elementos de la interfaz con elementos del código de la actividad.

  Las actividades tienen ciclos de vida definidos, que gestionan diferentes eventos. Estos eventos engloban desde su creación, hasta cuando la aplicación pasa a estar en primer plano, deja de estarlo, etc.

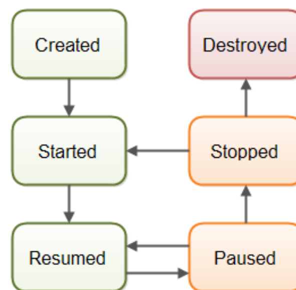


Figura 10. Ciclo de vida de una actividad Android.

- **Layouts:** se pueden crear como archivos XML, en el cual se recogen los elementos de la interfaz de usuario que tendrá una actividad. Los layouts también pueden ser modificados desde el código de las actividades.

- **Servicios:** los servicios son componentes de una aplicación que ejecutan código en segundo plano, sin necesidad de tener una interfaz de usuario asociada. Los servicios se pueden lanzar desde otros componentes, como una actividad.

- Al igual que las actividades, representan archivos de código escritos en Java o Kotlin, y también tienen un ciclo de vida definido.

- Los servicios pueden lanzarse manualmente a través de un Intent o enlazados con otro componente a través de *bindService*.

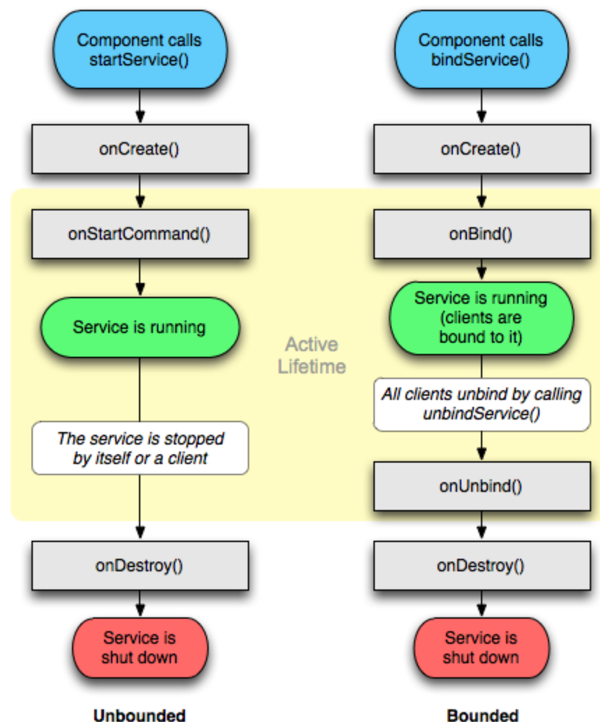


Figura 11. Ciclo de vida de un servicio Android.

- **Intents:** los intents son una forma de comunicación entre componentes. Los Intents pueden utilizarse para recoger datos desde el sistema operativo Android, como forma de iniciar actividades o servicios, enviar mensajes a otros componentes de una aplicación, etc. Los Intents tienen un campo llamado "extras" en los que se puede almacenar información. Esta información puede ser recuperada por el componente que reciba el Intent.

Una vez conocemos los componentes básicos de una aplicación Android, pasamos a estudiar las distintas formas de almacenar datos en el dispositivo Android.

El sistema operativo Android nos da las siguientes posibilidades a la hora de almacenar la información de las aplicaciones desarrolladas:

- **Almacenamiento específico de la aplicación:** hace referencia a la creación de archivos en directorios reservados para nuestra aplicación, ya sea en el almacenamiento interno del dispositivo o en tarjetas de expansión. En los directorios del almacenamiento interno se suelen guardar datos que no deben ser accedidos por otras aplicaciones.

- **Almacenamiento compartido:** se almacenan datos que puedan ser compartidos con otras aplicaciones, como datos multimedia, documentos, etc.

- **Preferencias:** es una forma de almacenamiento que guarda valores primitivos y privados a nuestra aplicación. Sigue un esquema basado en clave-valor.

- **Bases de datos:** Android permite la creación de bases de datos en nuestro sistema para almacenar datos masivos. Para ello, hace uso de la librería de persistencia Room. Esta librería permite almacenar datos creando una base de datos en código Java o Kotlin, o hacer uso del sistema de bases de datos SQLite.

Debido a que la aplicación desarrollada no tiene necesidad de almacenar datos de gran tamaño, necesitando simplemente almacenar información para los que se pueden usar tipos primitivos y que deben ser privados para nuestra aplicación, se ha decidido hacer uso de las preferencias para almacenar datos como el identificador del usuario y su nivel de privilegio en la aplicación.

3.4.2 API de Android para NFC

La API de Android para NFC [15] nos da una multitud de herramientas a la hora de manejar etiquetas NFC. De cara a la gestión de mensajes que siguen el formato NDEF [16], tenemos dos opciones principales de implementación:

- Android Beam.
- Implementación manual de lector y emulador de etiquetas.

3.4.2.1 Android Beam

Android Beam es una funcionalidad de Android introducida en la API 14 (Android 4.0) que permite compartir información entre dos teléfonos móviles con NFC activo a través de un protocolo propio.

Este modo de actuación del protocolo NFC sigue una comunicación de tipo *peer-to-peer*. Android Beam permite el envío de mensajes que siguen el formato NDEF entre teléfonos. Su implementación es bastante más sencilla que la del desarrollo de un emulador y lector de etiquetas, pero tiene un problema cuando se está desarrollando una aplicación para el control de asistencia basado puramente en NFC.

Debido a que todo el sistema de control de acceso depende de NFC, no se puede hacer uso de Android Beam ya que su implementación está limitada al uso de teléfonos móviles Android con NFC, es decir, si algún empleado no tuviese un dispositivo con estas características, quedaría excluido totalmente del sistema, ya que no se pueden utilizar etiquetas físicas como si se puede en el caso de un emulador y un lector de etiquetas NFC especializados.

Por esta razón se desestimó la implementación de un sistema con Android Beam.

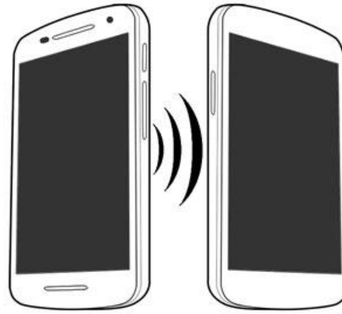


Figura 12. Esquema de funcionamiento de Android Beam.

3.4.2.2 Emulador y lector de etiquetas NFC

Esta es la segunda opción a la hora de manejar mensajes en formato NDEF, y la que finalmente fue escogida para el desarrollo de la aplicación.

En cuanto al emulador de etiquetas, el sistema operativo Android tiene un servicio conocido como *Host-based Card Emulation* (HCE). Este servicio fue implementado en la API 19 (Android 4.4) y les da la posibilidad a los desarrolladores de crear aplicaciones capaces de servir etiquetas NFC emuladas en el teléfono.

Las etiquetas que se pueden emular a través de este servicio son las etiquetas definidas por el foro de NFC [17] como etiquetas de tipo 4, basadas en ISO/IEC 14443-4. Hacen uso de los APDUs definidos anteriormente en la memoria como forma de intercambio de información.

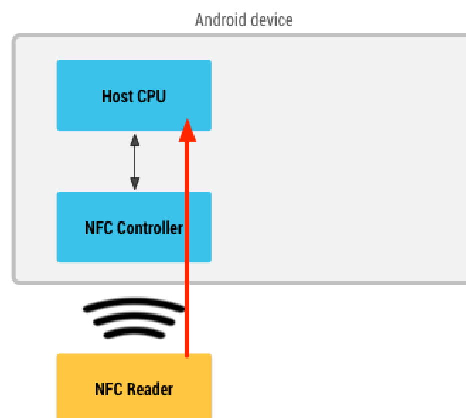


Figura 13. Emulación de etiquetas NFC en Android.

Este servicio tiene un método que el desarrollador debe implementar llamado "processCommandApdu". Los APDU serán enviados por el lector de etiquetas NFC y deberán ser procesados por el servicio de emulación contestando según la especificación definida en el apartado reservado para las etiquetas de tipo 4.

En segundo lugar, tenemos la API para implementar un lector de etiquetas NFC desde un dispositivo Android.

Normalmente, los teléfonos con NFC activo y la pantalla desbloqueada están constantemente sondeando en busca de etiquetas NFC cercanas con las que interactuar, cuando se encuentra una se filtra a través de un sistema conocido como *Tag Dispatch System*.

Cuando este sistema descubre una etiqueta NFC, lee sus contenidos y, dependiendo este, decide que aplicaciones están preparadas para leer la etiqueta NFC, iniciando dicha aplicación en el caso de que solo haya una, o mostrando un menú para que el usuario escoja entre aplicaciones posibles en caso de que haya más de una capaz de leer la etiqueta encontrada.

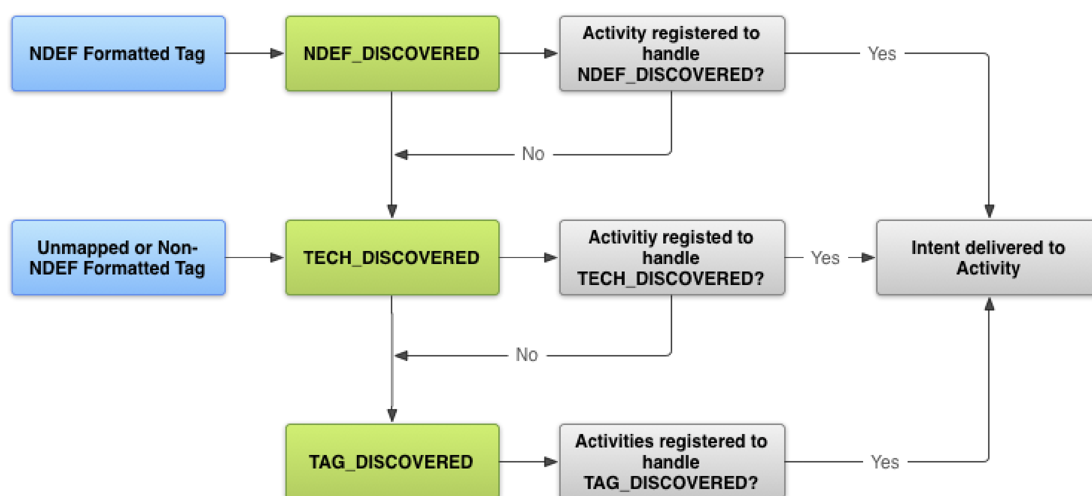


Figura 14. Funcionamiento de Tag Dispatch System.

Si nuestra aplicación está registrada para poder leer un tipo de etiquetas en el manifiesto, al encontrar una etiqueta de ese tipo, el sistema Android lanzará nuestra aplicación. Desde la aplicación, simplemente tendremos que hacer uso de las herramientas de Android para descomponer la etiqueta y recuperar sus datos.

Hay dos formas de obligar al sistema a redirigir las etiquetas a nuestra aplicación en todos los casos, estas son:

- **ForegroundDispatchSystem:** este sistema recoge todas las etiquetas NFC que lea el teléfono mientras la actividad de nuestra aplicación esté en primer plano y las intenta redirigir a nuestra aplicación antes de a cualquier otra aplicación, dándole prioridad a nuestra aplicación.

- **ReaderMode:** sobrecarga el modo de lectura de etiquetas para que cuando sean leídas sean enviadas a nuestra aplicación. Su funcionamiento en este sentido es bastante parecido al anterior, pero nos permite configurar el lector con mayor profundidad, haciendo que nuestro lector solamente acepte tarjetas NFC que sigan el protocolo NFC-A, el protocolo NFC-B, que no tengan mensajes NDEF en su interior, etc. Además, con el uso de esta funcionalidad, mientras esté activa, el chip NFC solo puede leer etiquetas, es decir, no puede servir etiquetas ni hacer uso de ninguna otra característica de NFC.

- Para la implementación del lector de la aplicación desarrollada se hizo uso de ReaderMode debido a la capacidad de obligar al chip NFC a estar en modo lectura. La explicación en detalle sobre esta decisión está recogida en el apartado de desarrollo del sistema de la memoria.

4. Desarrollo del sistema

En este apartado se trata la arquitectura del sistema desarrollado y se habla en detalle de las distintas partes componentes del sistema software, haciendo hincapié en las partes más importantes y complejas de cada una de estas partes.

4.1. Arquitectura del sistema

El sistema desarrollado cuenta con los siguientes subsistemas:

- **Servidor HTTP:** el servidor HTTP implementa una API ReST encargada de unir el resto de los subsistemas del proyecto con la base de datos.

- **Base de datos:** alojada en el mismo servidor físico que el servidor HTTP. En ella se guarda toda la información necesaria para poder gestionar la asistencia y registro horario de todos los empleados de la empresa. La base de datos es accedida a través de la API ReST implementada en el servidor HTTP.

- **Aplicación Android:** desde ella se realiza la acción de "fichar" en la aplicación, compartiendo mensajes NDEF entre dos dispositivos Android y mandando la información al servidor HTTP.

- **Aplicación de escritorio para administradores:** desde esta aplicación los usuarios con privilegios de administrador serán capaces de gestionar y acceder a todos los datos referentes a información sobre empleados, horarios, y registros de asistencia guardados en la base de datos a través del servidor HTTP.

A continuación, se presenta un diagrama que resume las interacciones entre las partes del sistema:

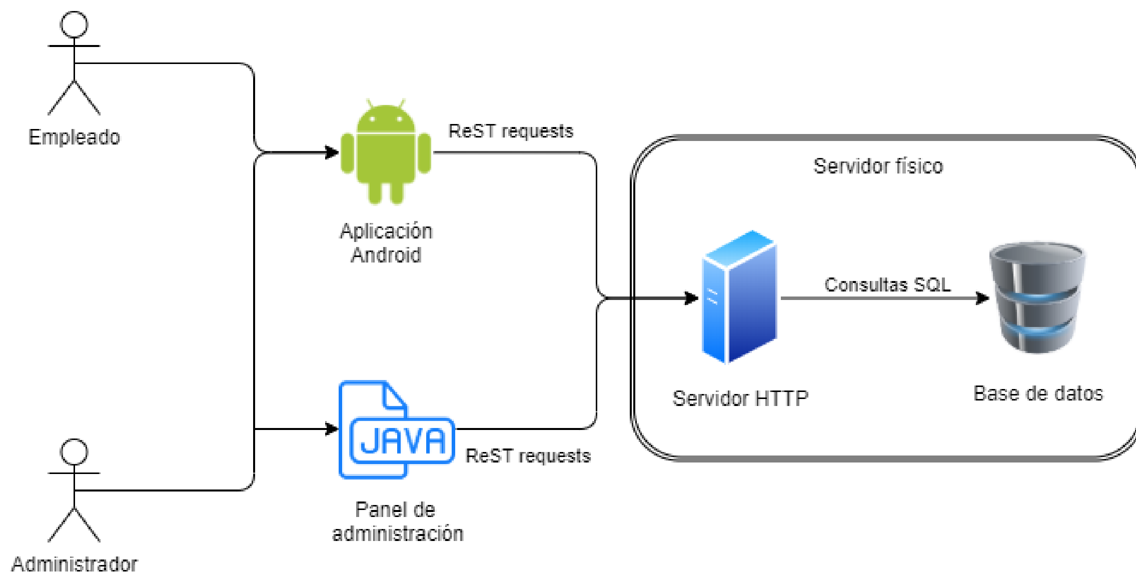


Figura 15. Visión general del sistema.

Para el correcto desarrollo del sistema propuesto se necesitan los siguientes componentes:

- Servidor con Python3 instalado con los módulos necesarios para el funcionamiento del servidor (Módulo de Flask y soporte de Swagger 2.0).

- Base de datos SQLite3 instalado en el servidor HTTP.

- Ordenador con Java instalado para el uso de la aplicación de administración.

- Al menos dos dispositivos Android con NFC (Para un usuario normal y un administrador). La API de Android mínima requerida para el funcionamiento de la aplicación es la API 23 (Android 6.0).

4.2. Base de datos

Para poder gestionar los datos referentes al control de asistencia de una organización, se ha creado el siguiente modelo de base de datos basado en tres tablas:

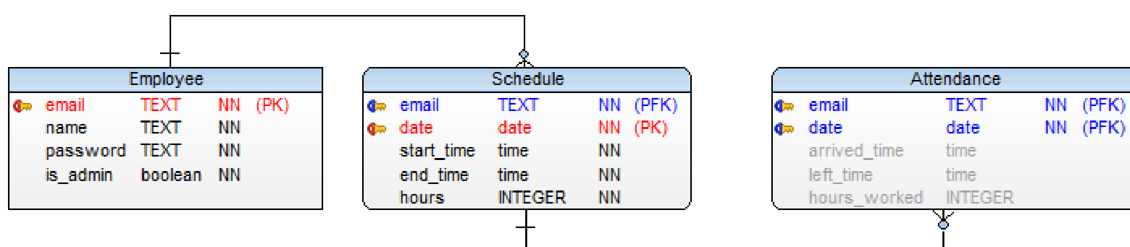


Figura 16. Modelo de datos.

Las relaciones entre las distintas tablas son las siguientes:

- **Employee-Schedule:** cada empleado puede tener múltiples registros de horarios asociados a su email.

- **Schedule-Attendance:** cada registro horario debe tener un registro de asistencia asociado siempre. Esta relación es 1:1, es decir, un registro horario debe tener una contraparte única en la tabla de asistencia y viceversa. Para realizar esta identificación bilateral, la tabla Schedule comparte sus claves primarias de email (email del empleado asociado al registro horario) y fecha (fecha del registro horario).

Las tablas representadas en la figura 16 tienen la siguiente estructura:

- **Tabla Employee:** esta tabla recoge toda la información referente a los datos de los empleados. Sus columnas son las siguientes:

    - **Email:** Tipo text, primary key de la tabla. Almacena el email del empleado, el cual será utilizado a la hora de hacer login en las aplicaciones.

- **Name:** Tipo text. Almacena el nombre del empleado.
- **Password:** Tipo text. Almacena la contraseña del empleado.
- **Is_admin:** Tipo boolean. Si el valor está a True (1), el empleado es un administrador. Si está a False (0), el empleado es un empleado normal sin privilegios.

- **Tabla Schedule:** esta tabla almacena toda la información sobre los horarios de los empleados. Sus columnas son las siguientes:
  - **Email:** Tipo text, primera parte de la primary key y foreign key de la tabla Employee. Almacena el email del empleado sobre el que se ha creado el registro de horario.
  - **Date:** Tipo date, segunda parte de la primary key. Almacena la fecha (formato yyyy-mm-dd) del día al que se refiere el horario.
  - **Start_time:** Tipo time. Representa la hora de inicio de la jornada de trabajo del empleado para el registro horario.
  - **End_time:** Tipo time. Representa la hora de finalización de la jornada de trabajo del empleado para el registro horario.
  - **Hours:** Tipo integer. Almacena en un número entero el número de horas de la jornada de trabajo para este registro horario. Al crear el registro horario, se calcula el número de horas que debe trabajar ese día el empleado.
- **Tabla Attendance:** esta tabla almacena la información sobre los registros de asistencia de los empleados. Es un reflejo de la tabla Schedule. Cada vez que se crea un registro en la tabla horario, se crea un registro en la tabla Attendance que refleja la asistencia de ese día.
  - **Email:** Tipo text, primera parte de la primary key y foreign key de la tabla Schedule. Almacena el email del empleado sobre el que se ha creado el registro de asistencia.
  - **Date:** Tipo date, segunda parte de la primary key y foreign key de la tabla Schedule. Almacena la fecha del registro de asistencia.
  - **Arrived_time:** Tipo time. Almacena la hora de llegada al trabajo por el empleado. El timestamp se genera cuando ficha el trabajador por primera vez en un día, es decir, cuando llega al trabajo.

