

Escuela Politécnica Superior

Universidad de Alcalá

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

Trabajo Fin de Grado

Grado en Ingeniería Informática - Curso 2019/2020

Eduardo Graván Serrano – 03212337L

índice

[Tecnología NFC 3](#_Toc48928442)

[Introducción 3](#_Toc48928443)

[NDEF 4](#_Toc48928444)

[Type 4 tags 5](#_Toc48928445)

[Sistema desarrollado 6](#_Toc48928446)

[Introducción 6](#_Toc48928447)

[Requisitos del sistema 6](#_Toc48928448)

[Arquitectura del sistema 7](#_Toc48928449)

[Base de datos 7](#_Toc48928450)

[ReST API y servidor HTTP 9](#_Toc48928451)

[Aplicación Android 15](#_Toc48928452)

[Login, información de usuario y conexión con el servidor HTTP 16](#_Toc48928453)

[Actividad y servicio de emulación de etiquetas 19](#_Toc48928454)

[Actividad de lectura de etiquetas 25](#_Toc48928455)

[Aplicación de escritorio para administradores 29](#_Toc48928456)

[Manual de usuario 32](#_Toc48928457)

[Aplicación Android 32](#_Toc48928458)

[Empleado 33](#_Toc48928459)

[Usuario administrador 33](#_Toc48928460)

[Panel de Administración 33](#_Toc48928461)

[Bibliografía 33](#_Toc48928462)

# Tecnología NFC

## Introducción

El término **NFC** (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.

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 tarjetas 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.

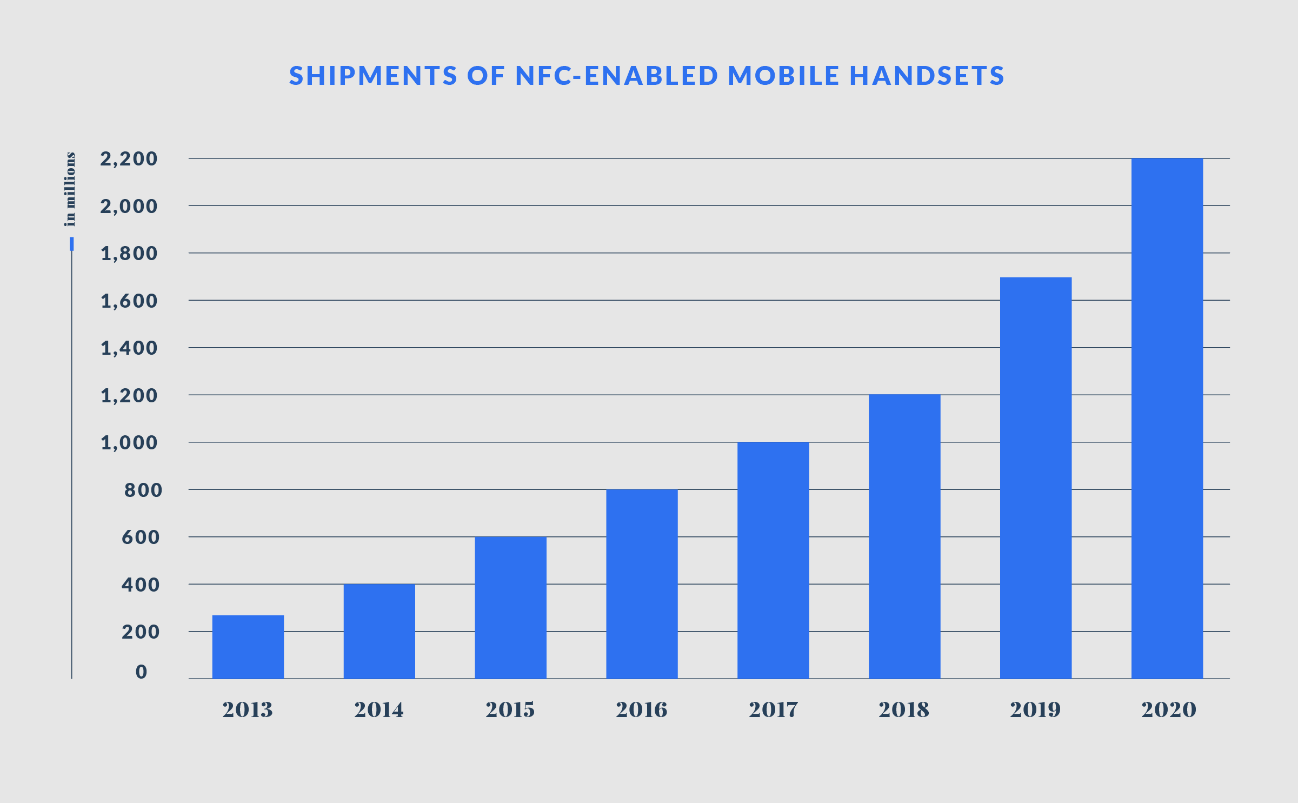
La aplicación a desarrollar contará con los dos primeros modos de operación descritos.

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.

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.

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).

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:



Shipments of NFC-enabled mobile handsets. BlueBite.

Las principales aplicaciones de la tecnología NFC son las 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 tarjetas, pudiendo reemplazar otros sistemas como contraseñas, pines, etc.
* **Seguimiento e identificación de productos:** se le pueden asignar tarjetas 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 tarjetas identificadoras para cada usuario de cierto sistema, pudiendo crear aplicaciones para el control de asistencia.

El proyecto se centrará en este último campo de aplicación de la tecnología.

## NDEF

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 tarjeta 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… y el propio “payload” o “mensaje” que se quiere servir a través de la comunicación.

Hay cuatro tipos de tarjeta NFC estandarizadas que pueden implementar el formato NDEF para compartir datos. Los estándares para estos tipos de tarjetas están definidos por el foro NFC.

## Type 4 tags

Para el desarrollo de la aplicación se ha escogido implementar un emulador de tarjetas de tipo 4 basado en la especificación “NFC Forum Type 4 Tag Operation Specification 2.0”.

Una tarjeta de tipo 4 basada en esta especificación debe contener una aplicación de tarjeta 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 tarjeta, e información sobre el resto de los archivos en la tarjeta 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 tarjeta o emulador de tarjetas de tipo 4 totalmente operativo, se puede iniciar la comunicación entre la tarjeta y un dispositivo de lectura. Para poder captar las etiquetas de tipo 4, el lector debe estar configurado para buscar tarjetas a través del protocolo NFC-A (basado en ISO/IEC 14443A).

La comunicación entre la tarjeta y el lector se hace a través de “application protocol data units” (APDUs). El lector de tarjetas estará sondeando para buscar tarjetas NFC-A a su alrededor, una vez encuentre una empezará una serie de mensajes para descubrir qué tipo de tarjeta es. El que nos interesa es el protocolo de comunicación entre el lector NFC-A y nuestra tarjeta NFC de tipo 4 con NDEF. En él, el lector mandará una serie de C-APDUs (Command APDUs), a los cuales la tarjeta deberá responder con R-APDUs (Response APDUs).

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

1. 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 tarjetas NFC con mensajes NDEF.

Tarjeta manda R-APDU: el R-APDU confirma si la lectura es correcta o si la tarjeta rechaza la conexión.

1. Lector manda C-APDU: **Capability Container (CC) Select**. El lector le pide “permiso” a la tarjeta para acceder a su fichero CC.

Tarjeta manda R-APDU: el R-APDU confirma si la lectura es correcta o si la tarjeta rechaza la conexión.

1. Lector manda C-APDU: **Capability Container Read**. El lector pide leer el archivo CC de la tarjeta.

Tarjeta manda R-APDU: el R-APDU contiene el archivo CC.

1. Lector manda C-APDU: **NDEF Select**. El lector le pide “permiso” a la tarjeta para acceder a su archivo NDEF.

Tarjeta manda R-APDU: el R-APDU confirma si la lectura es correcta o si la tarjeta rechaza la conexión.

1. Lector manda C-APDU: **NDEF Read**. El lector le pide a la tarjeta el archivo NDEF.

Tarjeta 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 tarjeta 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 tarjetas y por lo tanto la comunicación ha terminado correctamente.

# Sistema desarrollado

## Introducción

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. Haciendo uso de esta tecnología, 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.

Con el objetivo de hacer un control de las horas trabajadas de cada empleado se ha desarrollado una aplicación de escritorio para acceder a todos los datos de la base de datos a través de una cómoda interfaz gráfica.

Por último, se ha creado un servidor HTTP que hace de intermediario entre el resto del sistema y la base de datos. Este servidor HTTP implementa una API ReST, proporcionando una interfaz de acceso a la base de datos al resto de subsistemas que forman parte del proyecto.

La base de datos guarda la información sobre empleados, horarios de empleados, y los registros de asistencia de todos los días de trabajo de cada empleado.

### Requisitos del sistema

Para el correcto funcionamiento del sistema se necesitan los siguientes componentes:

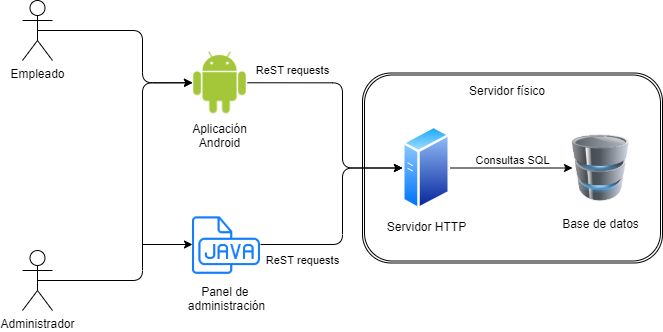
* Servidor con Python3 instalado con los módulos necesarios para el funcionamiento del servidor (Módulo de Flask y soporte de Swagger 2.0).
* Base de datos SQLite3 instalado en el servidor HTTP.
* Ordenador con Java instalado para el uso de la aplicación de administración.
* Al menos dos dispositivos 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 19 (Android 4.4 - KITKAT).

## Arquitectura del sistema

El sistema desarrollado cuenta, a modo resumen, con los siguientes subsistemas:

* **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.
* **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.

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



## Base de datos

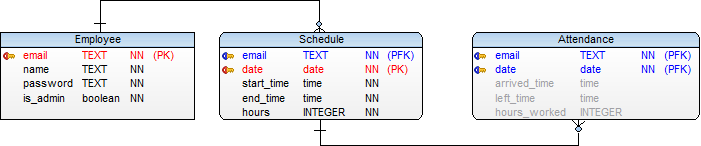
Para poder gestionar los datos referentes al control de asistencia de una organización, se ha creado el siguiente modelo de base de datos basado en tres tablas:

* **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.

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

Se presenta el siguiente diagrama para tener una visión más clara de la base de datos:



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.

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.

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.

## ReST API y 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.

El servidor HTTP está implementando en **Python 3** haciendo uso del framework **Flask** para crear el servidor HTTP. 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.

De cara a implementar la API ReST, se ha decidido seguir la especificación de la **OpenAPI 2.0**, 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 password=\'{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]}\' WHERE 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'], timestamp[0])

     cursor.execute(f"UPDATE Attendance SET left\_time=\'{timestamp[0]}\', hours\_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:

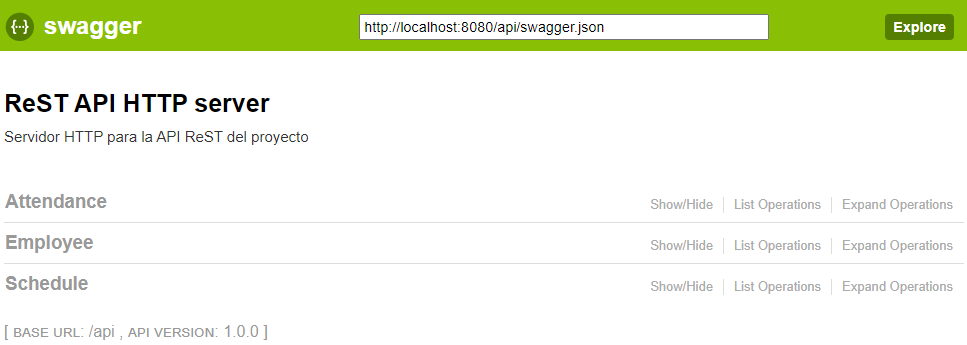
*return* make\_response(response\_str, 200)