- **Left_time:** Tipo time. Almacena la hora de salida del trabajo por el empleado. El timestamp se genera cuando ficha el trabajador por segunda vez en un día, es decir, cuando sale del trabajo.

- **Hours_worked:** Tipo integer. Cuando el trabajador fiche por segunda vez, se calcula la diferencia de horas entre su hora de llegada y su hora de salida. Debido a que es un número entero, se dan 10 minutos de margen de maniobra. Esto es, si el trabajador trabaja 3 horas y 50 minutos, se cuentan como 4 horas de trabajo.

La base de datos está pensada para que cada vez que se cree un registro de horario, se cree a su vez un registro de asistencia asociado a ese horario. Para ello, los campos de "Arrived_time", "Left_time" y "Hours_worked" de la tabla "Attendance" se inicializan a null para ser actualizados más adelante. Asimismo, cuando se elimina un registro horario de un empleado, se borran también sus registros de asistencia para ese horario.

Debido a la simplicidad del modelo de datos de la aplicación, se ha decidido hacer uso de **SQLite 3**, ya que permite la creación de bases de datos ligeras y muy fáciles de implementar y mantener cuando la base de datos tiene estas características. Ya que SQLite no cuenta con un gestor de conexiones remotas, el servidor HTTP debe estar en la misma máquina física que la base de datos.

Figura 17. Logo de SQLite.

SQLite no cuenta con algunos de los tipos de datos especificados anteriormente. Por ejemplo, el tipo boolean se almacena como un integer con valores [0, 1], los tipos date se almacenan como text, etc. A efectos prácticos que no existan estos tipos no es relevante ya que se pueden hacer gestiones externas bastante sencillas para suplir esta falta de tipos.

Toda la lógica encargada de comprobar cada uno de los estados de la base de datos está implementada en el servidor HTTP.

Con la finalidad de que se pueda probar el funcionamiento de la aplicación, se presentan dos scripts SQL encargados de crear las tablas de la base de datos y poblar las tablas con datos de prueba respectivamente. Se presenta también un script en formato .bat para automatizar la creación de la base de datos y la llamada a estos scripts SQL.

4.3. Servidor HTTP

Para proporcionar una interfaz de acceso entre la aplicación de Android y el panel de administración con la base de datos se ha decidido crear un servidor HTTP que ofrezca una API ReST encargada de hacer todas las consultas a la base de datos.

La decisión radica en que Android no soporta el driver JDBC (Java Database Connectivity) para conectarse directamente a bases de datos. Hay una implementación de JDBC utilizable desde Android llamada JDTS, pero solo tiene soporte para bases de datos basadas en Microsoft SQL Server. Debido a que no se tiene experiencia con esta base de datos y que, el uso de un gestor de base de datos complejo para un modelo de datos tan pequeño parecía demasiado, se decidió implementar un servidor HTTP que hiciese de interfaz entre el resto de la aplicación y la base de datos basada en SQLite.

En un primer momento se planteó la creación de una API ReST haciendo uso de tecnologías en Java Enterprise Edition, pero debido a la simplicidad que ofrece Python y sus framework en la creación de servidores HTTP, se acabó por descartar esta idea.

El servidor HTTP está implementando en **Python 3** haciendo uso del framework **Flask** [18]. Python junto con Flask nos permite desplegar servidores HTTP funcionales de una forma muy rápida y sencilla. Otro motivo que influyó en la decisión de escoger Python fue la gran interoperabilidad que hay entre SQLite y Python, pudiendo simplificar lo máximo posible las conexiones entre el servidor HTTP y la base de datos.

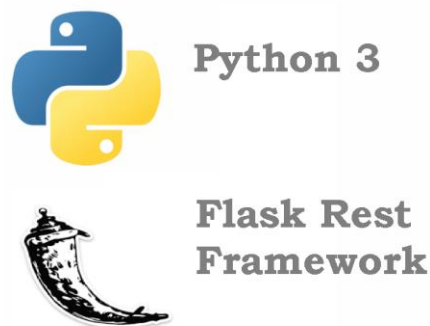


Figura 18. Logos de Python3 y el framework Flask.

De cara a implementar la API ReST, se ha decidido seguir la especificación de la **OpenAPI 2.0** [19], conocida como **Swagger**. Swagger permite definir ficheros de despliegue en formato YAML o JSON en el cual se definen las rutas que soportará nuestra API, los métodos HTTP que soportará cada ruta, y el enlace al método que gestionará cada una de las llamadas a nuestra API de forma sencilla.

La estructura de ficheros del servidor HTTP es la siguiente:

- **Employee.py:** fichero Python que recoge todo el código que responde a las llamadas ReST relacionadas con la tabla Employee de la base de datos.

- **Schedule.py:** fichero Python que recoge todo el código correspondiente a las llamadas ReST relacionadas con las tablas Schedule y Attendance de la base de datos.

- **time_library.py:** fichero Python que tiene una librería con algunas funciones que gestionan timestamps, cálculos de diferencias entre horas, etc.

- **Server_TFG.py:** fichero Python con el "main" del servidor. Simplemente instancia la API leyendo el fichero de despliegue y crea el servidor en el puerto 8080.

- **swagger.yml:** fichero de despliegue de la aplicación web.

Haciendo uso de Swagger, los métodos de gestión del servidor HTTP están gestionados internamente y no tenemos que implementarlos manualmente. Simplemente tenemos que enlazar nuestra aplicación web con la API haciendo la siguiente llamada en el código:

```
app = connexion.App(__name__, specification_dir='./')
app.add_api('swagger.yml')
```

Donde "swagger.yml" es el fichero de despliegue definido.

Para explicar el funcionamiento del fichero de despliegue, se muestra el siguiente ejemplo:

```
  /login:
    post:
      operationId: Employee.login
      tags:
        - Employee
      summary: Intenta hacer login en el servidor
      description: Intenta hacer login en el servidor con las credenciales
      parameters:
        - name: employee
          in: body
          description: Empleado intentando hacer login
          required: True
          schema:
            type: object
            properties:
              username:
                type: string
                description: Nombre de usuario del empleado (email)
              password:
                type: string
                description: Contraseña del empleado
      responses:
        200:
          description: Login correcto
```

Este ejemplo especifica la respuesta del servidor HTTP cuando se acceda a la ruta "http://[…]/api/login". Si se accede a esta ruta con el método HTTP "POST", se llamará al método de Python del archivo "Employee" llamado "login()". Como indica el descriptor de despliegue,

este método necesita un parámetro de tipo object (JSON). El JSON deberá tener 2 campos llamados "username" y "password".

A la vez de especificar todos los requerimientos técnicos de la API, también sirve de documentación para los desarrolladores, ya que aporta información sobre cada uno de los métodos de la API (Descripciones, posibles respuestas de cada método, tipos de datos de los parámetros, etc.).

Para entender mejor el funcionamiento del fichero de despliegue, veamos ahora la contraparte en Python a la que está llamando el ejemplo anterior.

```
def login(employee):
    username = employee.get("username", None)
    password = employee.get("password", None)

    con = sqlite3.connect("./Database/DB.db")
    con.row_factory = sqlite3.Row
    cursor = con.cursor()

    cursor.execute(f"SELECT * FROM Employee WHERE email=\'{username}\' AND pa
ssword=\'{password}\'")

    result = cursor.fetchall()
    cursor.close()
    con.close()

    if result:
        resultJson = json.dumps([dict(i) for i in result], indent=1)
        return make_response(resultJson, 200)
    else:
        abort(401, "Unsuccessful login")
```

Cuando el usuario accede a "http://[…]/api/login" con método POST y el JSON especificado en el cuerpo de la petición HTTP, se llama al método anterior, pasándole como parámetro el objeto JSON.

Se recuperan los datos del JSON accediendo al parámetro, se crea la conexión con la base de datos y se ejecuta la consulta recuperando todos los datos del empleado si el email y la contraseña coinciden.

Se cierra la conexión con la base de datos y, en caso de que haya resultado, se devuelve un objeto JSON con los datos del empleado recuperados de la base de datos, con código de respuesta 200 como especificaba el fichero de despliegue. En caso de que no haya resultado, se devuelve el código de error 401.

Todos los métodos de los ficheros de Python siguen la misma estructura (conexión con la base de datos, consulta SQL, crear resultado, responder al cliente).

El método más complejo del servidor HTTP, y también el más relevante para el proyecto, es el encargado de actualizar los datos de asistencia de un empleado cuando el usuario ficha con el teléfono.

La parte del fichero YAML encargada de gestionar esta llamada a la API es la siguiente:

```
put:
      operationId: Schedule.create_attendance_record
      tags:
        - Attendance
      summary: Actualiza la información de asistencia para un empleado en la
base de datos
      description: Actualiza la información de asistencia para un empleado en
 la base de datos
      parameters:
        - name: email
          in: path
          description: Email del empleado
          type: string
          required: True
      responses:
        200:
          description: Datos actualizados correctamente
        400:
          description: Error al intentar actualizar los datos de asistencia
```

Se debe hacer una llamada a la ruta /attendance/{email} con método put para acceder a esta función. La contraparte en Python va a ser explicada por partes para simplificar la función.

En un primer momento, se prepara una serie de variables y se abre la conexión con la base de datos:

```
timestamp = time_library.get_timestamp().split(" ")
response_code = 400
response_str = ""

con = sqlite3.connect("./Database/DB.db")
con.row_factory = sqlite3.Row
cursor = con.cursor()
```

Seguidamente, se recupera la información de asistencia para el día de hoy en base al timestamp creado:

```

cursor.execute(f"SELECT * FROM Attendance WHERE email=\'{email}\' AND date=\'
{timestamp[1]}\';")
result = cursor.fetchone()
```

Una vez hemos recuperado esta información, valoramos el resultado.

En caso de que no haya ningún registro de asistencia para esa combinación de usuario/fecha, se deja el código de respuesta a 400 y se actualiza el string de respuesta de la siguiente manera:

```
else:
        response_str = "no data to update"
```

En caso de que sí haya un registro de asistencia para esa combinación de usuario/fecha, se comprueba a ver si en el resultado el campo de "arrived_time" es null. Si así fuese, se lanza la consulta actualizando este campo en la base de datos con el timestamp actual, creando el registro de asistencia que refleja la hora de entrada al trabajo. Se actualizan el string de respuesta y el código de respuesta:

```
if result['arrived_time'] is None:
   cursor.execute(f"UPDATE Attendance SET arrived_time=\'{timestamp[0]}\' WHE
RE email=\'{email}\' AND date=\'{timestamp[1]}\';")
   response_str = "arrived_time updated"
   response_code = 200
```

En caso de que "arrived_time" no sea null, es decir, de que ya se haya fichado 1 vez para este registro de asistencia, se comprueba que "left_time" no sea nulo y que haya una diferencia válida de tiempo entre el timestamp de entrada y el de ahora. Una diferencia válida se considera que haya pasado más de un minuto. Esto se hace para tener medidas de protección adicionales de cara a evitar que se pueda fichar dos veces sin querer. En el caso de que se cumpla que "left_time" sea nulo, se actualiza este campo en la base de datos, esencialmente fichando por segunda vez y marcando el tiempo de salida del trabajo. Se actualizan el string de respuesta y el código de respuesta:

```
elif result['left_time'] is None:
 if time_library.valid_time_difference(result['arrived_time'], timestamp[0]):
     hours_worked = time_library.hour_difference(result['arrived_time'], time
stamp[0])
     cursor.execute(f"UPDATE Attendance SET left_time=\'{timestamp[0]}\', hou
rs_worked={hours_worked} WHERE email=\'{email}\' AND date=\'{timestamp[1]}\';
")
      response_str = "left_time updated"
      response_code = 200
 else:
      response_str = "update too soon"
```

El último caso posible es que sí haya un registro de asistencia para el día de hoy pero que tanto "arrived_time" como "left_time" ya estén actualizados y tengan un valor en la base de datos. En ese caso, el empleado ya ha fichado 2 veces el mismo día, por lo que este nuevo intento de fichar no es correcto. Se actualiza el string de respuesta y se deja el código de respuesta a 400.

```
else:
    response_str = "table already updated"
```

Por último, cerramos la conexión con la base de datos y se gestiona la respuesta del servidor:

```
    if response_code == 200:
        return make_response(response_str, 200)
    else:
        return make_response(response_str, 400)
```

El último punto a remarcar del servidor HTTP es una funcionalidad ofrecida por Swagger que consiste en una interfaz gráfica desde la cual se nos sirven todas las llamadas a la API con toda la documentación que habíamos especificado en el fichero YAML. Esta interfaz gráfica ha sido la principal herramienta utilizada a la hora de hacer pruebas sobre el funcionamiento de la base de datos y el servidor HTTP.

Para acceder a ella simplemente tenemos que abrir un navegador, acceder a la IP y puerto del servidor y dirigirnos a "/api/ui/":

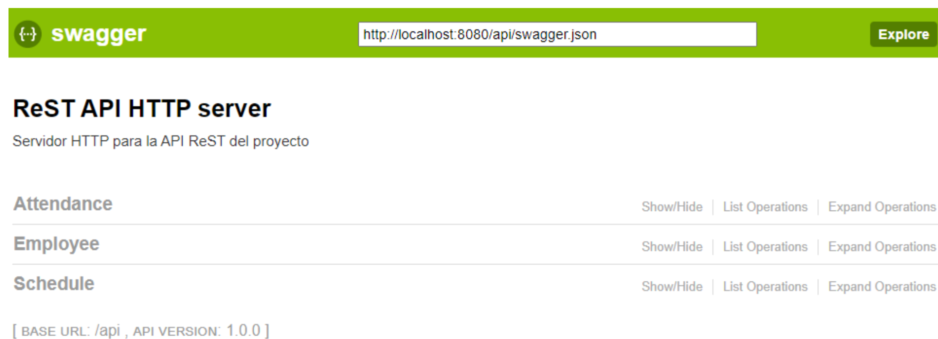


Figura 19. Visión general de la GUI de Swagger.

Si accedemos a Attendance, se nos muestra todas las llamadas ReST registradas bajo la etiqueta Attendance:

Figura 20. Ejemplo de llamadas a la API con la GUI de Swagger.

Si hacemos click en cualquiera de ellas, se nos muestra toda su información y se nos permite hacer una llamada y comprobar el resultado de esta:

GET /attendance/{email} — Lee la información de asistencia de un empleado

Implementation Notes

Lee la información de asistencia de un empleado

Response Class (Status 200)

Datos de asistencia recuperados correctamente

Model | Example Value

```
[
  {
    "arrived_time": "string",
    "date": "string",
    "email": "string",
    "left_time": "string"
  }
]
```

Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
email	edu@tfg.net	Email del empleado	path	string

Try it out! Hide Response

Figura 21. Ejemplo de llamada con la GUI de Swagger.

Curl

```
curl -X GET --header 'Accept: text/html' 'http://localhost:8080/api/attendance/edu%40tfg.net'
```

Request URL

```
http://localhost:8080/api/attendance/edu%40tfg.net
```

Response Body

```
[
 {
  "email": "edu@tfg.net",
  "date": "2020-08-01",
  "arrived_time": "08:59:30",
  "left_time": "17:00:30"
 },
 {
  "email": "edu@tfg.net",
  "date": "2020-08-02",
  "arrived_time": "09:30:00",
  "left_time": "17:00:00"
 },
 {
  "email": "edu@tfg.net",
  "date": "2020-08-03",
  "arrived_time": null,
  "left_time": null
 },
 {
```

Response Code

```
200
```

Response Headers

```
{
  "content-length": "555",
  "content-type": "text/html; charset=utf-8",
  "date": "Tue, 18 Aug 2020 15:06:06 GMT",
  "server": "Werkzeug/0.16.0 Python/3.7.3"
}
```

Figura 22. Ejemplo de respuesta con la GUI de Swagger.

Aprovechando esta interfaz gráfica, se puede enseñar un resumen de las llamadas a la API implementadas:

Attendance — Show/Hide | List Operations | Expand Operations

Método	Ruta	Descripción
GET	/analysis/{email}/{month}/{year}	Devuelve la información de análisis sobre la asistencia de un empleado en un determinado mes
GET	/attendance/{email}	Lee la información de asistencia de un empleado
PUT	/attendance/{email}	Actualiza la información de asistencia para un empleado en la base de datos
GET	/attendance/{email}/{month}/{year}	Lee la información de asistencia para un empleado en una fecha concreta

Employee — Show/Hide | List Operations | Expand Operations

Método	Ruta	Descripción
GET	/employee	Lee la lista de empleados completa
POST	/employee	Crear un nuevo empleado
DELETE	/employee/{email}	Borrar un empleado de la base de datos
GET	/getInfo/{name}	Lee los datos de los empleados a partir del nombre especificado
POST	/login	Intenta hacer login en el servidor

Schedule — Show/Hide | List Operations | Expand Operations

Método	Ruta	Descripción
POST	/schedule	Crea un nuevo horario para un empleado
DELETE	/schedule/{email}/{date}	Borra un registro de horario de la base de datos
GET	/schedule/{email}/{date}	Lee la información de horario para un empleado en una fecha especificada
GET	/schedule/{email}/{month}/{year}	Lee la información de horario para un empleado en concreto

Figura 23. Vista general de la API ReST Implementada.

4.4. Aplicación Android

La aplicación de Android es el eje central del proyecto y la parte del sistema que cuenta con una mayor complejidad. La principal funcionalidad desde la que se parte el proceso de diseño es la necesidad de tener una aplicación capaz de emular tarjetas NFC con información de los empleados que quieren fichar al llegar y salir del trabajo, y que esta misma aplicación en manos de un administrador pueda ser usada como lector de tarjetas NFC para recoger las tarjetas emuladas por los empleados, creando registros de control de asistencia y horarios de esta forma.

La solución adoptada cuenta con las siguientes clases Java:

- **AlertDialogFactory:** es una clase que extrae la funcionalidad de crear AlertDialogs basados en el layout “progress_alert_dialog”, el cual cuenta con una barra de carga.

- **CheckScheduleActivity:** actividad de Android accesible por empleados no administradores. Permite a los empleados obtener información sobre sus horarios.

- **RegisterEmployeeActivity:** actividad de Android accesible por empleados administradores. Les da la posibilidad a los administradores de dar de alta a nuevos empleados en la base de datos desde el teléfono móvil.

- **LoginActivity:** actividad de Android que cuenta con la función de login en la aplicación. Para poder acceder al resto de actividades y servicios de la aplicación es necesario que el empleado haya hecho login previamente, estando identificado en todo momento.

- **MainActivity:** actividad de Android que hace de menú principal de la aplicación. Dependiendo de si el usuario es un administrador o un empleado sin privilegios, permite acceder a unas funcionalidades u otras de la aplicación.

- **ReadNFCActivity:** actividad de Android accesible por empleados administradores. Recoge la funcionalidad de leer tarjetas NFC y recuperar mensajes NDEF para mostrárselos al administrador por pantalla y mandárselos a la base de datos para actualizar los registros de asistencia.

- **EmulateNFCTagActivity:** actividad de Android accesible por empleados no administradores. Esta actividad prepara la emulación de una tarjeta NFC, notificando al empleado de la información que se va a emular. Cuando el empleado pulsa el botón de emulación, se lanza el servicio encargado de emular la tarjeta.

- **EmulateNFCTagService:** servicio de Android que hereda del servicio HostApduService, dando la posibilidad de acceder al servicio HCE (Host Card Emulation) de Android. El servicio se encarga de hacer el intercambio de mensajes entre la tarjeta emulada y el lector, sirviendo la información del usuario a través de un mensaje NDEF.

Una vez conocemos la estructura básica de la aplicación, pasamos a explicar en detalle las funcionalidades más importantes.

4.4.1. Login, información de usuario y conexión con el servidor HTTP

Para el funcionamiento de la aplicación, es necesario que el usuario esté identificado en todo momento, y conocer si el usuario tiene permisos de administrador o no para cambiar la interfaz del menú principal, cambiando las funcionalidades a las que puede acceder.

El almacenamiento de esta información se hace a través de la interfaz de Android introducida en la API 1 llamada "SharedPreferences". Esta interfaz nos permite almacenar variables en la memoria interna de la aplicación, permitiendo el acceso desde cualquier punto, denegando al mismo tiempo el acceso a otras aplicaciones que quieran recuperar esta información.

Al lanzar la aplicación se abre la actividad principal (MainActivity). Dentro de esta información se recupera la información de las SharedPreferences. En caso de que el usuario no esté autenticado en la aplicación, se redirige al usuario a la actividad de login. El siguiente fragmento del código es el encargado de implementar esta funcionalidad.

```java
private void checkLoginStatus() {
    sharedPreferences = getSharedPreferences( name: "LOGIN_INFO", Context.MODE_PRIVATE);
    if(sharedPreferences.getString( key: "USER",  defValue: "").equals("")) {
        startActivity(new Intent( packageContext: this, LoginActivity.class));
        finish();
    }
    else {
        Log.i( tag: "MainActivity",
                 msg: (sharedPreferences.getBoolean( key: "ADMIN",  defValue: false) ? "ADMIN " : "EMPLOYEE ")
                        + sharedPreferences.getString( key: "USER",  defValue: ""));

        Toast.makeText( context: this,  text: "Bienvenido " + sharedPreferences.getString( key: "USER",  defValue: "null"), Toast.LENGTH_LONG).show();
    }
}
```

Figura 24. Función checkLoginStatus.

Con esto conseguimos que el usuario solo se tenga que autenticar una vez en la aplicación. Aunque la aplicación se cierre totalmente (borrándose de memoria), la aplicación recordará estas SharedPreferences, permitiendo que el usuario solo tenga que autenticarse una vez. El usuario es capaz de cerrar sesión desde el menú principal, borrando estas SharedPreferences y siendo redirigido a la actividad de login.

En la actividad de login, el empleado deberá proporcionar un usuario (email) y contraseña. Al pulsar el botón de login, se lanzan los datos contra el servidor HTTP y se analiza la respuesta del servidor. En caso de que sea correcta, se recuperan los datos del JSON y se crean las SharedPreferences con los datos relevantes, el email y los privilegios del empleado (si tiene permisos de administración o no):

```java
switch (responseCode) {
    case 200:
        try {
            JSONObject obj = new JSONArray(jsonResponse).getJSONObject( index: 0);
            sharedPreferences.edit().putString("USER", obj.getString( name: "email")).apply();
            sharedPreferences.edit().putBoolean("ADMIN", obj.getInt( name: "is_admin") == 1).apply();
            startActivity(new Intent( packageContext: this, MainActivity.class).addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP));
            finish();
        } catch (JSONException e) {
            Log.e( tag: "JSONException", Objects.requireNonNull(e.getMessage()));
        }
        break;
```

Figura 25. Establecimiento de las SharedPreferences.

Por último, en cuanto al aspecto de las conexiones con el servidor HTTP remoto. Para poder realizar conexiones remotas desde una actividad se deben declarar los permisos de Android relacionados con Internet en el manifiesto:

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

Figura 26. Permisos necesarios para el uso de conexiones de red.

Además, Android no permite hacer llamadas bloqueantes a servidores remotos desde el hilo principal, ya que este hilo es el encargado de gestionar la interfaz gráfica. Para poder conectarse

con el servidor HTTP se tienen que crear clases que extiendan a la clase “AsyncTask”. La clase AsyncTask permite ejecutar tareas de fondo, devolviéndole la información al hilo principal una vez haya terminado de ejecutar la tarea.

Todas las actividades que necesitan conectarse al servidor HTTP tienen una clase interna privada que extiende a AsyncTask con el objetivo de gestionar estas llamadas. La clase que extiende a AsyncTask recibe todos los datos necesarios de la interfaz a través de su constructor y después se lanza el hilo. Por ejemplo, en el caso del login:

```
AsyncGetLoginInfoTask task = new AsyncGetLoginInfoTask(user, password);
task.execute();
```

Figura 27. Lanzamiento de una AsyncTask.

Al llamar a ejecutar una AsyncTask, internamente se llama al método “doInBackGround()”, que será el método que implementa toda la lógica de la comunicación con el servidor HTTP. En este método se prepara la conexión con el servidor HTTP configurando los parámetros de la conexión y se lanza la petición HTTP. Después, se recupera la respuesta del servidor y si se debe recuperar información de la respuesta, se guarda en un String. En el caso del login:

```java
@Override
protected Void doInBackground(Void... voids) {
    try {
        // Configurar la llamada a la API ReST del servidor HTTP
        HttpURLConnection connection = (HttpURLConnection) new URL( spec: "http://192.168.1.136:8080/api/login").openConnection();
        connection.setConnectTimeout(5000);
        connection.setRequestMethod("POST");
        connection.setRequestProperty("Content-Type", "application/json");
        connection.setRequestProperty("Accept", "application/json");
        connection.setDoOutput(true);

        // Construir el JSON y escribirlo en la petición
        String jsonRequest = "{\"username\": \"" + user + "\", \"password\": \"" + password + "\"}";
        OutputStream os = connection.getOutputStream();
        byte[] input = jsonRequest.getBytes(StandardCharsets.UTF_8);
        os.write(input,  off: 0, input.length);

        responseCode = connection.getResponseCode();
        if (responseCode == 200) {
            response = "";
            Scanner scanner = new Scanner(connection.getInputStream());

            while (scanner.hasNextLine()) {
                response += scanner.nextLine() + "\n";
            }
            scanner.close();
            Log.i( tag: "AsyncGetLoginInfo",  msg: "Succesful login. Code 200 in response.");
        }
        connection.disconnect();
    }
    catch (java.net.SocketTimeoutException e) {
        Log.e( tag: "AsyncGetLoginInfo",  msg: "Connection timeout");
    }
    catch (IOException e) {
        Log.e( tag: "AsyncGetLoginInfo", Objects.requireNonNull(e.getMessage()));
    }

    return null;
}
```

Figura 28. Código de ejemplo de una petición HTTP en una AsyncTask.

Una vez esta función haya terminado se llama al método “onPostExecute()”. Este método simplemente se encarga de llamar a un método de la clase padre para dejarle saber al usuario el resultado de la tarea del hilo, es decir, si la conexión con el servidor ha sido satisfactoria. En caso de que el usuario esté esperando una respuesta, se le notificará de alguna forma a través de la interfaz. En el caso del login:

```java
@Override
protected void onPostExecute(Void result) {
    updateUserInfo(responseCode, response);
}
```

Figura 29. Ejemplo de función onPostExecute.

Y la función “updateUserInfo”:

```java
protected void updateUserInfo(int responseCode, String jsonResponse) {
    dialog.dismiss();

    switch (responseCode) {
        case 200:
            try {
                JSONObject obj = new JSONArray(jsonResponse).getJSONObject( index: 0);
                sharedPreferences.edit().putString("USER", obj.getString( name: "email")).apply();
                sharedPreferences.edit().putBoolean("ADMIN", obj.getInt( name: "is_admin") == 1).apply();
                startActivity(new Intent( packageContext: this, MainActivity.class).addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP));
                finish();
            } catch (JSONException e) {
                Log.e( tag: "JSONException", Objects.requireNonNull(e.getMessage()));
            }
            break;

        case 401:
            passwordTextField.setText("");
            Toast.makeText( context: this,  text: "Datos incorrectos", Toast.LENGTH_LONG).show();
            break;

        default:
            passwordTextField.setText("");
            Toast.makeText( context: this,  text: "No se ha podido conectar con el servidor", Toast.LENGTH_LONG).show();
    }
}
```

Figura 30. Ejemplo de actualización de la interfaz de usuario con la información recogida de la petición HTTP.

En este caso, se analiza el código de respuesta del servidor. En caso de que sea 200 (conexión correcta), se almacena la información del usuario en las SharedPreferences y se lanza la actividad principal a través de un Intent. En caso de que la respuesta sea 401, hay un error con el login y se le hace saber a través de un mensaje Toast. En cualquier otro caso, ha habido problemas de conexión con el servidor.

Esta estructura de conexión con el servidor HTTP en el método “doInBackground” seguido con una llamada a un método que actualice la interfaz desde “onPostExecute” y la forma de analizar la respuesta del servidor se sigue en todas las actividades que necesiten conectarse al servidor HTTP. Puede haber ligeros cambios entre las clases, pero la estructura general es la expuesta anteriormente.

4.4.2. Actividad y servicio de emulación de etiquetas

La funcionalidad de la emulación de etiquetas de la aplicación se encuentra dividida entre una actividad y un servicio de Android.

La actividad sigue el esquema general de actividades de Android. Al ser creada, se enlazan los elementos de la interfaz de usuario con variables interna de la clase. Se recupera la información de usuario de las SharedPreferences y se le deja saber al usuario el contenido de la etiqueta NFC a emular, dándole un timestamp con el tiempo aproximado en el que está fichando en el trabajo.

Otro elemento que es inicializado al crear la actividad es un AlertDialog. Este AlertDialog se muestra por pantalla sirviendo un mensaje que indica que el teléfono está emulando una etiqueta NFC cuando se pulsa el botón encargado de iniciar la emulación.

Cuando se pulsa el botón para servir la etiqueta, se llama al método "initEmulationService()". Este método se encarga de comprobar si NFC está activado, crear un Intent registrando la información del usuario recuperada desde las SharedPreferences, hacer que el AlertDialog sea visible, y lanzar el servicio a través del Intent de Android:

```
private void initEmulationService() {
    // Solo podemos proceder si NFC está activado
    if (checkNFCStatus()) {
        Intent intent = new Intent( packageContext: this, EmulateNFCTagService.class);
        intent.putExtra( name: "ndefMessage", sharedPreferences.getString( key: "USER",  defValue: "error"));
        dialog.show();
        startService(intent);
        Log.i( tag: "initEmulationService",  msg: "Launched emulation service");
    }
}
```

Figura 31. Inicialización del servicio de emulación de etiquetas.

Asimismo, al crear el AlertDialog se registra un listener para que cuando se haga "dismiss" de cualquier forma, se pare el servicio de emulación a través de otro Intent. Esto está recogido en el método "stopEmulationService()":

```
private void stopEmulationService() {
    stopService(new Intent( packageContext: this, EmulateNFCTagService.class));

    Log.i( tag: "stopEmulationService",  msg: "Stopped emulation service");
}
```

Figura 32. Método encargado de parar el servicio de emulación de etiquetas.

El último elemento importante que explicar de esta actividad es el atributo de tipo BroadcastReceiver de la clase. Un BroadcastReceiver es un objeto encargado de recoger broadcast lanzados en la aplicación como forma de comunicación entre diferentes componentes de la aplicación, en este caso, entre el servicio de emulación de etiquetas y la actividad de emulación de etiquetas.

Cada vez que la actividad entra en primer plano se debe registrar el BroadcastReceiver:

```
@Override
protected void onResume() {
    super.onResume();
    LocalBroadcastManager.getInstance(this).registerReceiver(broadcastReceiver, new IntentFilter( action: "message"));
```

Figura 33. Método onResume de la actividad de emulación de etiquetas.

De la misma forma, cada vez que la actividad deja de estar en primer plano, se debe borrar este registro:

```
@Override
protected void onPause() {
    super.onPause();
    LocalBroadcastManager.getInstance(this).unregisterReceiver(broadcastReceiver);
```

Figura 34. Método onPause de la actividad de emulación de etiquetas.

En cuanto a la implementación del BroadCastReceiver, este objeto estará esperando un mensaje del servicio que le indique si la lectura de la etiqueta ha sido correcta. En caso de que así sea, se le notificará al usuario a través de AlertDialogs:

```
@Override
public void onReceive(Context context, Intent intent) {
    if(intent.hasExtra( name: "success") && showDialog) {
        dialog.dismiss();
        if(intent.getBooleanExtra( name: "success",  defaultValue: false)) {
            Log.i( tag: "BroadcastReceiver",  msg: "Successful write: " + showDialog);
            showOkDialog();
            showDialog = false;
        }
        else {
            Log.i( tag: "BroadcastReceiver",  msg: "Error while sending NFC tag");
            showErrorDialog();
        }
    }
    else {
        showDialog = true;
    }
}
```

Figura 35. Método encargado de recoger los broadcasts lanzados por el servicio de emulación de etiquetas.

showDialog es un atributo encargado de evitar que la aplicación enseñe dos AlertDialogs. La necesidad de este atributo radica en que desde que se hizo la implementación con la API "ReaderMode" (explicada en el apartado de lectura de etiquetas), el lector necesita leer la tarjeta dos veces para que sea correcta. Es por esto que el servicio se ejecuta dos veces, enviando dos broadcasts y creándose dos AlertDialogs.

Esto es un comportamiento del cual no se ha encontrado más información y es algo que no ocurría cuando se hizo la implementación con "ForegroundDispatch" (explicado en el apartado de lectura de etiquetas), por lo que se entiende que es un bug de la API de "ReaderMode". Se ha intentado solventar de esta forma y funciona correctamente para los dos teléfonos con los que se ha probado, pero puede dar lugar a errores si para otros teléfonos no se tienen que hacer las dos lecturas o se tienen que hacer más de 2 para que se lea correctamente.

Con esto queda explicada la actividad de Android y pasamos al servicio de emulación de tarjetas.

En primer lugar, para el funcionamiento tanto del emulador como del lector de tarjetas, se tiene que registrar la petición de permisos para acceder a la función de NFC en el manifiesto:

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

Figura 36. Declaración de permisos de NFC.

Tenemos que especificar que la aplicación solo funcionará si el teléfono cuenta con la característica de NFC. En el caso del emulador, necesitamos también de la característica HCE (Host Card Emulation):

```
<uses-feature
        android:name="android.hardware.nfc"
        android:required="true"/>
<uses-feature
        android:name="android.hardware.nfc.hce"
        android:required="true"/>
```

Figura 37. Declaración de características necesarias (NFC y HCE).

Por último, al declarar el servicio en el manifiesto debemos pedir el permiso "BIND_NFC_SERVICE". Este permiso es necesario para que solo el sistema operativo Android sea capaz de enlazarse con el servicio de emulación de tarjetas. En la declaración de servicio hay que declarar un Intent-Filter.

```
<service
        android:name=".EmulateNFCTagService"
        android:enabled="true"
        android:exported="true"
        android:permission="android.permission.BIND_NFC_SERVICE">
    <intent-filter>
        <action android:name="android.nfc.cardemulation.action.HOST_APDU_SERVICE"/>
    </intent-filter>

    <meta-data
            android:name="android.nfc.cardemulation.host_apdu_service"
            android:resource="@xml/apduservice"/>
</service>
```

Figura 38. Descriptor del servicio de emulación de etiquetas.

Esto es debido a que el servicio de HCE necesita recoger los Intents a través de este Intent-Filter, no puede hacerlo directamente. Para que la aplicación recoja solo los Intents de NFC que nos interesan, debemos crear un archivo XML especificando que tipos de aplicaciones ofrece nuestro servicio de emulación de etiquetas.

En este fichero XML se deben recoger todas las AID (application ID) que nuestro servicio esté dispuesto a procesar. En nuestro caso, solamente tenemos el AID que identifica a la aplicación de etiquetas de tipo 4 con mensajes NDEF:

```xml
<?xml version="1.0" encoding="utf-8"?>

<host-apdu-service xmlns:android="http://schemas.android.com/apk/res/android"
                   android:description="TFG - APDU" android:requireDeviceUnlock="false">
    <aid-group android:description="NDEF Application - TFG"  android:category="other" >
        <aid-filter android:name="D2760000850101"/>
    </aid-group>
</host-apdu-service>
```

Figura 39. Archivo XML que recoge los AID asociados al servicio de emulación de etiquetas.

La clase Java que implementa el servicio de emulación de etiquetas debe heredar de la clase HostApduService, el cual es un servicio especial de Android para HCE.

Cuando un servicio se lanza a partir de un Intent, el primer método de esta clase que se ejecuta es “onStartCommand”. En nuestro caso, se intenta recuperar la información adicional del Intent con el que se lanzó el servicio, es decir, la información de usuario que va a ser servida como mensaje NDEF. Se calcula NLEN y se prepara para poder servir el mensaje al recibir comandos APDU:

```java
public int onStartCommand (Intent intent, int flags, int startId) {
    if (intent.hasExtra( name: "ndefMessage")) {
        ndefMessage = new NdefMessage(createTextRecord( language: "en", intent.getStringExtra( name: "ndefMessage"), NDEF_ID));
        ndefMessageBytes = ndefMessage.toByteArray();
        ndefMessageNLEN = fillByteArrayToFixedDimension(BigInteger.valueOf(ndefMessageBytes.length).toByteArray(),  fixedSize: 2);
        Log.i( tag: "EmulateNFCTagService",  msg: "onStartCommand() | NDEF" + ndefMessage.toString());
    }

    return Service.START_STICKY;
}
```

Figura 40. Método onStartCommand del servicio de emulación de etiquetas.

Para crear el mensaje NDEF, se debe crear un registro NDEF llamando al método auxiliar “createTextRecord”. Este método se encarga de codificar el mensaje añadiendo información sobre el idioma construyendo de esta forma la payload:

```java
private NdefRecord createTextRecord(String language, String text, byte[] id) {
    byte[] languageBytes;
    byte[] textBytes;

    languageBytes = language.getBytes(StandardCharsets.US_ASCII);
    textBytes = text.getBytes(StandardCharsets.UTF_8);

    byte[] recordPayload = new byte[1 + (languageBytes.length & 0x03f) + textBytes.length];

    recordPayload[0] = (byte) (languageBytes.length & 0x03F);

    System.arraycopy(languageBytes, srcPos: 0, recordPayload, destPos: 1, length: languageBytes.length & 0x03F);
    System.arraycopy(textBytes, srcPos: 0, recordPayload, destPos: 1 + (languageBytes.length & 0x03F), textBytes.length);

    return new NdefRecord(NdefRecord.TNF_WELL_KNOWN, NdefRecord.RTD_TEXT, id, recordPayload);
}
```

Figura 41. Método auxiliar createTextRecord del servicio de emulación de etiquetas.