*else*:

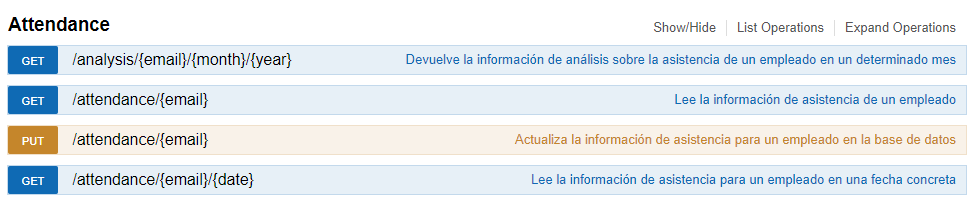
*return* make\_response(response\_str, 400)

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.

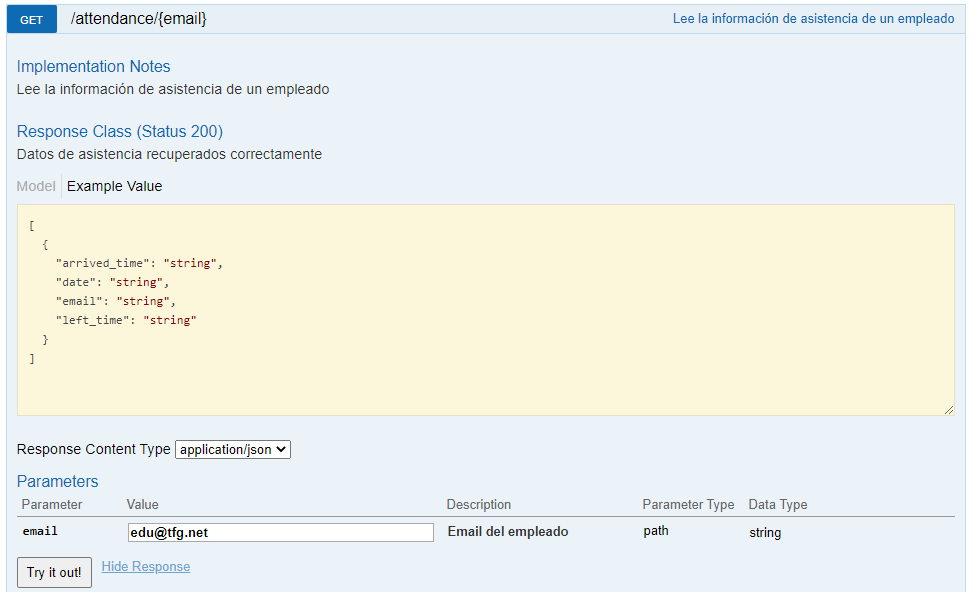
Para acceder a ella simplemente tenemos que abrir un navegador, acceder a la IP y puerto del servidor y dirigirnos a “/api/ui/”:

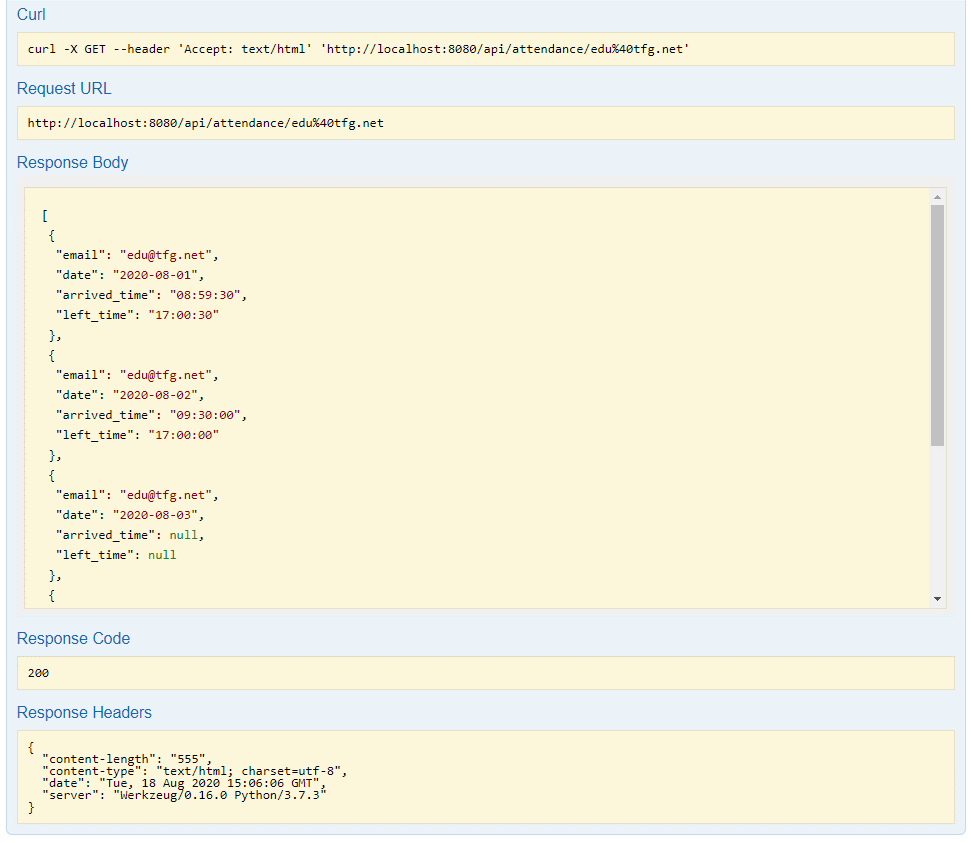


Si accedemos a Attendance, se nos muestra todas las llamadas ReST registradas bajo la etiqueta Attendance:

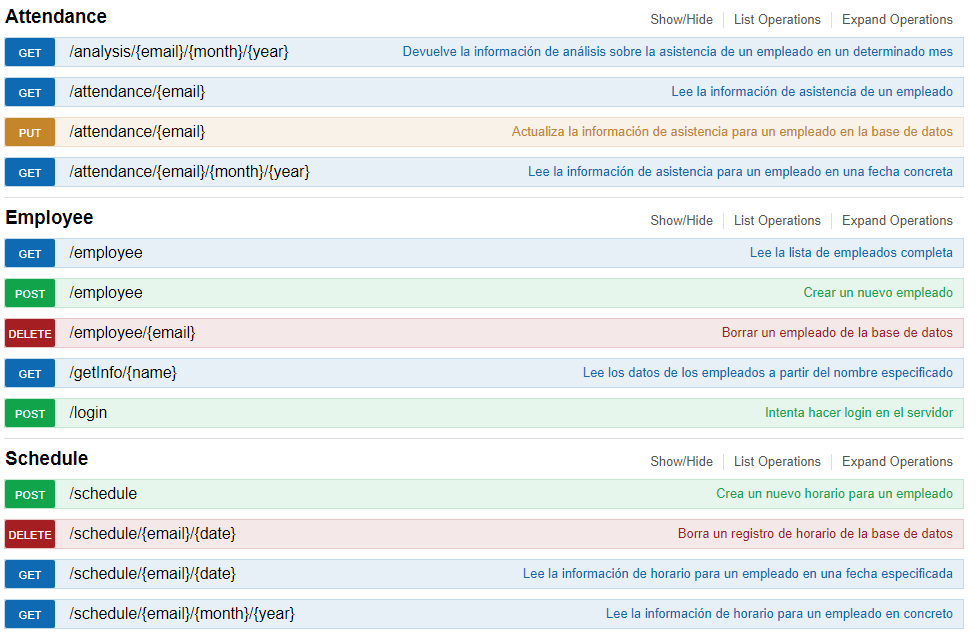


Si hacemos click en cualquiera de ellas, se nos muestra toda su información y se nos permite hacer una llamada y comprobar el resultado de esta:





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



## 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.

### 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.

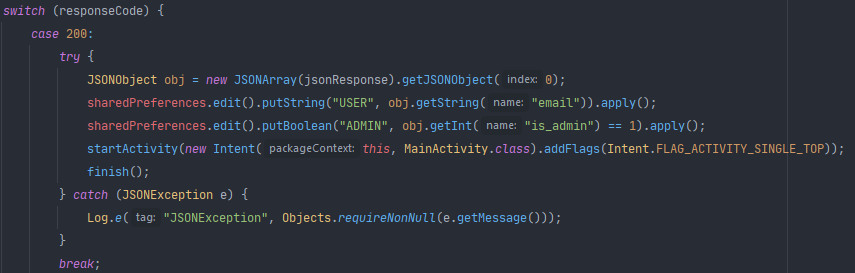
El almacenamiento de esta información se hace a través de la interfaz de Android introducida en la API 1 llamada “SharedPreferences”. Esta interfaz nos permite almacenar variables en la memoria interna de la aplicación, permitiendo el acceso desde cualquier punto, denegando al mismo tiempo el acceso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



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):



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:

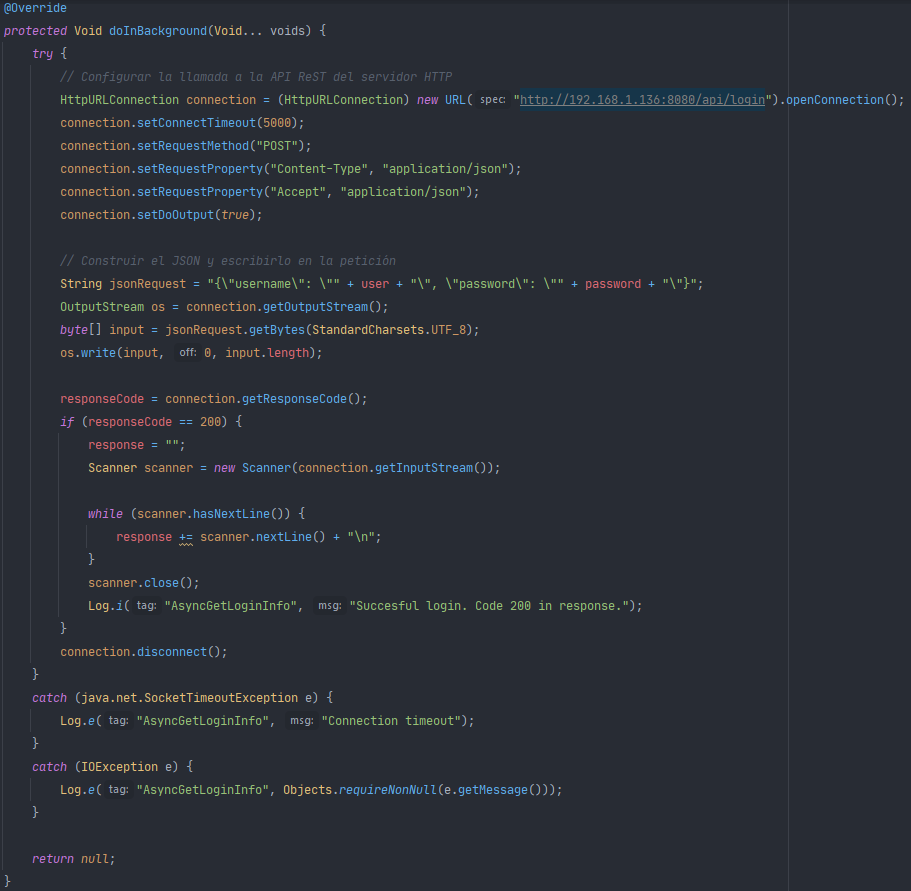


Además, Android no permite hacer llamadas bloqueantes a servidores remotos desde el hilo principal, ya que este hilo es el encargado de gestionar la interfaz gráfica. Para poder conectarse con el servidor HTTP se tienen que crear clases que extiendan a la clase “AsyncTask”. La clase AsyncTask permite ejecutar tareas de fondo, devolviéndole la información al hilo principal una vez haya terminado de ejecutar la tarea.

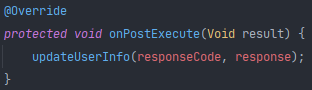
Todas las actividades que necesitan conectarse al servidor HTTP tienen una clase interna privada que extiende a AsyncTask con el objetivo de gestionar estas llamadas. La clase que extiende a AsyncTask recibe todos los datos necesarios de la interfaz a través de su constructor y después se lanza el hilo.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



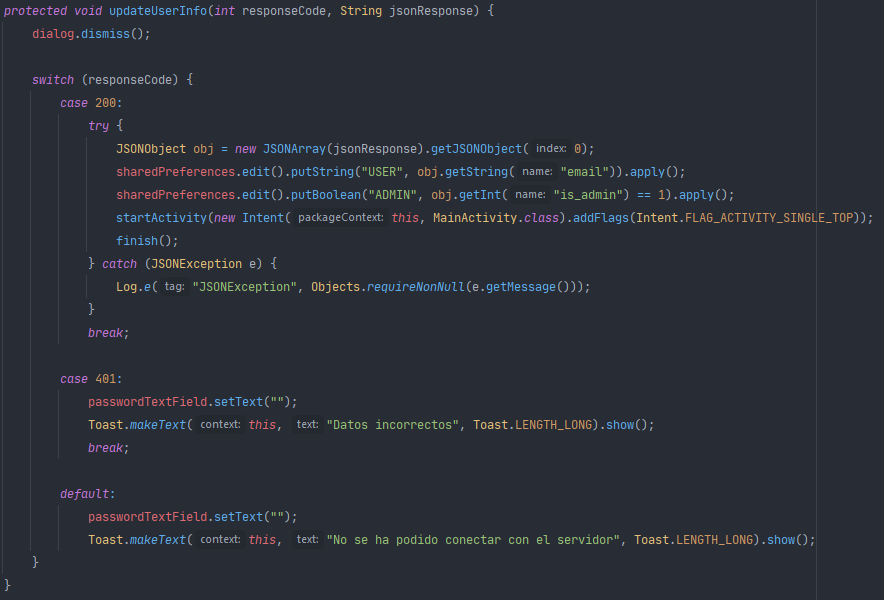
Al llamar a ejecutar una AsyncTask, internamente se llama al método “doInBackGround()”, que será el método que implementa toda la lógica de la comunicación con el servidor HTTP. En este método se prepara la conexión con el servidor HTTP configurando los parámetros de la conexión y se lanza la petición 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:



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:



Y la función “updateUserInfo”:



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.

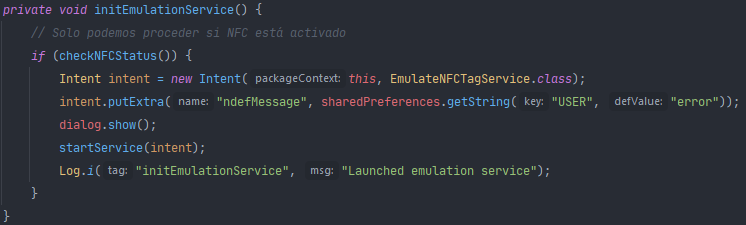
### Actividad y servicio de emulación de etiquetas

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

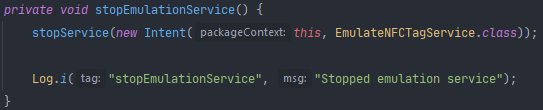
La actividad sigue el esquema general de actividades de Android. Al ser creada, se enlazan los elementos de la interfaz de usuario con variables interna de la clase. Se recupera la información de usuario de las SharedPreferences y se le deja saber al usuario el contenido de la etiqueta NFC a emular, 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:

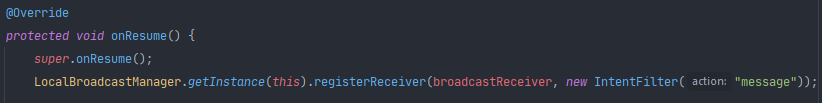


Asimismo, al crear el AlertDialog se registra un listener para que cuando se haga “dismiss” de cualquier forma, se pare el servicio de emulación a través de otro Intent. Esto está recogido en el método “stopEmulationService()”:

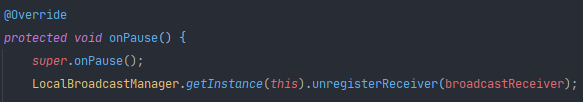


El último elemento importante que explicar de esta actividad es el atributo de tipo BroadcastReceiver de la clase. Un BroadcastReceiver es un objeto encargado de recoger broadcast lanzados en la aplicación como forma de comunicación entre diferentes componentes de la aplicación, en este caso, entre el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

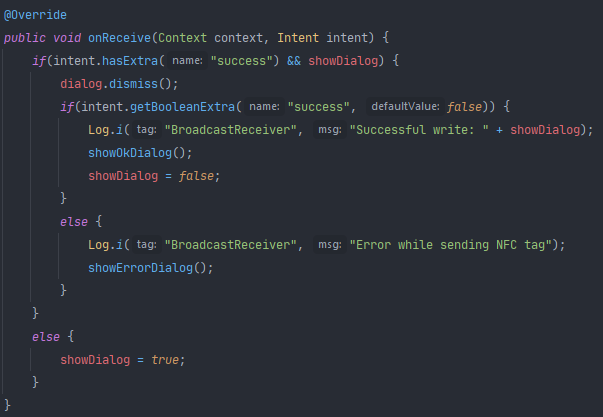
Cada vez que la actividad entra en primer plano se debe registrar el BroadcastReceiver:



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



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:



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.