Una vez tenemos el mensaje NDEF, se convierte a un array de bytes y se calcula su longitud, rellenando el array de bytes de NLEN para que tenga un tamaño fijo de 2 posiciones.

En este punto, el servicio está activo y esperando comandos APDUs que procesar. Cuando un lector de tarjetas NFC se acerca al teléfono y encuentra el servicio de emulación, se le sirve la etiqueta. El lector y la tarjeta siguen el protocolo de comunicación definido al principio de la memoria en el apartado Type 4 tags.

Para ello, la clase Java tiene arrays de bytes como atributos que identifican todos y cada uno de los comandos APDU y las respuestas APDU a estos comandos. A modo de ejemplo:

```java
private final byte[] CAPABILITY_CONTAINER_OK = {
        (byte) 0x00, // CLA - Class - Class of instruction
        (byte) 0xa4, // INS - Instruction - Instruction code
        (byte) 0x00, // P1  - Parameter 1 - Instruction parameter 1
        (byte) 0x0c, // P2  - Parameter 2 - Instruction parameter 2
        (byte) 0x02, // Lc field    - Number of bytes present in the data field of the command
        (byte) 0xe1, (byte) 0x03 // file identifier of the CC file
};

private final byte[] READ_CAPABILITY_CONTAINER = {
        (byte) 0x00, // CLA - Class - Class of instruction
        (byte) 0xb0, // INS - Instruction - Instruction code
        (byte) 0x00, // P1  - Parameter 1 - Instruction parameter 1
        (byte) 0x00, // P2  - Parameter 2 - Instruction parameter 2
        (byte) 0x0f  // Lc field    - Number of bytes present in the data field of the command
};
```

Figura 42. Ejemplo de comandos APDU.

Y las respuestas:

```java
private final byte[] APDU_OKAY = {
        (byte) 0x90, // SW1 Status byte 1 - Command processing status
        (byte) 0x00  // SW2 Status byte 2 - Command processing qualifier
};

private final byte[] APDU_ERROR = {
        (byte) 0x6A, // SW1 Status byte 1 - Command processing status
        (byte) 0x82  // SW2 Status byte 2 - Command processing qualifier
};
```

Figura 43. Ejemplo de respuestas APDU.

Estos arrays de bytes son los bytes especificados en el documento referente a la especificación de las etiquetas de tipo 4. El enlace a este documento se puede encontrar en la bibliografía.

Cuando el teléfono recibe un C-APDU, se llama al método "processCommandApdu", pasando el C-APDU como parámetro. Simplemente tenemos que comprobar que C-APDU es comparándolo con los C-APDU que tenemos reconocidos y responder en consecuencia.

Antes de que empecemos a comprobar los C-APDU recibidos y compararlos con los del protocolo, primero debemos comprobar que tenemos un mensaje NDEF que servir, es decir, que el servicio de emulación de etiquetas NFC se ha lanzado a través de un Intent de la actividad de emulación de etiquetas.

Es necesario hacer esta comprobación debido a que, como se ha explicado anteriormente, el servicio de emulación se podría haber iniciado si el teléfono ha registrado un Intent de NFC dirigido al AID especificado en el archivo XML. En caso de que esto haya ocurrido, el servicio no tiene la información del usuario y, por lo tanto, la aplicación accederá a posiciones de memoria que no han sido inicializadas, por lo que debemos parar el servicio.

```java
if(ndefMessage == null) {
    Log.i( tag: "EmulateNFCTagService",  msg: "processCommandApdu() | No NDEF message to write.");
    return APDU_ERROR;
}
```

Figura 44. Comprobación de la inicialización del servicio de emulación de etiquetas.

En caso contrario, si ndefMessage no es null, los C-APDU pueden empezar a ser procesados y podemos generar respuestas. Por ejemplo, en el caso de que se mande el primer comando (NDEF Tag Application Select):

```java
// First command: NDEF Tag Application Select
if(Arrays.equals(APDU_SELECT, commandApdu)) {
    Log.i( tag: "EmulateNFCTagService",  msg: "APDU_SELECT triggered.");
    return APDU_OKAY;
}
```

Figura 45. Ejemplo de procesamiento de los C-APDU.

El lector seguirá respondiendo con C-APDU, hasta que al final se pida el mensaje NDEF. La siguiente captura corresponde a la construcción de la respuesta para conseguir este mensaje NDEF:

```
// Sixth command: Read NDEF message
if (Arrays.equals(Arrays.copyOfRange(commandApdu,  from: 0,  to: 2), NDEF_READ_BINARY)) {
    int offset = Integer.parseInt(toHex(Arrays.copyOfRange(commandApdu,  from: 2,  to: 4)),  radix: 16);
    int length =  Integer.parseInt(toHex(Arrays.copyOfRange(commandApdu,  from: 4,  to: 5)),  radix: 16);

    byte[] fullResponse = new byte [ndefMessageNLEN.length + ndefMessageBytes.length];
    System.arraycopy(ndefMessageNLEN,  srcPos: 0, fullResponse,  destPos: 0, ndefMessageNLEN.length);
    System.arraycopy(ndefMessageBytes,  srcPos: 0, fullResponse, ndefMessageNLEN.length, ndefMessageBytes.length);

    Log.i( tag: "EmulateNFCTagService",  msg: "NDEF_READ_BINARY triggered");
    Log.i( tag: "EmulateNFCTagService",  msg: "READ_BINARY - OFFSET: " + offset + " - LEN: " + length);

    byte[] slicedResponse = Arrays.copyOfRange(fullResponse, offset, fullResponse.length);

    // Construimos la respuesta
    int realLength = Math.min(slicedResponse.length, length);
    byte[] response = new byte[realLength + APDU_OKAY.length];

    System.arraycopy(slicedResponse,  srcPos: 0, response,  destPos: 0, realLength);
    System.arraycopy(APDU_OKAY,  srcPos: 0, response, realLength, APDU_OKAY.length);

    READ_CAPABILITY_CONTAINER_CHECK = false;
    sendBroadcast( success: true);
    return response;
}
```

Figura 46. Creación de R-APDU con el mensaje NDEF.

Como podemos comprobar, justo antes de enviar el R-APDU con el mensaje NDEF, se llama al método "sendBroadCast", pasándole como parámetro un booleano a true.

En caso de que algún comando NDEF no esté reconocido, se registra el error, se manda un broadcast con el booleano a false y se devuelve un R-APDU de error:

```
Log.e( tag: "EmulateNFCTagService",  msg: "processCommandApdu() | Cannot recognize APDU command.");
sendBroadcast( success: false);
return APDU_ERROR;
```

Figura 47. Respuesta con R-APDU de error.

En cuanto al método "sendBroadcast", simplemente se crea un Intent que registra si la lectura de la etiqueta NFC emulada es correcta y se manda el broadcast a través del LocalBroadCastManager:

```
private void sendBroadcast(boolean success) {
    Intent intent = new Intent( action: "message");
    intent.putExtra( name: "success", success);
    LocalBroadcastManager.getInstance(this).sendBroadcast(intent);
}
```

Figura 48. Método encargado de enviar los broadcasts desde el servicio de emulación de etiquetas.

El broadcast llegará a la actividad de emulación de etiquetas y seguirá la lógica definida anteriormente.

4.4.3. Actividad de lectura de etiquetas

La funcionalidad de lectura de etiquetas NFC se recoge en la actividad "ReadNFCActivity". Esta clase, como el resto de las actividades de Android, extienden a la clase "AppCompatActivity". En este caso, para poder leer las etiquetas internamente en la clase y no necesitar de una clase auxiliar, la clase "ReadNFCActivity" también implementa la interfaz "NfcAdapter.ReaderCallback".

La interfaz de usuario asociada a la actividad es bastante sencilla, cuenta con un botón para lanzar la lectura de etiquetas NFC y un TextView para dejarle saber al usuario administrador el resultado de la lectura.

De cara a la implementación de la lectura de etiquetas se probaron dos acercamientos. El primero de ellos fue haciendo uso de la clase "ForegroundDispatch".

ForegroundDispatch es un sistema de Android diseñado para la lectura de etiquetas NFC. Este sistema se activa cuando la actividad de Android está en primer plano, y debe ser desactivado si la actividad deja de estar en primer plano. ForegroundDispatch se encarga de interceptar los Intents de NFC que le llegan al sistema Android, teniendo prioridad sobre el resto del sistema. La implementación es bastante simple y funciona perfectamente siempre que el teléfono móvil no sirva también etiquetas NFC a través de HCE.

El problema es que el chip NFC del teléfono está haciendo un sondeo constante, buscando etiquetas cercanas. Debido a que el emulador de etiquetas necesita registrar un Intent-Filter en su manifiesto para su funcionamiento, este servicio de emulación de etiquetas puede ser lanzado y recogerá los Intents de NFC que coincidan con el AID D2760000850101h, aunque la aplicación no esté iniciada.

Cuando dos teléfonos se acercan entre sí, uno de ellos en modo lectura y el otro en modo emulación, aunque el segundo teléfono esté en modo emulación, el sondeo de búsqueda de etiquetas NFC cercanas continúa. El teléfono en modo emulación manda el Intent de NFC con el AID registrado en el Intent Filter, por lo que el servicio de emulación de etiquetas del teléfono que está en modo lectura es lanzado. Esto produce que la lectura de etiquetas sea cancelada. Es

por esto que ForegroundDispatch tuvo que ser descartado en el momento en el que se empezaron a hacer pruebas sobre el sistema y se descubrió el problema.

La segunda opción y la implementada finalmente es haciendo uso de la funcionalidad "ReaderMode" de la clase "NFCAdapter".

ReaderMode limita el controlador de NFC del teléfono a que pare de servir ningún tipo de servicio NFC que no sea la lectura de etiquetas cercanas. Permite también limitar el tipo de etiquetas que se quieren leer, teniendo que especificar el tipo de etiquetas NFC que se quieren leer, si se quieren obviar las etiquetas con mensajes NDEF, etcétera. El principal problema que se encontró a la hora de implementar el lector con ReaderMode fue la falta de documentación que hay con esta API.

Volviendo al funcionamiento de la actividad, empecemos a analizar el código.

En un primer momento, como en el resto de las actividades Android, se enlazan los elementos de la interfaz con atributos de la clase y se registra un listener para el botón de lectura. El listener está encargado de llamar al método "enableReaderMode()", cuyo código es el siguiente:

```
private void enableReaderMode() {
    // Solo podemos proceder si NFC está activado
    if (checkNFCStatus()) {
        txtTagContent.setText("");
        Bundle options = new Bundle();
        // Se crea una opción para que haya un delay de 1 segundo entre lecturas
        options.putInt(NfcAdapter.EXTRA_READER_PRESENCE_CHECK_DELAY, 1000);

        // Se activa el modo lectura para tarjetas NFC de tipo A
        nfcAdapter.enableReaderMode( activity: this,  callback: this, NfcAdapter.FLAG_READER_NFC_A, options);
        Log.i( tag: "enableReaderMode",  msg: "Reader mode enabled");

        dialog.show();
    }
}
```

Figura 49. Método encargado de activar ReaderMode.

Este código se encarga de limpiar el TextView que ve el usuario en la interfaz gráfica. Después prepara la llamada para activar ReaderMode. Se activa un flag para evitar que haya un delay de un segundo entre lecturas, y se activa un segundo flag para que el modo lectura recoja solo etiquetas NFC de tipo A. A parte de esto, se le debe proporcionar una instancia de la actividad, y la una instancia de la clase que implementa "NfcAdapter.ReaderCallback", en este caso, por simplicidad, se escogió que la propia actividad implementase los métodos de la interfaz y que simplemente se registrase a sí misma como ReaderCallback. Por último, se llama a enseñar un AlertDialog en pantalla que estará activo hasta que pare la lectura.

El AlertDialog es creado al crear la actividad. Se registra un listener para que cuando el AlertDialog se haga desaparecer (dismiss), ya sea porque se ha pulsado el botón de cancelar,

minimizando la aplicación, o haciendo click en otras partes de la pantalla, se desactive el modo lectura.

El código es el siguiente:

```
// Se crea el AlertDialog
AlertDialog.Builder builder = new AlertDialog.Builder( context: this);
LayoutInflater inflater = getLayoutInflater();
View customView = inflater.inflate(R.layout.progress_alert_dialog,  root: null);
TextView messageView = (TextView) customView.findViewById(R.id.loading_msg);
builder.setView(customView);
builder.setNegativeButton( text: "Cancelar",  listener: null);
builder.setOnDismissListener(new DialogInterface.OnDismissListener() {
    @Override
    public void onDismiss(DialogInterface dialog) {
        disableReaderMode();
    }
});
messageView.setText("Leyendo etiquetas NDEF cercanas...");
dialog = builder.create();
```

Figura 50. Creación del AlertDialog de lectura de etiquetas.

El método encargado de desactivar el modo lectura es mucho más sencillo:

```
private void disableReaderMode() {
    nfcAdapter.disableReaderMode( activity: this);
    Log.i( tag: "disableReaderMode",  msg: "Reader mode disabled");
}
```

Figura 51. Método encargado de desactivar ReaderMode.

Recapitulando, cuando el usuario administrador haga click en el botón de lectura de etiquetas, se registrará el teléfono en modo lectura NFC y se creará un AlertDialog. Si se cierra este AlertDialog de cualquier forma, se desactivará el modo lectura.

Cuando la actividad registre un Intent de NFC que coincida con el flag que hemos establecido anteriormente (NFC tipo A), se llamará al método de la interfaz "NfcAdapter.ReaderCallback", "onTagDiscovered".

Este método recoge un objeto de tipo Tag. Los objetos de tipo Tag representan etiquetas NFC descubiertas. Una vez se ha descubierto la etiqueta, debemos recuperar su "TechList", es decir, la lista de tecnologías aceptadas por la etiqueta. En caso de que alguna de estas tecnologías sea NDEF, podemos proceder ya que coincide con lo que estamos buscando.

Se recupera el contenido NDEF de la etiqueta, se recupera el mensaje NDEF dentro de la etiqueta, el registro NDEF que haya dentro del mensaje y, por último, la payload de este mensaje, es decir, el texto:

```java
public void onTagDiscovered(Tag tag) {
    for (String tech : tag.getTechList()) {
        if (tech.equals(Ndef.class.getName())) {
            Ndef ndef = Ndef.get(tag);
            try {
                // Recuperamos el mensaje NDEF de la tarjeta
                ndef.connect();
                NdefMessage ndefMessage = ndef.getNdefMessage();
                NdefRecord record = ndefMessage.getRecords()[0];
                byte[] payload = record.getPayload();
```

Figura 52. Primera parte del método encargado de recoger las etiquetas leídas por el lector.

Una vez tenemos el contenido del mensaje NDEF, tenemos que parsear el mensaje quitando los bits de la codificación de texto y los indicadores de idioma del mensaje:

```java
// Parseamos el mensaje quitándole los caracteres innecesarios en base a su codificación.
String textEncoding = ((payload[0] & 128) == 0) ? "UTF-8" : "UTF-16";
int languageSize = payload[0] & 0063;
tagContent = new String(payload,  offset: languageSize + 1,  length: payload.length - languageSize - 1, textEncoding);
Log.i( tag: "onTagDiscovered", tagContent);
ndef.close();
```

Figura 53. Segunda parte del método encargado de recoger las etiquetas leídas por el lector.

De esta forma, en el atributo tagContent tenemos el contenido "limpio" de la etiqueta NDEF que se ha leído, que debería ser el email identificador del empleado que está fichando. El siguiente paso es llamar a la AsyncTask encargada de enviársela al servidor HTTP. El funcionamiento de estas AsyncTask ha sido descrito anteriormente, por lo que simplemente explicaremos la gestión de la respuesta del servidor.

El método "onPostExecute" llama al método "updateInterface", el cual recoge como parámetro el String de respuesta que el servidor HTTP está programado para devolver. En base a este String, se le notifica al usuario del resultado de la operación. Por último, se hace dismiss del AlertDialog, desactivando el modo lectura:

```
protected void updateInterface(String result) {
    switch (result.trim().toLowerCase()) {
        case "arrived_time updated":
            txtTagContent.setText("Se ha actualizado correctamente la hora de llegada del usuario " + tagContent);
            break;
        case "left_time updated":
            txtTagContent.setText("Se ha actualizado correctamente la hora de salida del usuario " + tagContent);
            break;
        case "table already updated":
            txtTagContent.setText("Error. El usuario " + tagContent + " ya ha fichado 2 veces hoy");
            break;
        case "update too soon":
            txtTagContent.setText("Error. No ha pasado 1 minuto desde que el usuario intentó fichar");
            break;
        case "no data to update":
            txtTagContent.setText("Error. El usuario " + tagContent + " no tiene datos de asistencia para el día de hoy");
            break;
        case "timeout":
            txtTagContent.setText("Error. No se ha podido conectar con el servidor. Inténtelo de nuevo más tarde");
            break;
    }

    dialog.dismiss();
}
```

Figura 54. Análisis de la respuesta del servidor frente a la lectura de etiquetas y actualización de la interfaz de usuario.

4.5. Aplicación de escritorio para administradores

Esta última aplicación se ha desarrollado para cubrir la necesidad de los usuarios administradores de tener una forma fácil de acceder y modificar los datos almacenados en la base de datos.

Se ha creado una aplicación Java haciendo uso de la librería Swing para crear la interfaz gráfica. La aplicación les da la posibilidad a los usuarios administradores de acceder a las llamadas de gestión a través de la ReST API. Las clases Java creadas todas heredan de la clase Swing [20] “javax.swing.Jframe”, es decir, todas las clases Java son clases de interfaz gráfica.

La lógica de la aplicación está pensada para mapear las tablas de la base de datos. Tenemos, por lo tanto, clases relacionadas con la tabla Employee, clases relacionadas con la tabla Schedule y, por último, clases relacionadas con la tabla Attendance.

En cuanto a la tabla Employee:

- **RegisterEmployee:** la clase tiene las herramientas necesarias para recuperar toda la información necesaria para registrar un nuevo empleado en la base de datos. Una vez se ha recogido toda la información, se hace la llamada al servidor HTTP para intentar insertarlo en la base de datos y se analiza la respuesta del servidor.

- **DeleteEmployee:** esta clase pide el email de un empleado y le manda una petición al servidor HTTP para eliminar al usuario de la base de datos.

- **EmployeeInfo:** esta clase está pensada para recuperar información sobre un usuario. La principal idea es recuperar el email del empleado para usarlo como identificador en el resto de las partes de la aplicación. La consulta para buscar el email se hace a partir del nombre del empleado.

Para la tabla Schedule:

- **AddSchedule:** se recogen los datos del empleado sobre el que crear el registro de horario y la fecha en la que se quiere crear el horario. Después se manda la petición al servidor y se analiza la respuesta para ver si la inserción ha sido correcta.

- **DeleteSchedule:** se intenta borrar el registro de horario de un empleado a partir de su email y la fecha registrada para el horario. Se analiza la respuesta del servidor para ver si la eliminación ha sido correcta.