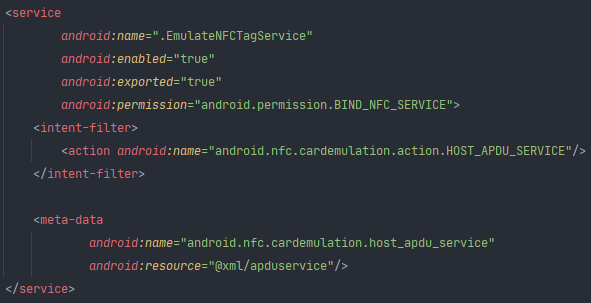
En primer lugar, para el funcionamiento tanto del emulador como del lector de tarjetas, se tiene que registrar la petición de permisos para acceder a la función de NFC en el manifiesto:



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):

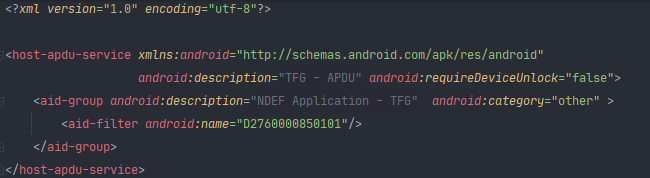


Por último, al declarar el servicio en el manifiesto debemos pedir el permiso “BIND\_NFC\_SERVICE”. Este permiso es necesario para que solo el sistema operativo Android sea capaz de enlazarse con el servicio de emulación de tarjetas. En la declaración de servicio hay que declarar un Intent-Filter.



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:

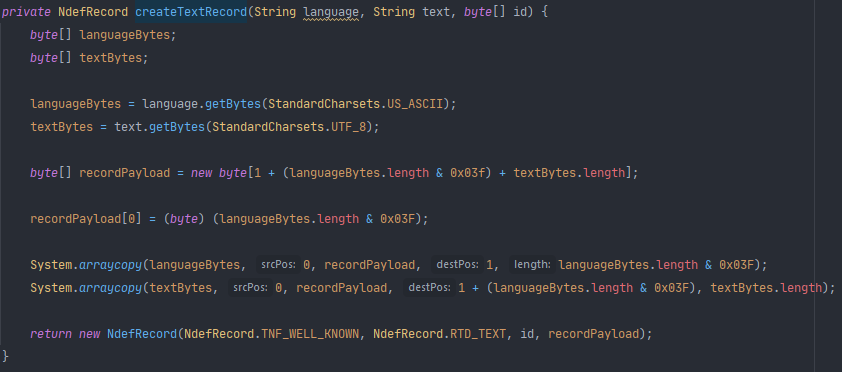


La clase Java que implementa el servicio de emulación de etiquetas debe heredar de la clase HostApduService, el cual es un servicio especial de Android para HCE.

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:



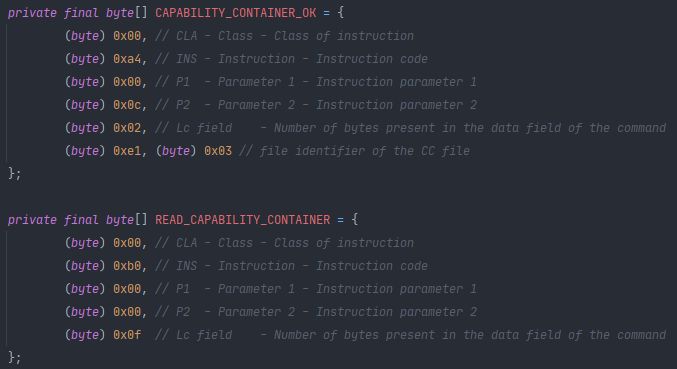
Para crear el mensaje NDEF, se debe crear un registro NDEF llamando al método auxiliar “createTextRecord”. Este método se encarga de codificar el mensaje añadiendo información sobre el idioma construyendo de esta forma la payload:



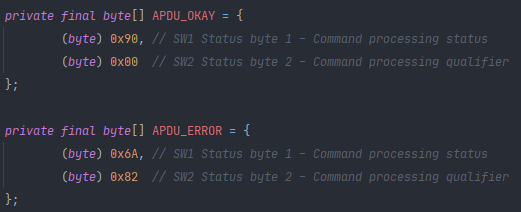
Una vez tenemos el mensaje NDEF, se convierte a un array de bytes y se calcula su longitud, rellenando el array de bytes de NLEN para que tenga un tamaño fijo de 2 posiciones.

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](#_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:



Y las respuestas:

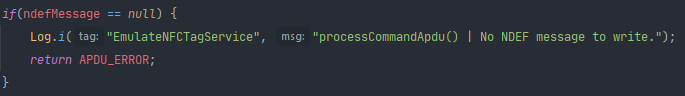


Estos arrays de bytes son los bytes especificados en el documento referente a la especificación de las etiquetas de tipo 4. El enlace a este documento se puede encontrar en la bibliografía.

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.

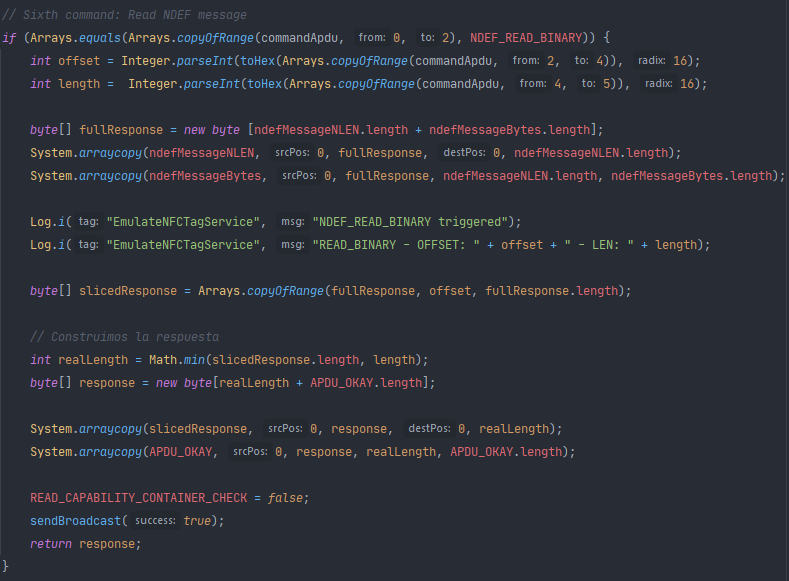
Es necesario hacer esta comprobación debido a que, como se ha explicado anteriormente, el servicio de emulación se podría haber iniciado si el teléfono ha registrado un Intent de NFC dirigido al AID especificado en el archivo XML. En caso de que esto haya ocurrido, el servicio no tiene la información 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.