- **ScheduleInfo:** clase pensada de cara a consultar la información de horarios de un empleado. Se pide el email del empleado a buscar y el mes y año de la información de horarios para acortarla. Se muestra la información en una tabla para su mejor visualización.

Y en cuanto a la tabla Attendance:

- **AttendanceInfo:** clase encargada de consultar los registros de asistencia de un empleado. Se pide el email del empleado, el mes y el año a buscar en la base de datos. Se presentan los datos recuperados en una tabla para su mejor visualización.

- **AttendanceHourAnalysis:** clase encargada de generar un reporte con la recopilación de información sobre horas de trabajo para un trabajador. Se pide el email, el mes y el año que se quiere analizar. Se devuelve información sobre horas trabajadas, horas totales que debía trabajar, faltas de asistencia, etc.

Por último, con la finalidad de enlazar todas las clases anteriores y debido a la necesidad de que el usuario se identifique antes de darle acceso a estas funcionalidades, tenemos las siguientes clases:

- **Login:** es la clase con la función Main y el punto de acceso a la aplicación. Se piden las credenciales del usuario y se lanzan contra la base de datos. En caso de que el usuario exista se analiza la respuesta para saber si el usuario es un administrador. Solo se le permitirá acceso a la aplicación en caso de que tenga privilegios de administrador.

- **MainMenu:** menú principal que hace de nexo para el resto de las clases de la aplicación. Tiene enlaces al resto de funcionalidades, separadas en menús para cada una de las tablas (Employee, Schedule y Attendance).

Todas las clases que tienen llamadas para registrar/eliminar/consultar datos de la base de datos tienen la misma estructura. La estructura sigue el siguiente patrón:

1 Una vez el usuario ha rellenado los campos que se piden para hacer la llamada a la API, el usuario pulsa un botón.

2. Se recogen los datos introducidos por el usuario de la interfaz y se procesan en busca de posibles errores o campos en blanco. En caso de que el campo necesite procesamiento, se hace ahí. (Ej: recogida de datos de cara a crear un registro horario):

```
String email = jTextFieldEmail.getText();
String day = (jComboBoxDay.getSelectedIndex() > 8)
        ? (String) jComboBoxDay.getSelectedItem()
        : (String) "0" + jComboBoxDay.getSelectedItem();

String month = (jComboBoxMonth.getSelectedIndex() > 8)
        ? "" + (1 + jComboBoxMonth.getSelectedIndex())
        : "0" + (1 + jComboBoxMonth.getSelectedIndex());

String year = (String) jComboBoxYear.getSelectedItem();
String date = year + "-" + month + "-" + day;

String startTime = (String) jComboBoxArriveTime.getSelectedItem() + ":00";
String endTime = (String) jComboBoxLeaveTime.getSelectedItem() + ":00";
```

Figura 55. Ejemplo de recogida y formateo de datos en la aplicación de administración.

3. Si es necesario, se compila la información en un JSON. En cualquier caso, se hace la llamada al endpoint de la API correspondiente en función de lo que tengamos que hacer:

```
try {
    // Configurar la llamada HTTP al servidor ReST
    HttpURLConnection connection = (HttpURLConnection) new URL("http://192.168.1.136:8080/api/schedule").openConnection();
    connection.setConnectTimeout(5000);
    connection.setRequestMethod("POST");
    connection.setRequestProperty("Content-Type", "application/json");
    connection.setDoOutput(true);

    // Escribir la petición JSON en el paquete
    OutputStream os = connection.getOutputStream();
    byte[] input = jsonRequest.getBytes(StandardCharsets.UTF_8);
    os.write(input, 0, input.length);
```

Figura 56. Ejemplo de conexión con el servidor HTTP en la aplicación de administración.

4. Se recoge la respuesta del servidor y en base a esta respuesta se le notifica al usuario de si la aplicación se ha realizado correctamente:

```
    switch (connection.getResponseCode()) {
        case 201:
            JOptionPane.showMessageDialog(this, "Horario añadido correctamente", "Horario añadido", JOptionPane.INFORMATION_MESSAGE);
            break;

        case 400:
            JOptionPane.showMessageDialog(this, "Error. El empleado no existe.", "Error", JOptionPane.ERROR_MESSAGE);
            break;

        case 409:
            JOptionPane.showMessageDialog(this, "Error. El empleado ya tiene un horario para esa fecha.", "Error", JOptionPane.ERROR_MESSAGE);
            break;

        default:
            JOptionPane.showMessageDialog(this, "Error. Ha ocurrido un error desconocido.", "Error", JOptionPane.ERROR_MESSAGE);
            break;
    }
} catch (java.net.SocketTimeoutException e) {
    JOptionPane.showMessageDialog(this, "Error. No se ha podido conectar con el servidor.", "Error", JOptionPane.ERROR_MESSAGE);
```

Figura 57. Ejemplo de procesamiento de respuesta del servidor HTTP en la aplicación de administración.

5. En caso de que se tenga que actualizar la interfaz porque es un método de consulta, se llama a una función auxiliar encargada de ello:

```java
private void fillTable (String jsonResponse) {
    try {
        JSONArray array = new JSONArray(jsonResponse);
        JSONObject obj;
        int arrayLength = array.length();
        tableModel.setRowCount(arrayLength);

        for(int i = 0; i < arrayLength; i++) {
            obj = array.getJSONObject(i);
            jTable.setValueAt(obj.getString("email"), i, 0);
            jTable.setValueAt(obj.getString("date"), i, 1);
            jTable.setValueAt(obj.getString("start_time"), i, 2);
            jTable.setValueAt(obj.getString("end_time"), i, 3);
        }
    } catch (JSONException e) {
        System.out.println(e.getMessage());
    }
}
```

Figura 58. Función encargada de rellenar un objeto de tipo jTable.

En cuanto a la clase Main, la clase implementa un “TabbedPane” para separar las funcionalidades de cada tabla. La lógica de la clase simplemente se encarga de establecer listeners para los eventos de pulsar los botones.

Cuando el usuario pulsa un botón, se instancia la nueva clase y se oculta la ventana del menú principal. Al terminar de hacer las gestiones que tenga que hacer el usuario en la nueva ventana, puede cerrar la ventana, abriéndose de nuevo la del menú principal.

La clase “MainMenu” tiene la siguiente forma:

```java
/**
 * Función encargada de gestionar el evento de pulsar el botón de
 * registro de empleado.
 * Se crea la nueva ventana y se hace que la actual pase a ser invisible.
 * @param evt evento de swing
 */
private void jButtonRegisterEmployeeActionPerformed(java.awt.event.ActionEvent evt) {
    new RegisterEmployee(this).setVisible(true);
    this.setVisible(false);
}

/**
 * Función encargada de gestionar el evento de pulsar el botón de
 * borrar empleados.
 * Se crea la nueva ventana y se hace que la actual pase a ser invisible.
 * @param evt evento de swing
 */
private void jButtonDeleteEmployeeActionPerformed(java.awt.event.ActionEvent evt) {
    new DeleteEmployee(this).setVisible(true);
    this.setVisible(false);
}
```

Figura 59. Ejemplos de métodos encargados de gestionar el evento de pulsar un botón en la interfaz.

En cuanto a la clase de Login, su estructura es bastante parecida a la de las clases de gestión de la base de datos. Recoge las credenciales del usuario, los manda contra el servidor y analiza los privilegios del usuario en caso de que exista. Solo en caso de que sea un administrador se lanzará el menú principal, en cualquier otro caso se le mostrará un mensaje de error al usuario:

```java
// Si la información recuperada está asociada a una cuenta
if (connection.getResponseCode() == 200) {
    Scanner scanner = new Scanner(connection.getInputStream());

    while (scanner.hasNextLine()) {
        response += scanner.nextLine() + "\n";
    }
    scanner.close();

    JSONObject obj = new JSONArray(response).getJSONObject(0);
    // Si el usuario es un admin
    if(obj.getInt("is_admin") == 1) {
        // Continuamos con el proceso de login
        JOptionPane.showMessageDialog(this, "Bienvenido "+ user +".", "Login correcto", JOptionPane.PLAIN_MESSAGE);
        new MainMenu().setVisible(true);
        this.dispose();
    }
    // En caso contrario, si el usuario no es un admin
    else {
        // El usuario no tiene permisos para entrar en el panel de administración
        jPasswordFieldPassword.setText("");
        JOptionPane.showMessageDialog(this, "Error. El usuario no es un administrador.", "Error", JOptionPane.ERROR_MESSAGE);
    }
}
// Si los credenciales no corresponden con los datos guardados en la base de datos
else {
    jPasswordFieldPassword.setText("");
    JOptionPane.showMessageDialog(this, "Error. Datos de inicio de sesión incorrectos.", "Error", JOptionPane.ERROR_MESSAGE);
}
```

Figura 60. Procesamiento de la respuesta del servidor HTTP en el caso del login.

5. Conclusiones

El uso de la tecnología NFC en el ámbito de la creación de sistemas para el control de asistencia de empleados puede tener muchas ventajas ya que permite que el proceso de fichaje de los empleados sea rápido y totalmente transparente para el trabajador. Nos permite estar seguros de la presencia del usuario en el lugar de trabajo a la hora de fichar, debido a las características del protocolo NFC.

La combinación de NFC con el sistema operativo Android permite la creación de sistemas como el desarrollado en este proyecto, eliminando el coste de infraestructura inherente al uso de tecnologías NFC (lectores NFC, software especializado, etiquetas físicas, etc.).

La implementación de emulación de tarjetas a través del servicio HCE de Android escogida en este proyecto nos permite además cubrir el caso de que los teléfonos Android de los trabajadores no cuenten con chip NFC. Debido a que el servicio HCE simplemente emula una etiqueta NFC física, se podrían hacer uso de etiquetas físicas de la misma forma que se usa el servicio de emulación, siempre que estas etiquetas implementen el mismo protocolo de comunicación.

Al haber escogido esta implementación a la hora de servir etiquetas NFC desde el dispositivo Android, tenemos un sistema que cubre ambos casos, el de los empleados que tienen un chip NFC en su dispositivo móvil, y el de los empleados que no tienen NFC en su dispositivo móvil y, por lo tanto, utilizan etiquetas NFC físicas. Esto proporciona una gran flexibilidad a los empleados de la empresa a la hora de trabajar con el sistema desarrollado.

En cuanto a la compleción de los objetivos propuestos, la idea de la que se partía inicialmente era la de la creación de un sistema basado en tecnologías móviles y NFC para el control de asistencia de los empleados de una organización. En ese sentido, se han cumplido el objetivo inicial de creación de este sistema, creándose además una serie de interfaces y herramientas para que los usuarios puedan interactuar de forma cómoda con el sistema.

Gracias a la modularización del sistema, si en un futuro se viese conveniente añadir nuevas funcionalidades, se podría hacer fácilmente.

6. Trabajo futuro

El proyecto, aunque cumple la idea principal del desarrollo de una aplicación de gestión de asistencia y control de horarios de empleados a través de un sistema totalmente funcional, tiene aun así algunos fallos o funcionalidades que faltan y que serían bastante importantes de cara a su implantación en un entorno profesional.

En primer lugar, el framework escogido para implementar el servidor HTTP (Flask), es un framework pensado para ser utilizado únicamente durante desarrollo, esto es, no se recomienda su uso para mantener servidores HTTP en un entorno de producción.

Esto se debe a que el servidor por defecto utilizado por Flask, Werkzeug, no funciona bien bajo una carga de trabajo alta. Werkzeug tiene varios problemas, entre los que se encuentran:

- Mal escalado del servidor en cuanto al número de peticiones.
- Seguridad mínima en el servidor HTTP.
- El servidor atiende a una sola petición concurrentemente.

Estos problemas se podrían solucionar implementando un WSGI (Web Server Gateway Interface) de Python, que se encargaría de recoger las peticiones HTTP que se hagan al servidor y redirigirlas a la capa de aplicación de Flask. Estos servidores WSGI son mucho más robustos que Werkzeug y aportan mucha más seguridad.

Se proponen dos servidores WSGI para utilizarse junto con Flask de cara al futuro; estos son *gunicorn* y *uWSGI*. Ambos son aptos para usarse en ambientes de producción.

En segundo lugar, la implementación de la API ReST está hecha a través de HTTP en vez de HTTPS, por lo que todas las peticiones se envían a través de texto plano. Otro de los cambios a realizar sería la implementación del protocolo HTTPS en el servidor consiguiendo certificados válidos para ello.

La API ReST tiene otra vulnerabilidad, y es que no se han hecho uso de tokens de autenticación para verificar la autenticidad del usuario en todo momento. Simplemente se requiere de un login inicialmente para verificar el tipo de usuario y darles acceso a las aplicaciones desarrolladas. Al no haber utilizado tokens de verificación, se podrían crear peticiones manualmente a través del navegador web o herramientas como cURL que manipulasen la base de datos sin necesidad de autenticarse en la aplicación. De cara al futuro, junto con la implantación de HTTPS, se debería crear una aplicación para la creación y verificación de tokens en las peticiones ReST a la API.

Por último, las interfaces gráficas asociadas a las aplicaciones son muy sencillas y, por lo tanto, bastante planas. Estas interfaces podrían ser enriquecidas para ser más llamativas para sus usuarios y, en el caso de que se viese oportuno, se podrían añadir nuevas funcionalidades a estas aplicaciones.

7. Coste del proyecto

El presupuesto para el presente proyecto se divide en el coste de la mano de obra y el coste de materiales e infraestructura.

En cuanto al coste de la mano de obra:

Tabla 2. Desglose de costes de mano de obra del proyecto.

CONCEPTO	**HORAS**	**COSTE/HORA**	**COSTE TOTAL**
INGENIERÍA			
Jefe de proyecto	25	32	800
Analista Programador	40	20	800
Programador	150	14	2.100
SECRETARÍA			
Secretaría	120	12	1.440
TOTAL			**5.140**

En el concepto de INGENIERÍA se incluye:

- Análisis del sistema en función de los requisitos:
  - Diseño y desarrollo de las partes que componen el sistema.
  - Diseño y desarrollo de las interfaces de usuario de las aplicaciones.
  - Estudio e implementación de los protocolos de comunicación de NFC a través de la API de Android.
- Depuración del código y pruebas sobre el sistema.
- Creación de la documentación del código.

En el concepto de SECRETARÍA se incluye:

- Redacción de la memoria.
- Creación de documentos e informes adicionales sobre el desarrollo del proyecto.

En cuanto al coste de los materiales, podemos hacer una diferenciación entre el coste de la infraestructura hardware, y el coste de las licencias software necesarias para el desarrollo del proyecto.

Tabla 3. Desglose del coste hardware.

CONCEPTO	UNIDADES	COSTE/UNIDAD	COSTE TOTAL
Ordenador de sobremesa, i5-6600k 3.5GHz, 512 GB SSD, tarjeta gráfica Nvidia GTX 970 4 GB GDDR5, 16GB memoria DDR4	1	1.118	1.118,87
Monitor de 24” AOC	2	120	240
Teléfono móvil LG L9 II	1	220	220
Teléfono móvil BQ Aquaris X Pro	1	260	260
TOTAL			**1.838,87**

Tabla 4. Desglose del coste de licencias software.

CONCEPTO	NÚMERO DE LICENCIAS	COSTE/UNIDAD	COSTE TOTAL
Windows 10	1	116,99	116,99
IntelliJ Idea 1 año	1	499	499
Microsoft Office	1	69,90	69,90
Toad Data Modeler	1	618,31	618,31
Python 3	1	0	0
Java	1	0	0
Apache Netbeans	1	0	0
Visual Studio Code	1	0	0
TOTAL			**1.304,20**

Tabla 5. Coste total de materiales.

CONCEPTO	COSTE TOTAL
COSTE TOTAL HARDWARE	1.838,87
COSTE TOTAL DE LICENCIAS SOFTWARE	1.304,20
COSTE TOTAL DE MATERIALES	**3.143,07**

Por último, se calculará el coste de los gastos generales. En este apartado se incluyen gastos como:

- Material de oficina (folios, bolígrafos, etc.).
- Dietas y desplazamientos.
- Gastos en infraestructura (luz, conexión a Internet, etc.).

Para calcular el coste de los gastos generales se aplicará un recargo del 20% sobre la suma del total de los costes materiales y el total de los costes de mano de obra.

Tabla 6. Gastos generales del proyecto.

CONCEPTO	COSTE TOTAL
GASTOS GENERALES	**1.656,61**

Con todo esto, nos queda el siguiente desglose del coste global del proyecto:

Tabla 7. Desglose del coste global del proyecto.

CONCEPTO	COSTE TOTAL
COSTE TOTAL DE MANO DE OBRA	5140
COSTE TOTAL DE MATERIALES	3.143,07
GASTOS GENERALES	1.656,61
COSTE TOTAL DE MATERIALES	**9.939,68**

El coste global del proyecto asciende a la cantidad de NUEVE MIL NOVECIENTOS TREINTA Y NUEVE EUROS CON SESENTA Y OCHO CÉNTIMOS.

8. Bibliografía

[1] Boletín oficial del estado – Real Decreto-ley 8/2019, de 8 de marzo, de medidas urgentes de protección social y de lucha contra la precariedad laboral en la jornada de trabajo. https://www.boe.es/boe/dias/2019/03/12/pdfs/BOE-A-2019-3481.pdf

[2] Deloitte – Estudio de Consumo Móvil en España. https://www2.deloitte.com/es/es/pages/technology-media-and-telecommunications/articles/consumo-movil-espana.html

[3] Ditendria – Todas las estadísticas sobre móviles que deberías conocer 2019. https://mktefa.ditrendia.es/blog/todas-las-estad%C3%ADsticas-sobre-m%C3%B3viles-que-deber%C3%ADas-conocer-mwc19

[4] Cloud-based web application with NFC for employee attendance management system. https://ieeexplore.ieee.org/document/8376516

[5] Near Field Communication (NFC) based Mobile Phone Attendance System for Employees. https://www.ijert.org/research/near-field-communication-nfc-based-mobile-phone-attendance-system-for-employees-IJERTV2IS3141.pdf

[6] Landoo. https://www.landoo.es/control-de-presencia-con-odoo-asistencias

[7] Time n Attendance (TNA): https://time-n-attendance.com/

[8] NFC – Wikipedia. https://en.wikipedia.org/wiki/Near-field_communication

[9] NFC Technology – Smart tec. https://www.smart-tec.com/en/auto-id-world/nfc-technology

[10] NFC – Blue Bite. https://www.bluebite.com/nfc

[11] NFC Forum Tag Types. https://www.dummies.com/consumer-electronics/the-nfc-forum-tag-types/

[12] ST25 NFC guide. https://www.st.com/content/ccc/resource/technical/document/technical_note/f9/a8/5a/0f/61/bf/42/29/DM00190233.pdf/files/DM00190233.pdf/jcr:content/translations/en.DM00190233.pdf

[13] NFC Forum Type 4 Tag Operation Specification 2.0. http://apps4android.org/nfc-specifications/NFCForum-TS-Type-4-Tag_2.0.pdf

[14] Android – Wikipedia. https://es.wikipedia.org/wiki/Android

[15] NFC – Documentación de Android. https://developer.android.com/guide/topics/connectivity/nfc

[16] NDEF – Documentación de Android. https://developer.android.com/reference/android/nfc/tech/Ndef

[17] NFC Forum. https://nfc-forum.org/

[18] Flask documentation. https://flask.palletsprojects.com/en/1.1.x/

[19] OpenAPI Specification – Swagger 2.0. https://swagger.io/specification/v2/

[20] Java documentation – Swing library for GUIs.
https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html

[21] Diagramas creados con la herramienta de draw.io. https://app.diagrams.net/

9. Anexo A - Manual de usuario

Este apartado de la memoria recopila los manuales de usuario de todas las partes de la aplicación accesibles por usuarios.

9.1. Aplicación Android

La aplicación de Android puede ser utilizada tanto por empleados sin privilegios como por usuarios administradores. Las funcionalidades de la aplicación a las que se puede acceder dependerán de qué tipo de usuario se haya identificado en la aplicación.

El proceso de login es el mismo para ambos tipos de usuarios. En la pantalla de login se nos pedirán las credenciales de nuestro usuario que deben coincidir con las credenciales guardadas en la base de datos.

En caso de que haya algún tipo de error, la interfaz gráfica nos lo hará saber a través de un mensaje Toast:

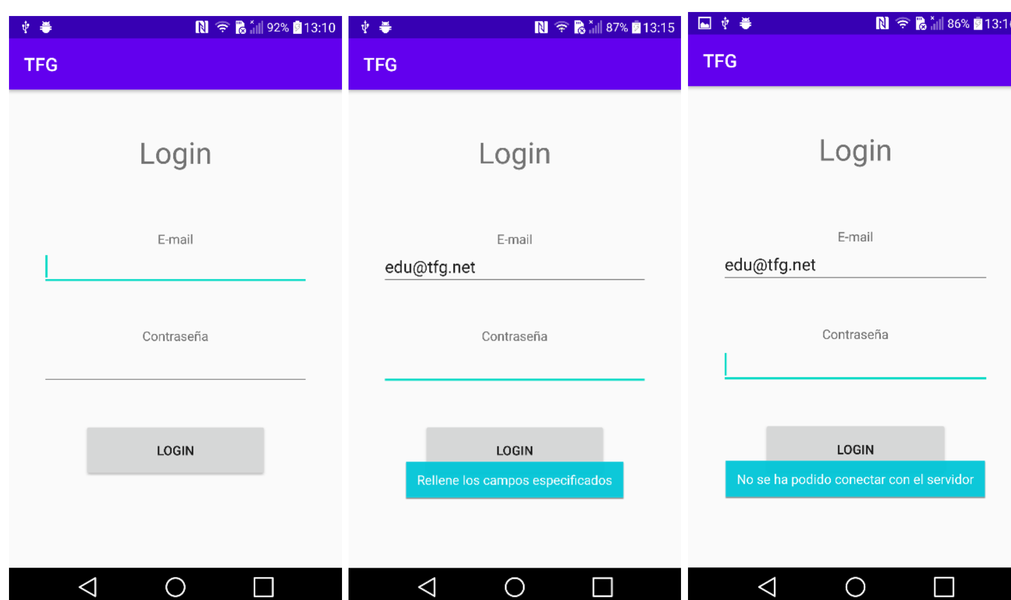


Figura 61. Capturas del panel de login en la aplicación Android.

Una vez hayamos iniciado sesión correctamente, se nos redirigirá al menú principal. Este menú principal cambiará en base a si el usuario tiene permisos de administración o no.

En caso de que el usuario quiera cerrar sesión, simplemente tiene que hacer click en el menú desplegable localizado en la esquina superior derecha en la pantalla del menú principal y cerrar sesión, redirigiéndole de nuevo a la pantalla de login:

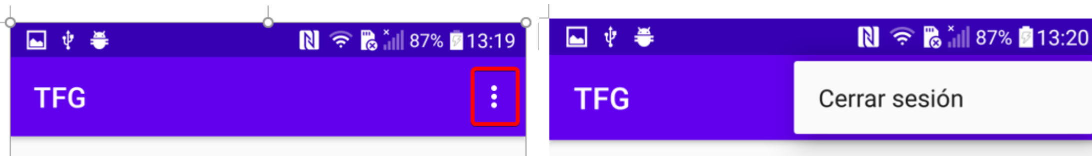


Figura 62. Menú desplegable dentro de la pantalla principal de la aplicación Android.

A partir de este punto, el manual de usuario se divide entre los tipos de usuarios que pueden utilizar la aplicación.

9.1.1. Empleado

Una vez hemos hecho login en la aplicación, la aplicación abre la actividad correspondiente al menú principal.

Desde el punto de vista de un empleado sin privilegios, podemos usar la aplicación para consultar nuestro horario de trabajo en un determinado día, y para hacer el proceso de fichaje a través de la emulación de etiquetas. Estas opciones se nos presentan a través de dos botones en el menú principal:

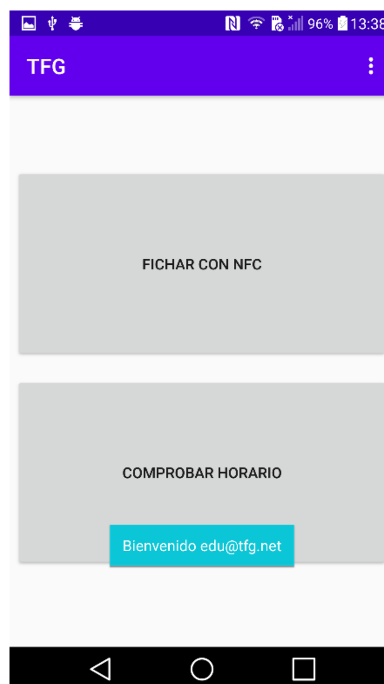


Figura 63. Menú principal de la aplicación Android para usuarios no administradores.

En el caso de que tengamos que fichar en el trabajo para registrar nuestra hora de entrada/salida, escogemos la opción de fichar con NFC.

Esto nos lleva a un menú en el cual se nos muestra la información que vamos a servir a través de NFC, esto es, nuestro correo electrónico que sirve de identificador en la aplicación y la hora aproximada a la que estamos fichando en el trabajo:

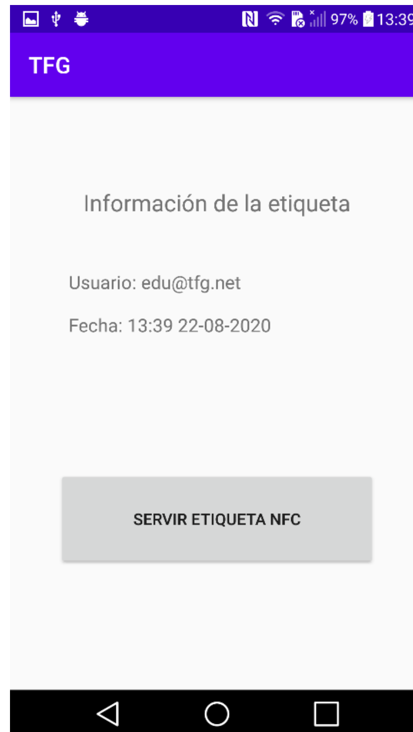


Figura 64. Menú de emulación de etiquetas.

Cuando estemos preparados para fichar en el trabajo, simplemente pulsamos el botón para servir la etiqueta NFC, con lo que saltará un diálogo informándonos de que estamos emulando la tarjeta y se está esperando a que se acerque el teléfono del administrador en modo lectura:

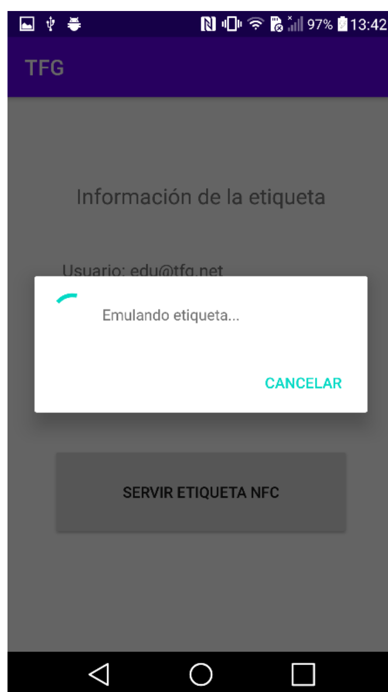


Figura 65. Emulación de etiquetas en proceso.

Mientras este mensaje se vea en pantalla, el servicio de emulación está funcionando. Si, por cualquier circunstancia, el mensaje desaparece, significa que la emulación ha terminado y se deberá volver a pulsar el botón de emulación en caso de que no hayamos fichado aún.

Cuando el usuario administrador acerque su teléfono en modo lectura y se produzca el intercambio de datos, el teléfono del empleado mostrará un mensaje comunicando el resultado de la comunicación:

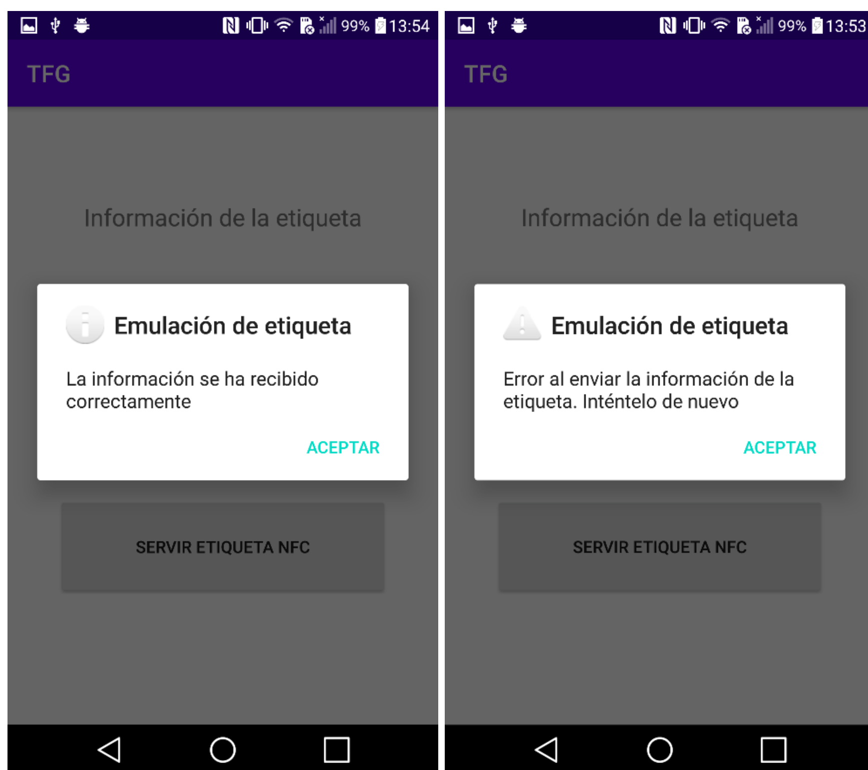


Figura 66. Posibles respuestas ante la emulación de etiquetas.

En el caso del mensaje que indica que la comunicación ha sido satisfactoria, esto no significa que el proceso de fichaje se haya producido correctamente, simplemente indica que el intercambio de mensajes a través de NFC ha funcionado. Para saber si el empleado ha fichado correctamente, dependemos de que el administrador que haya leído el mensaje nos deje conocer el mensaje que ha aparecido en su móvil.

Con esto concluye el servicio de emulación de etiquetas para fichar. El otro servicio disponible para empleados sin privilegios es el de consultar una fecha para comprobar el horario.

Al hacer click en el botón de Comprobar Horario en el menú principal, se nos redirige a una nueva pantalla. En esta interfaz se nos pide introducir una fecha en el formato especificado. En el caso de que no lo hagamos, se nos muestra un mensaje Toast pidiéndonos que cambiemos el formato de la fecha:

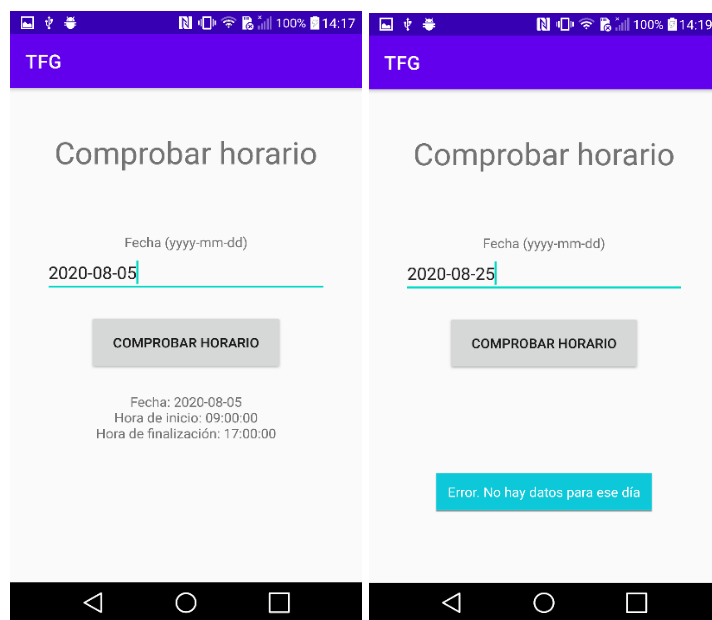


Figura 67. Menú de comprobar horarios para usuarios no administradores en la aplicación Android.

Una vez introduzcamos la fecha en el formato especificado, se lanzará la petición contra el servidor, mostrándonos la respuesta. En el caso de en la fecha especificada haya un horario para el empleado que está haciendo la petición, se nos mostrará la información sobre ese horario en un cuadro de texto justo debajo del botón:

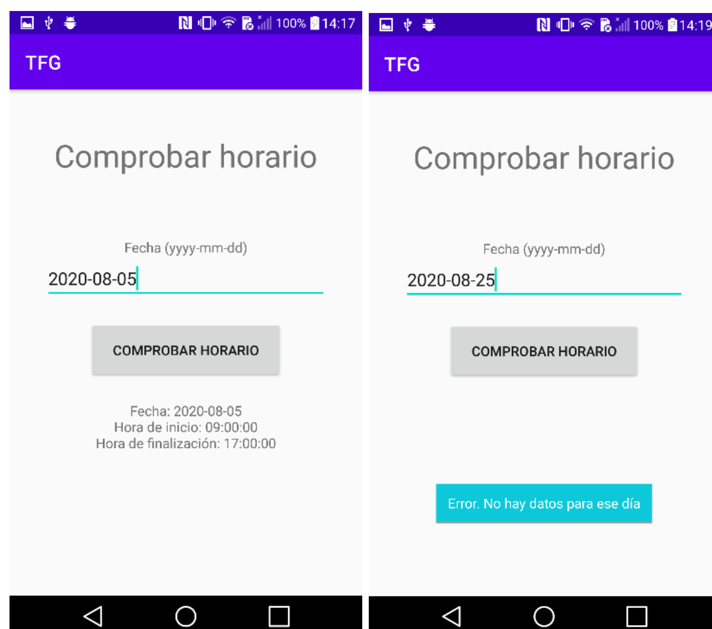


Figura 68. Posibles respuestas del menú de comprobar horario.

Con esto concluye el manual de usuario de la aplicación de Android para empleados sin privilegios de administración.

9.1.2. Administrador

Cuando hacemos login en la aplicación, también se nos redirige al menú principal siendo un administrador, pero en este caso los botones son distintos y llevan a otros servicios. El menú principal para un usuario administrador es de la siguiente manera:

Figura 69. Menú principal de la aplicación Android para usuarios administradores.

Se nos da la posibilidad de poner nuestro teléfono en modo lectura para ayudar a un empleado a fichar al entrar en su puesto de trabajo, y también de acceder a una interfaz para dar de alta a un empleado.

En la interfaz referente a la lectura de etiquetas NFC tenemos un cuadro de texto inicialmente vacío y un botón para iniciar la lectura de etiquetas. Al pulsar el botón salta un mensaje en primer plano indicándonos de que la lectura de etiquetas está activa. Si, por cualquier circunstancia, este mensaje se cierra, significa que el modo lectura se ha desactivado y tendremos que volver a pulsar el botón de lectura de etiquetas:

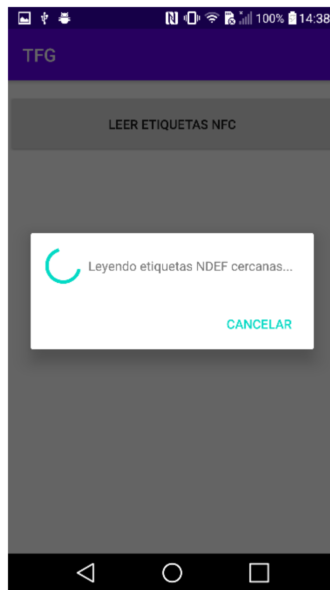


Figura 70. Lectura de etiquetas en proceso.

Cuando el empleado que está intentando fichar nos acerca su teléfono móvil en modo emulación de etiquetas, el teléfono del administrador recoge la información por NFC e intenta fichar en la base de datos. La respuesta del servidor se mostrará en el campo de texto que inicialmente estaba vacío.

Los posibles mensajes que indican que se ha fichado correctamente son:

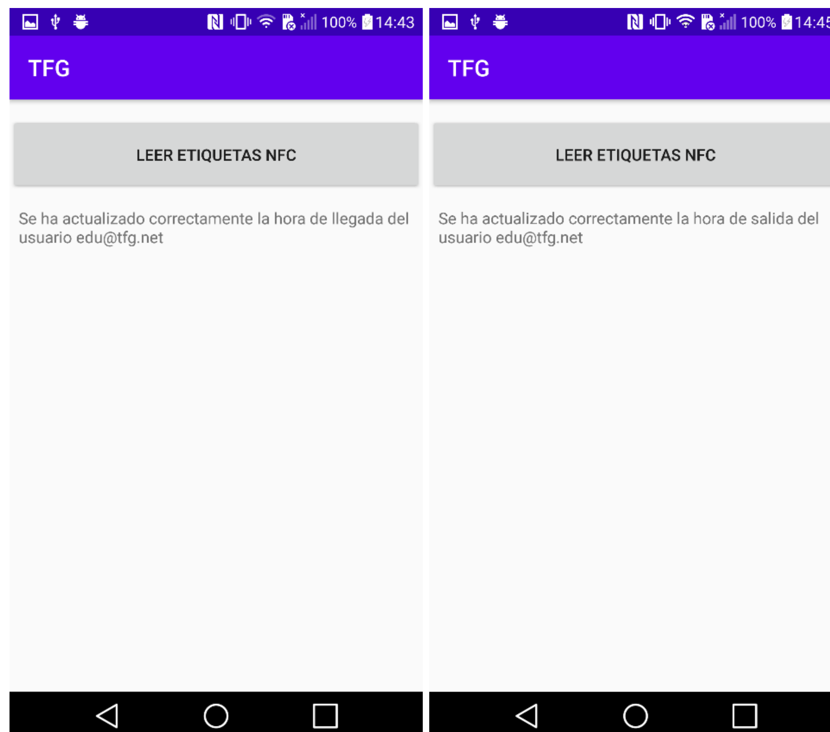


Figura 71. Mensajes de fichaje correcto en la aplicación.

Mientras que los posibles mensajes de error son:

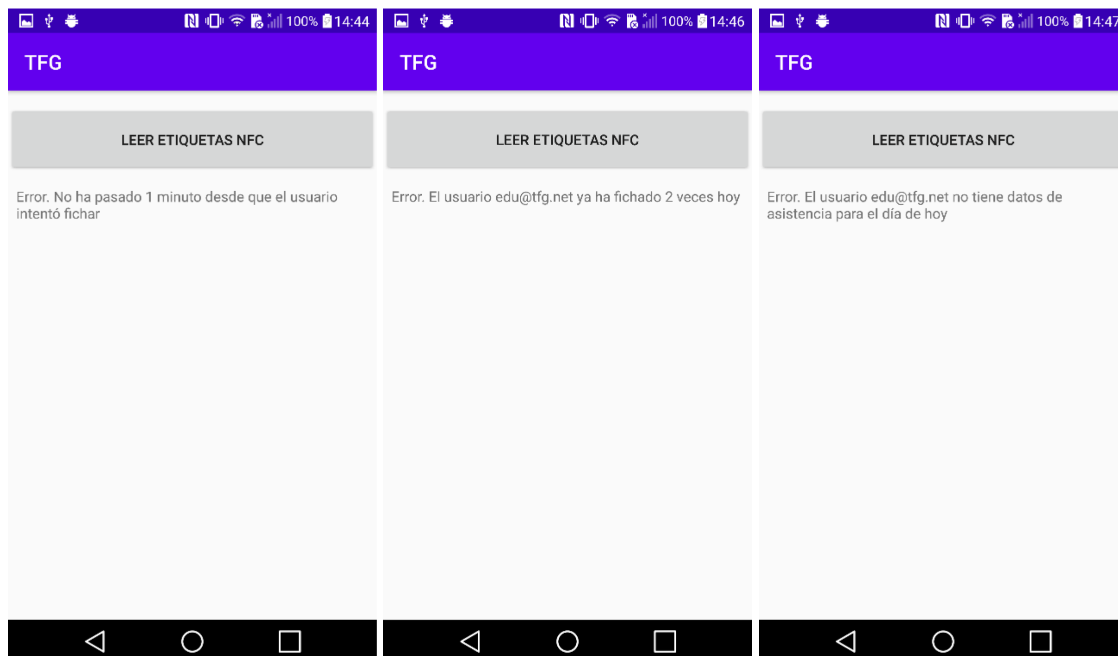


Figura 72. Mensajes de fichaje erróneo en la aplicación.

Es importante saber que el usuario sabe si el mensaje se ha recibido correctamente por NFC, pero no sabe si el proceso de fichar en la base de datos ha sido correcto. El usuario administrador se deberá encargar de hacerle saber la respuesta del servidor al empleado que está fichando.

Con esto, solo nos queda la interfaz de usuario referente a la creación de nuevos usuarios. Para acceder a esta interfaz, tenemos que hacer click en el botón de Alta de Empleado desde el menú principal. La interfaz tiene la siguiente forma:

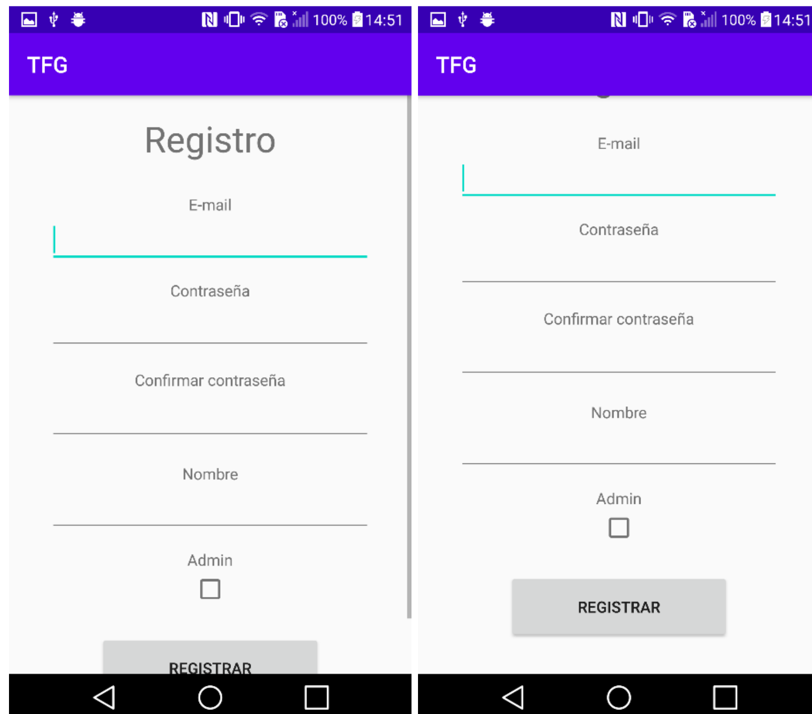


Figura 73. Menú de registro de nuevos empleados para usuarios administradores.

Esta interfaz implementa una ScrollView, es decir, se puede hacer scroll para ver las partes que no quepan en la pantalla. El funcionamiento de esta pantalla es bastante simple, simplemente tenemos que rellenar los datos del empleado que queramos crear, especificando si el nuevo usuario va a ser un administrador o no. En caso de que haya algún error al introducir los datos, la aplicación nos lo hará saber a través de un mensaje Toast:

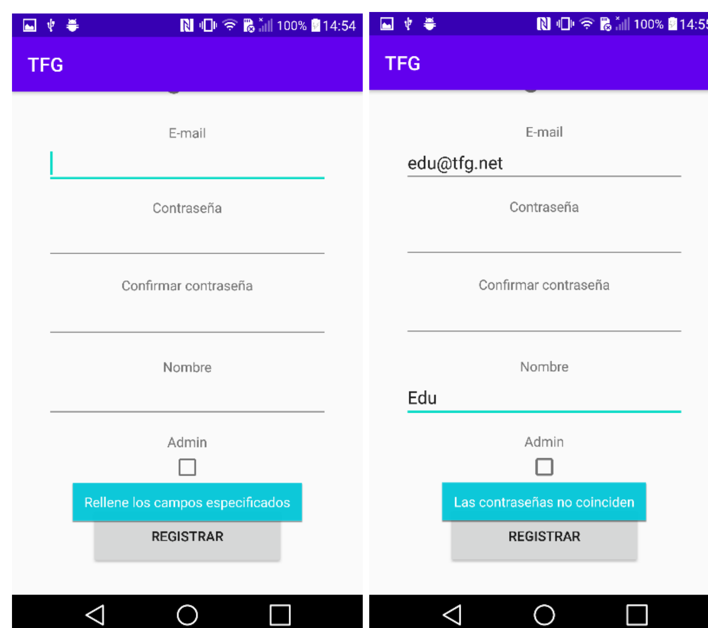


Figura 74. Posibles errores a la hora de rellenar el formulario de registro de nuevos empleados.

Si todos los campos son correctos, pulsamos el botón de Registrar y se enviarán los datos al servidor. La aplicación notificará al administrador del resultado de la operación a través de un mensaje popup:

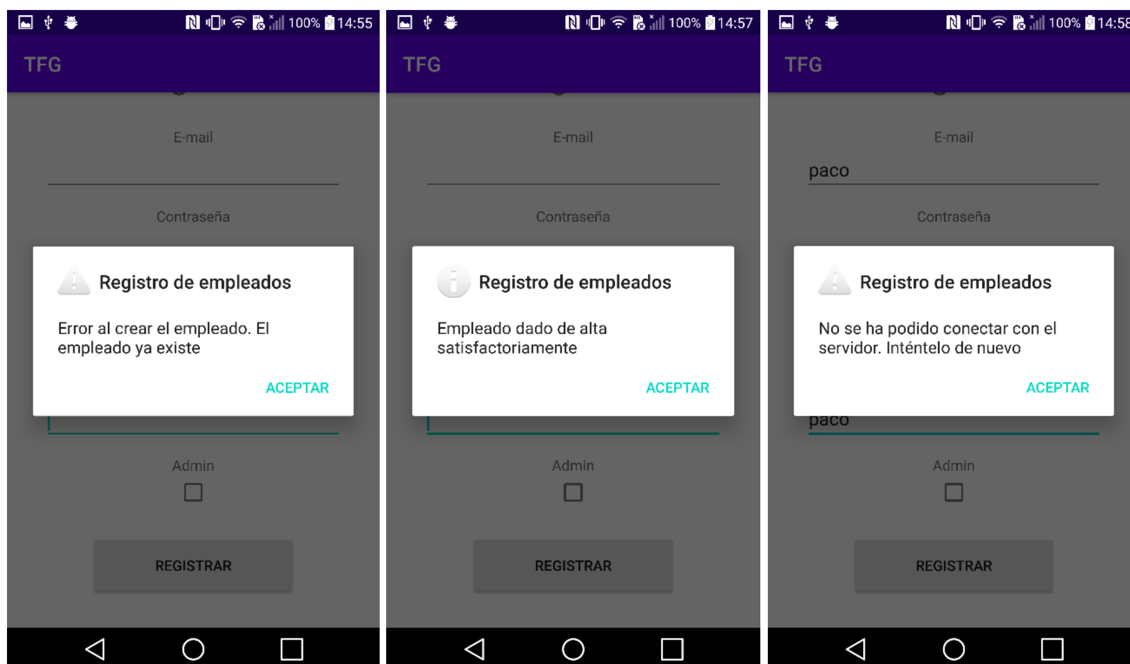


Figura 75. Posibles respuestas del servidor ante el intento de creación de un nuevo empleado a través de la aplicación Android.

9.2.Panel de administración

El panel de administración es una aplicación de escritorio con una interfaz gráfica basada en la librería Swing. Este panel de administración solo puede ser utilizado por usuarios con privilegios de administrador.

Cuando se abre la aplicación, se nos presenta una interfaz para iniciar sesión. Para ello, se nos piden las credenciales para compararlas con la base de datos. En caso de que haya algún tipo de error, se nos mostrará un mensaje indicándonoslo.

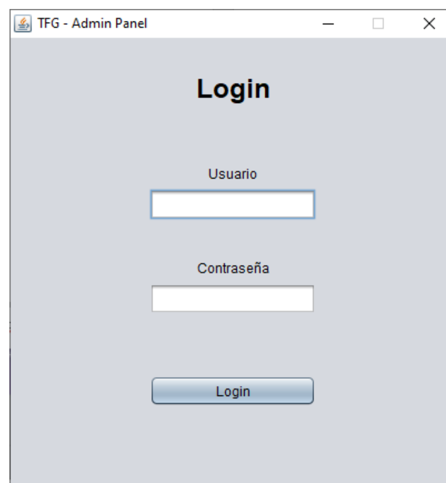


Figura 76. Pantalla de login en la aplicación de escritorio.

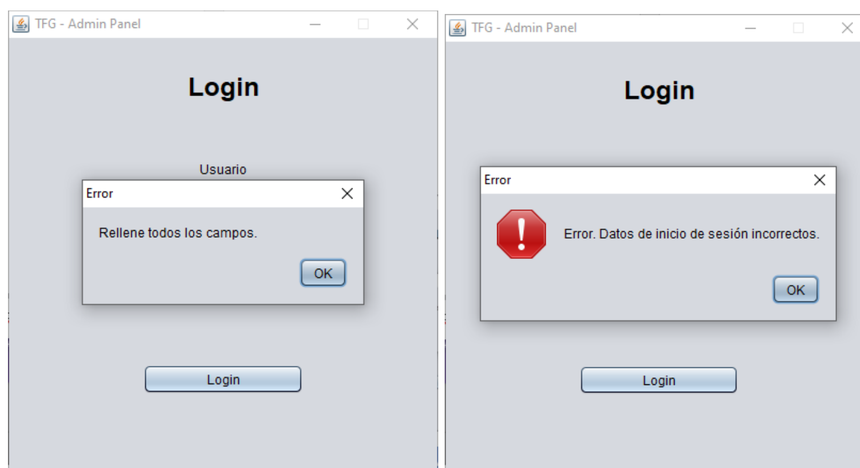


Figura 77. Mensajes de error ante el inicio de sesión en la aplicación de escritorio.

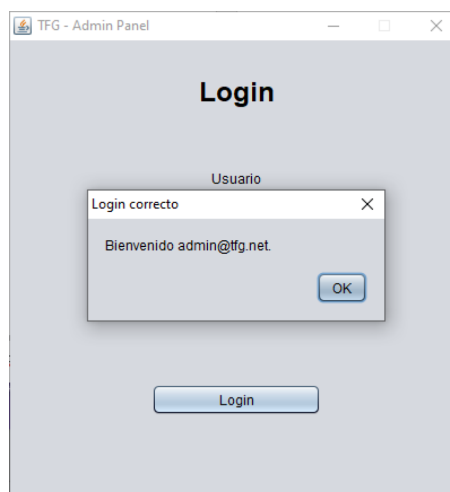


Figura 78. Mensaje de inicio de sesión correcto en la aplicación de escritorio.

Cuando se haga un login correcto, la aplicación nos redirige al menú principal de la aplicación. Este menú hace de nexo para acceder a todas las funcionalidades que ofrece el panel de administración.

Está dividido en 3 pestañas: la primera con funcionalidades de empleados, la segunda con funcionalidades sobre horarios de empleados, y la tercera con funcionalidades sobre registros de asistencia y control horario de empleados:

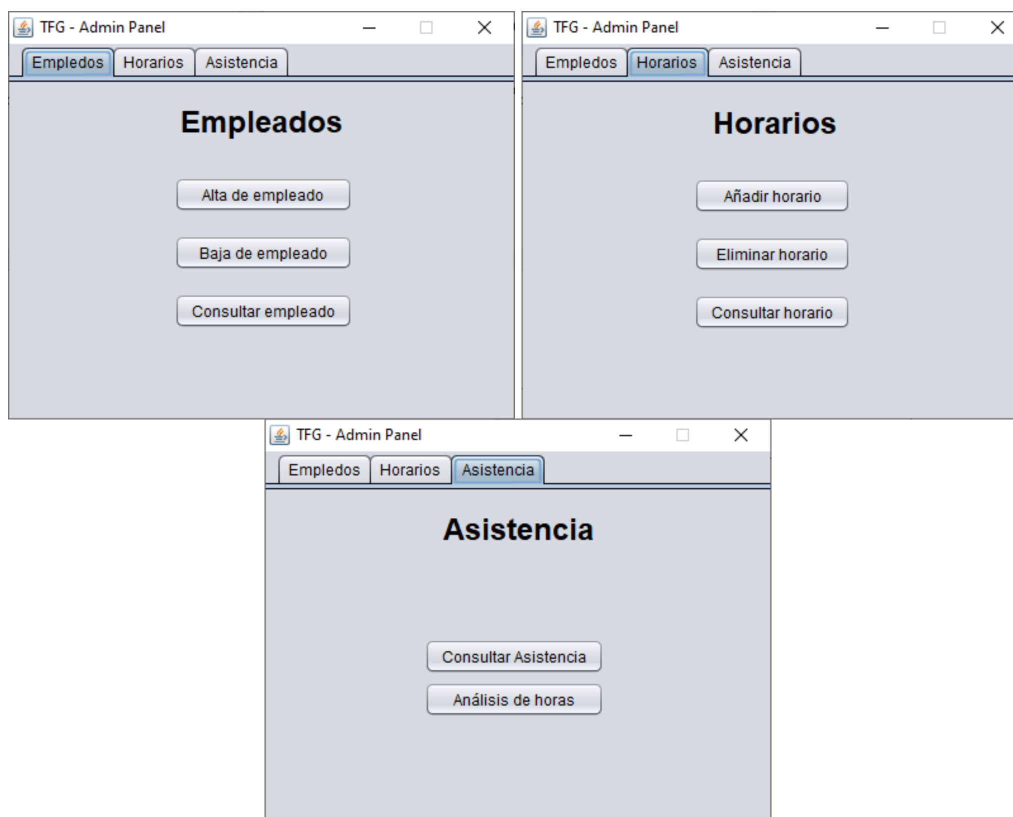


Figura 79. Pestañas del menú principal en la aplicación de escritorio para administradores.

Desde este menú, podemos decidir acceder a todas las funcionalidades de la aplicación. Cuando se abra una ventana nueva como resultado de haber pulsado alguno de estos botones, la ventana del menú principal se hace invisible. Para volver al menú principal, simplemente hay que cerrar la ventana de la funcionalidad que hayamos abierto.

En cuanto a las funcionalidades de empleado, si accedemos a la Alta de empleados nos recibe un menú con un formulario a rellenar. Si rellenamos todo con datos correctos, se envían los datos al servidor. La respuesta del servidor se enseñará por pantalla en un mensaje de tipo popup:

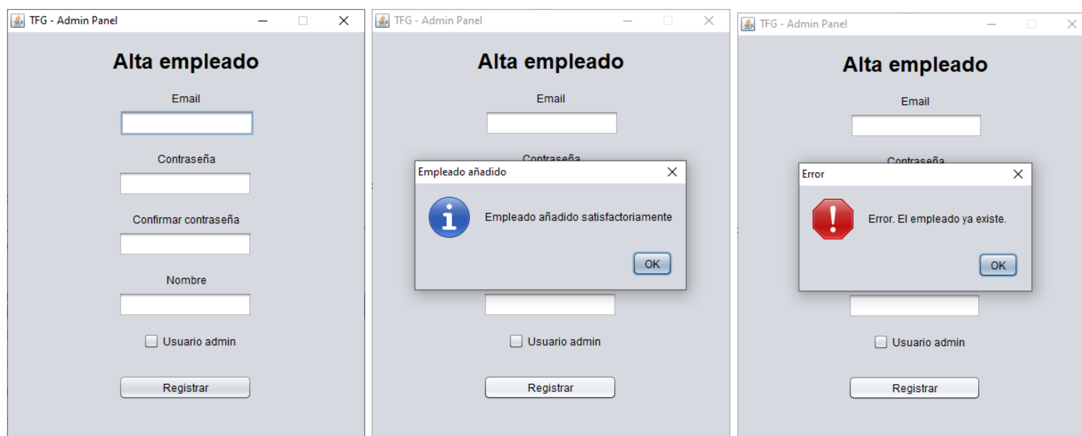


Figura 80. Menú de alta de empleado y posibles respuestas de la aplicación.

Si accedemos al menú de Baja de empleado, tenemos un menú en el cual se nos pide el email del empleado que queremos dar de baja. De nuevo, el sistema nos mostrará por pantalla un mensaje con la respuesta del servidor:

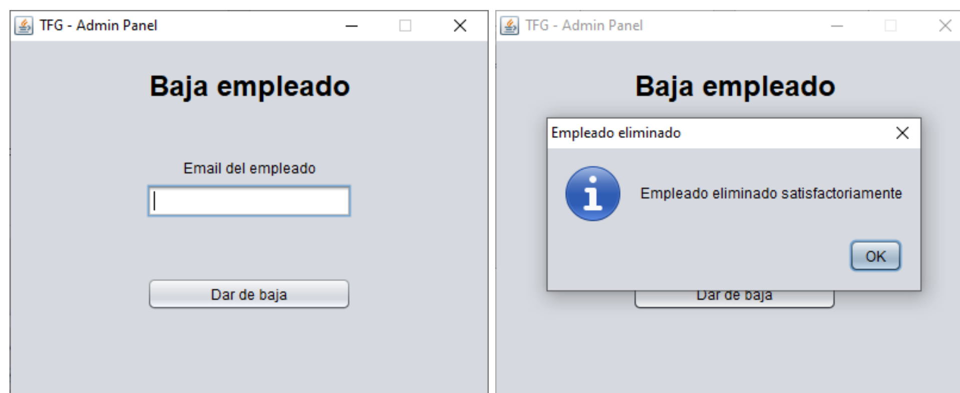


Figura 81. Menú de baja de empleado y respuesta correcta del servidor.

Por último, tenemos el menú de Consultar empleado. Este menú tiene la funcionalidad de recuperar información sobre empleados en base a su nombre. Está pensado para recuperar el email u otros datos de un empleado cuando el administrador no se acuerde de su email y solo tenga el nombre (o un nombre parcial).

El menú cuenta con un campo de búsqueda en el cual se nos pide el nombre y una tabla. La tabla será rellenada con los datos recuperados del servidor:

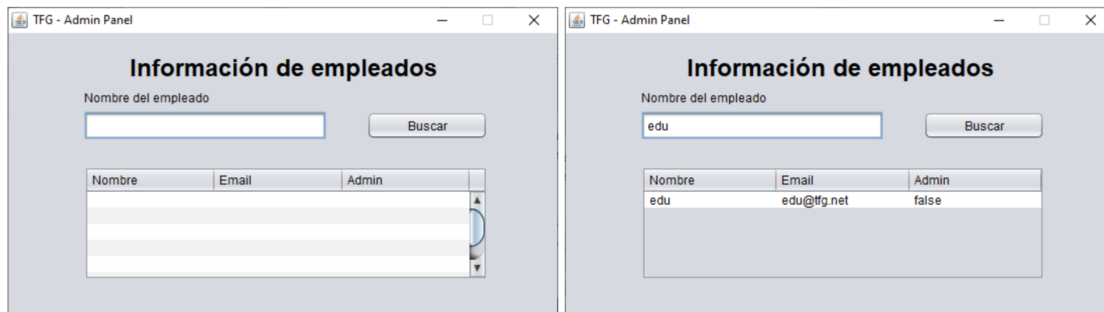


Figura 82. Menú de información de usuario y respuesta de la aplicación.

Pasándo al menú de gestión de horarios, volvemos a tener tres posibilidades. La primera de ellas es la de añadir nuevos horarios.

El menú para añadir nuevos horarios nos pide el email del empleado sobre el que queremos crear el horario, y nos presenta con una serie de menus de elección para escoger tanto la fecha exacta del registro horario, como las horas de entradas y salida que tendrá el registro para ese día:

Figura 83. Menú de creación de horarios.

Cuando pulsemos el botón para crear, la aplicación nos mostrará un mensaje con la respuesta del servidor:

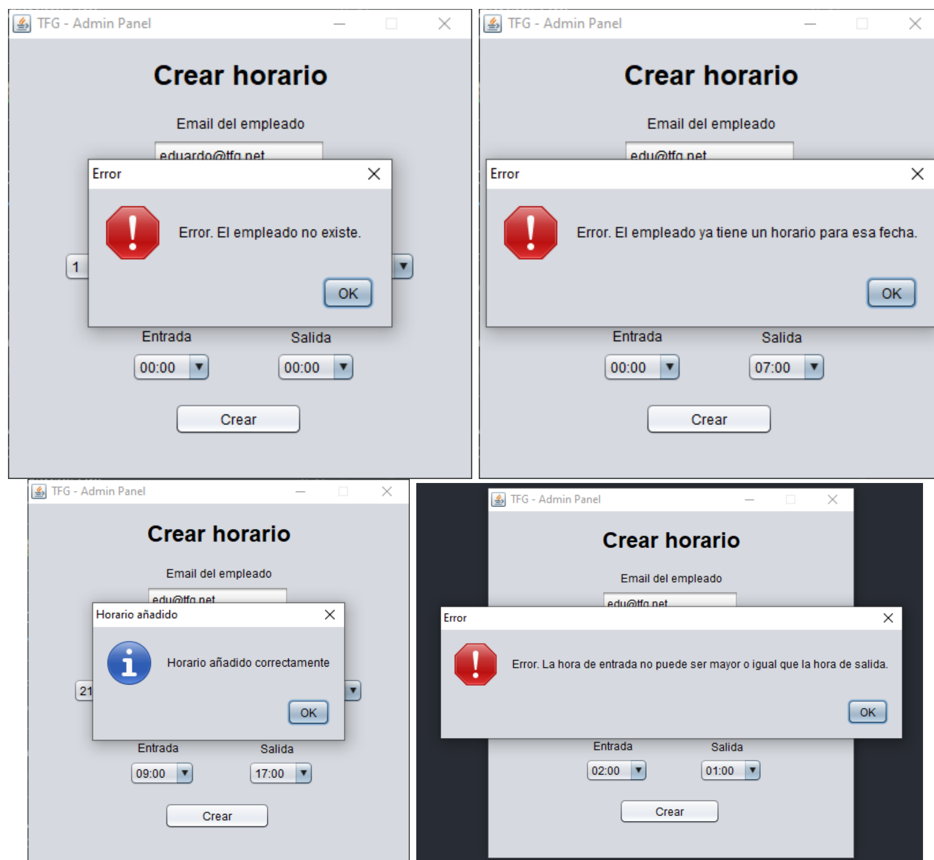


Figura 84. Posibles respuestas del servidor ante la creación de un nuevo horario para un empleado.

La segunda funcionalidad relacionada con los horarios es la de eliminar un horario. En este caso, el menú nos pedirá el email del empleado sobre el que se quiera borrar el horario, y la fecha del horario que se quiere borrar. De nuevo, la aplicación nos enseñará la respuesta del servidor a través de un popup:

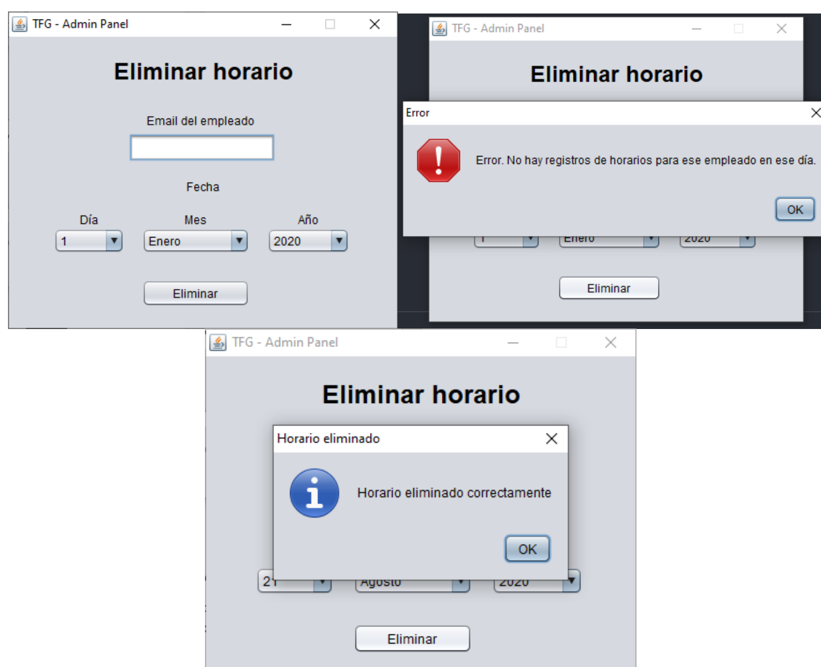


Figura 85. Menú de eliminación de un horario para un empleado.

Por último, tenemos la funcionalidad de consultar horarios. En esta ventana, se nos pide el correo electrónico del empleado al que estamos consultando, y tendremos que escoger el mes y año sobre el que estamos buscando los registros horarios. De esta forma, se reducen los resultados y se hace que todo sea más visible.

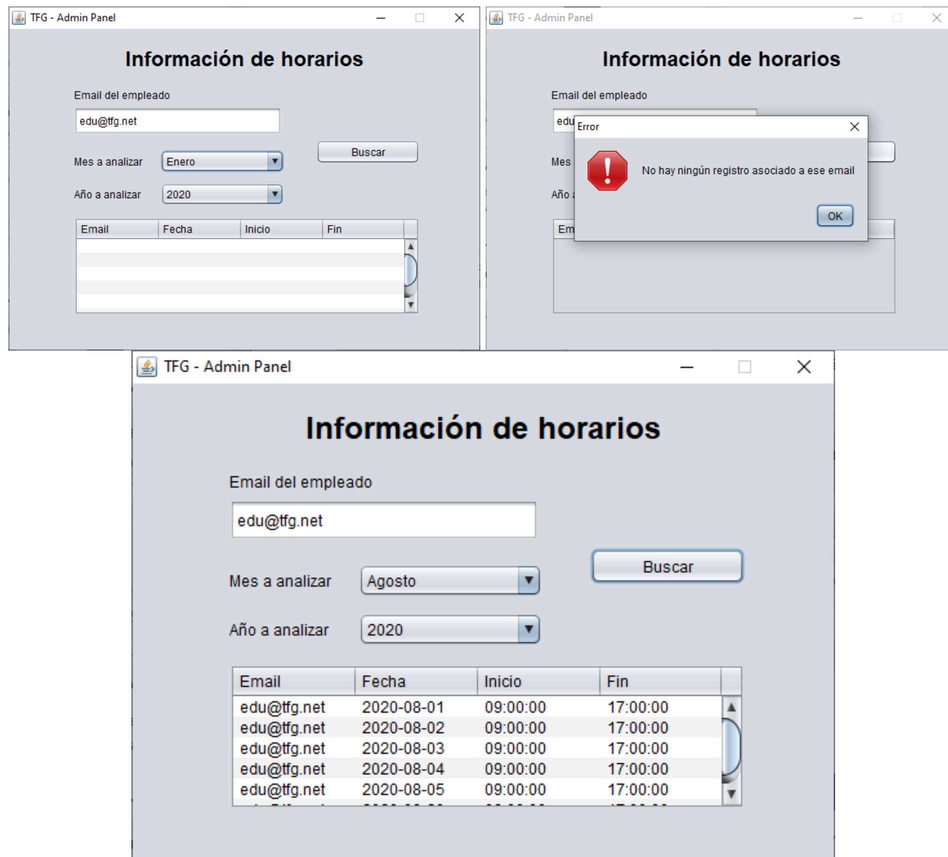


Figura 86. Menú de consulta de información de horarios de empleados.

Nos quedan las funcionalidades referentes a los registros de asistencia de los empleados. La primera funcionalidad nos da la posibilidad de consultar la asistencia de los empleados. Este menú es muy parecido al de consulta de información de horarios.

Se nos pide introducir el email y el mes y año sobre el que consultar. Una vez hemos introducido estos datos, se nos recupera toda la información de asistencia para ese mes del empleado cuyo email hemos introducido.

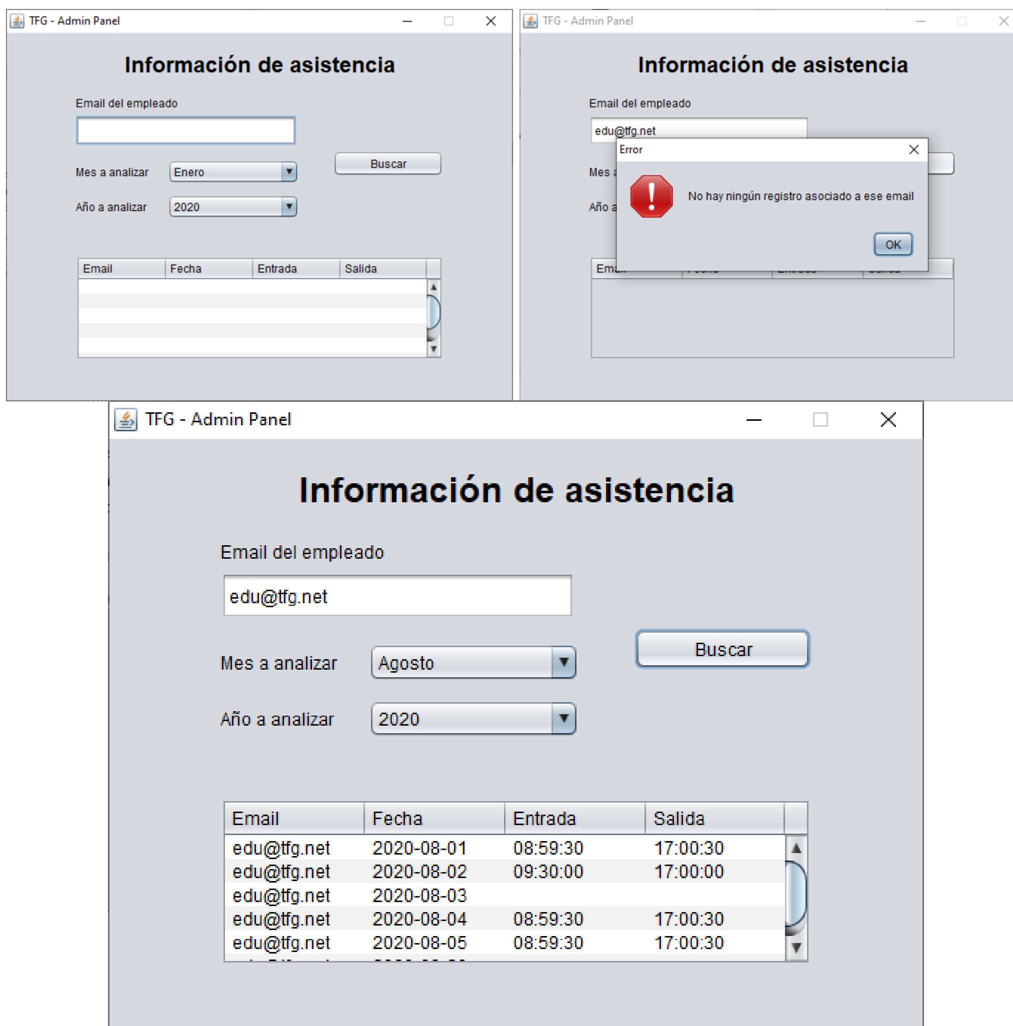


Figura 87. Menú de comprobación de asistencia de un empleado.

En la tabla, los campos de entrada/salida que estén en blanco, indican que ese el día el empleado no fichó, es decir, son faltas de asistencia.

La segunda funcionalidad es la de generar informes en base a la asistencia de un empleado. De nuevo, se nos pide el email, el mes y el año sobre el que generar el informe. En caso de que hayamos escogido el mes actual, la aplicación nos informa de que el informe puede no ser 100% representativo en cuanto a la relación de horas asignadas/horas trabajadas:

Figura 88. Menú de análisis de horas trabajas por un empleado.

Figura 89. Posibles respuestas después de rellenar el formulario de análisis de horas de empleados.

TFG - Admin Panel

Análisis de horas

Email del empleado

edu@tfg.net

Mes a analizar Junio Buscar

Año a analizar 2020

Total de horas asignadas 184

Total de horas trabajadas 163

Faltas en el mes 3

Horas extra 0

Horas que debe 21

Figura 90. Respuesta de la consulta de horas trabajadas de un empleado en un mes con faltas.

TFG - Admin Panel

Análisis de horas

Email del empleado

edu@tfg.net

Mes a analizar Julio Buscar

Año a analizar 2020

Total de horas asignadas 184

Total de horas trabajadas 187

Faltas en el mes 0

Horas extra 3

Horas que debe 0

Figura 91. Respuesta de la consulta de horas trabajadas de un empleado en un mes con horas extra.

10. Anexo B – Elementos adicionales entregables

Este anexo recoge una descripción de los elementos que no son entregables directamente con el proyecto, pero que se apreciaría que se tuviesen en cuenta a la hora de evaluar el proyecto.

Para entregar estos elementos, se ha creado un repositorio de GitHub público que se puede encontrar en el siguiente enlace: https://github.com/Tuskk15/TFG

El repositorio de GitHub ha sido utilizado como forma de almacenar las versiones en una plataforma online a modo de backup y control de versiones.

Los contenidos que se podrán encontrar en este repositorio son:

- Código fuente de la aplicación de escritorio para administradores. La estructura de este directorio sigue la estructura básica de un proyecto Maven creado con Apache Netbeans.

- Código fuente de la aplicación Android desarrollada. La estructura de este directorio sigue la estructura básica de un proyecto Gradle creado con IntelliJ Idea (en su defecto, se puede usar Android Studio perfectamente para abrirlo).

- Código fuente del servidor HTTP, contando con todas las clases Python que implementan la lógica de la aplicación, el fichero YAML desplegable, etc.

- Dentro del directorio del servidor HTTP, se encuentra también la carpeta de la base de datos SQLite. Se entrega una versión con la base de datos inicializada y cargada con datos, así como los scripts SQL necesarios para crear las tablas y poblarlas con datos de prueba. Para automatizar la ejecución de estos scripts SQL, se ha creado un script en formato .bat.

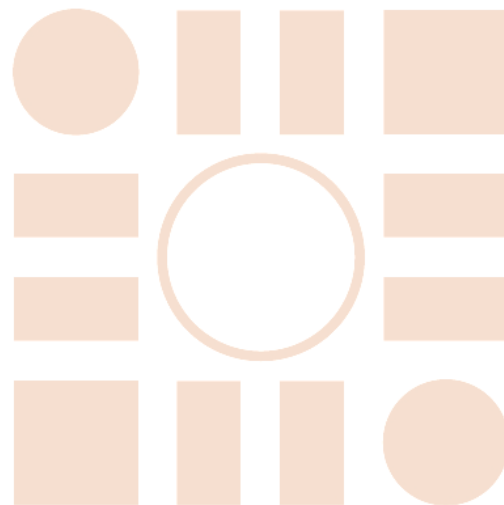
- Directorio de documentación. En este directorio se podrá encontrar la memoria entregada, la presentación PowerPoint utilizada en la defensa, y una carpeta que guarda los docs de las aplicaciones creadas. Estos docs son Javadoc en formato HTML en el caso del proyecto de Android y la aplicación de escritorio, y un sucedáneo de Javadoc para Python3 conocido como Sphinx. Los docs recogen la documentación de todos los métodos y clases creadas para el proyecto.

Si se quisiese acceder a los Javadocs, simplemente tenemos que descargar el repositorio, entrar en la carpeta que almacena los docs y abrir el archivo *index.html* con del proyecto que queramos consultar con el navegador. Una vez lo hayamos abierto, podemos navegar por toda la documentación de ese proyecto internamente.

En el caso de querer probar las aplicaciones, es importante saber que los valores de las IPs del servidor HTTP están “hardcodeadas” para conectarse al servidor alojado en el PC de la red local de mi casa, por lo que se tendría que, o bien cambiar todas las IPs de los ficheros de código antes

de ejecutarlos, o modificar el router para que asigne la misma IP utilizada durante pruebas; esta es: “192.168.1.136”.

Universidad de Alcalá
Escuela Politécnica Superior



ESCUELA POLITECNICA
SUPERIOR