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):

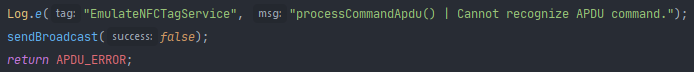


El lector seguirá respondiendo con C-APDU, hasta que al final se pida el mensaje NDEF. La siguiente captura corresponde a la construcción de la respuesta para conseguir este mensaje NDEF:

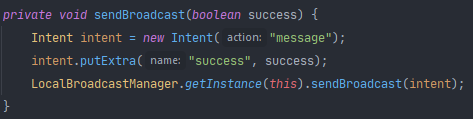


Como podemos comprobar, justo antes de enviar el R-APDU con el mensaje NDEF, se llama al método “sendBroadCast”, pasándole como parámetro un booleano a true.

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:



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:



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

### 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.

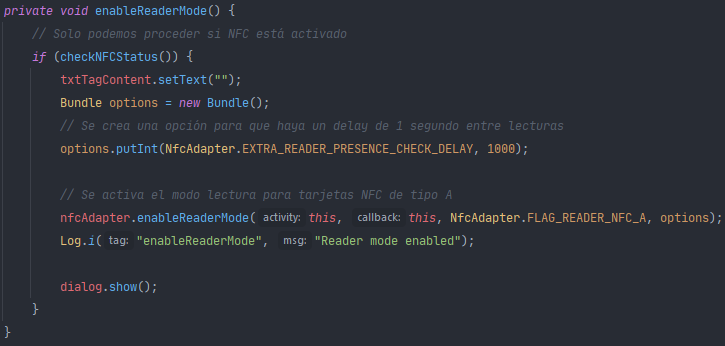
Cuando dos teléfonos se acercan entre sí, uno de ellos en modo lectura y el otro en modo emulación, aunque el segundo teléfono esté en modo emulación, el sondeo de búsqueda de etiquetas NFC cercanas continúa. El teléfono en modo emulación manda el Intent de NFC con el AID registrado en el Intent Filter, 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.

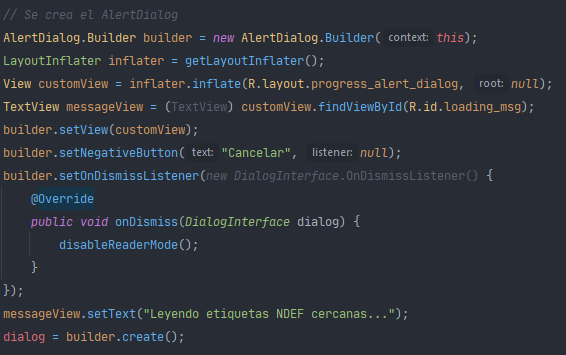
En un primer momento, como en el resto de las actividades Android, se enlazan los elementos de la interfaz con atributos de la clase y se registra un listener para el botón de lectura. El listener está encargado de llamar al método “enableReaderMode()”, cuyo código es el siguiente:



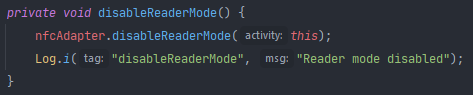
Este código se encarga de limpiar el TextView que ve el usuario en la interfaz gráfica. Después prepara la llamada para activar ReaderMode. Se activa un flag para evitar que haya lecturas consecutivas si no ha habido al menos un segundo entre lecturas, y se activa un segundo flag para que el modo lectura 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:



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

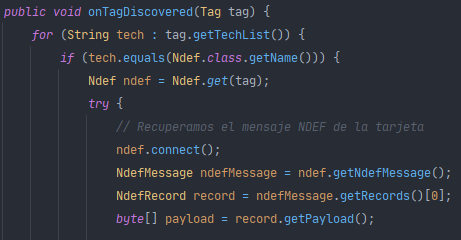


Recapitulando, cuando el usuario administrador haga click en el botón de lectura de etiquetas, se registrará el teléfono en modo lectura NFC y se creará un AlertDialog. Si se cierra este AlertDialog de cualquier forma, se desactivará el modo lectura.

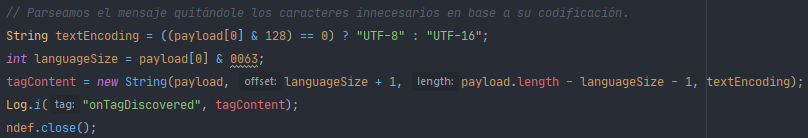
Cuando la actividad registre un Intent de NFC que coincida con el flag que hemos establecido anteriormente (NFC tipo A), se llamará al método de la interfaz “NfcAdapter.ReaderCallback”, “onTagDiscovered”.

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:

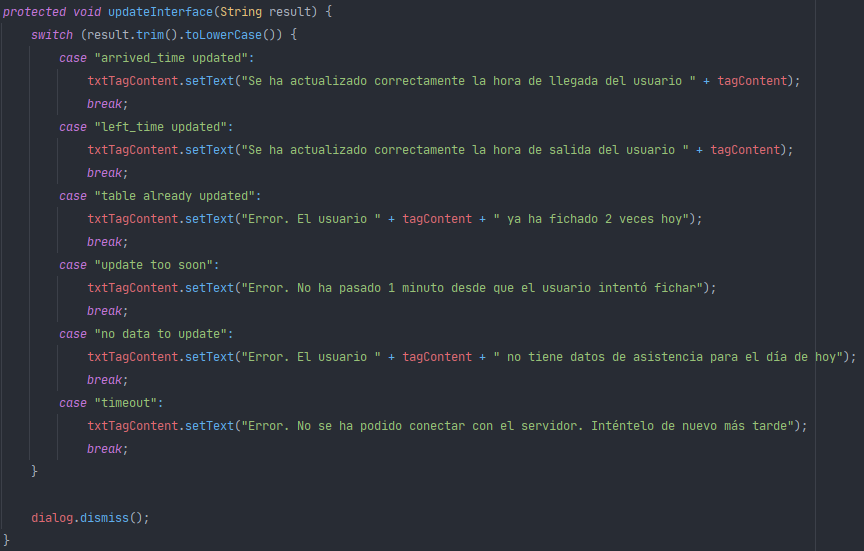


Una vez tenemos el contenido del mensaje NDEF, tenemos que parsear el mensaje quitando los bits de la codificación de texto y los indicadores de idioma del mensaje:



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:



## 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 “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:

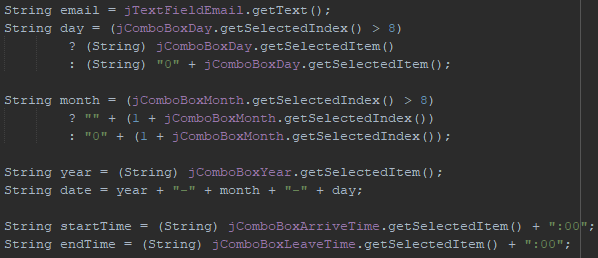
* **AttendanceInfo:** clase encargada de consultar los registros de asistencia de un empleado. Se pide el email del empleado, el mes y el año a buscar en la base de datos. Se presentan los datos recuperados en una tabla para su mejor visualización.
* **AttendanceHourAnalysis:** clase encargada de generar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

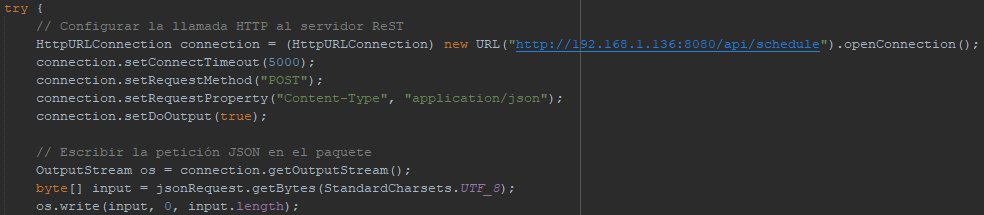
* **Login:** es la clase con la función Main y el punto de acceso a la aplicación. Se piden las credenciales del usuario y se lanzan contra la base de datos. En caso de que el usuario exista se analiza la respuesta para saber si el usuario es un administrador. Solo se le permitirá acceso a la aplicación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

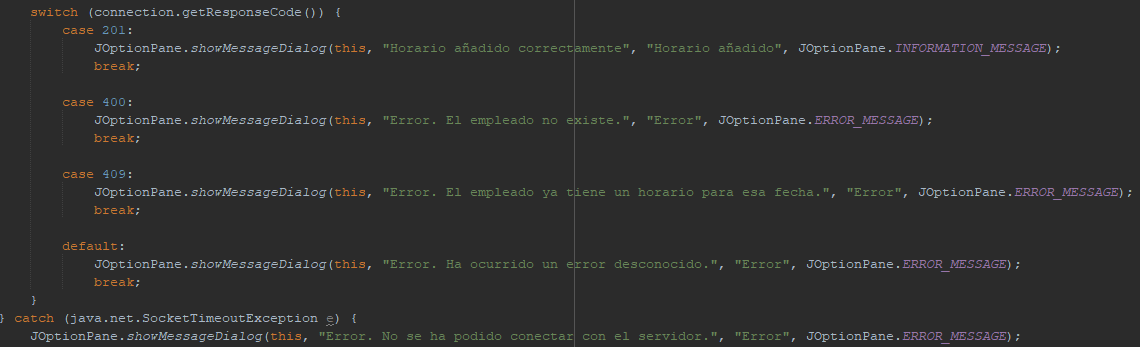
1. Una vez el usuario ha rellenado los campos que se piden para hacer la llamada a la API, el usuario pulsa un botón.
2. Se recogen los datos introducidos por el usuario de la interfaz y se procesan en busca de posibles errores o campos en blanco. En caso de que el campo necesite procesamiento, se hace ahí. (Ej: recogida de datos de cara a crear un registro horario):



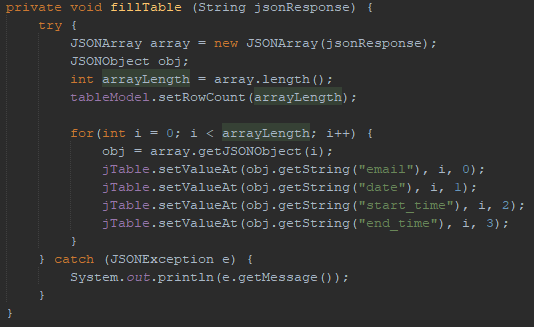
1. Si es necesario, se compila la información en un JSON. En cualquier caso, se hace la llamada al endpoint de la API correspondiente en función de lo que tengamos que hacer:



1. 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:



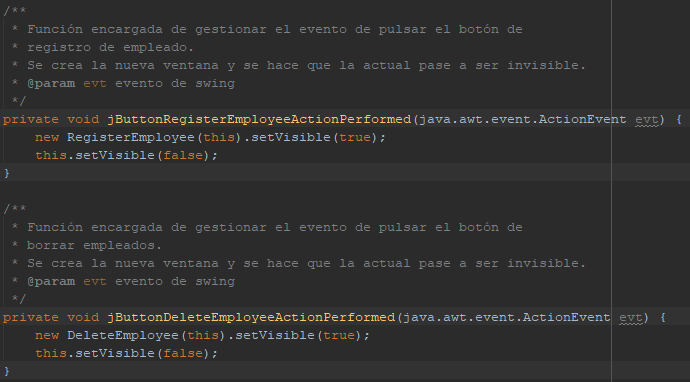
1. En caso de que se tenga que actualizar la interfaz porque es un método de consulta, se llama a una función auxiliar encargada de ello:



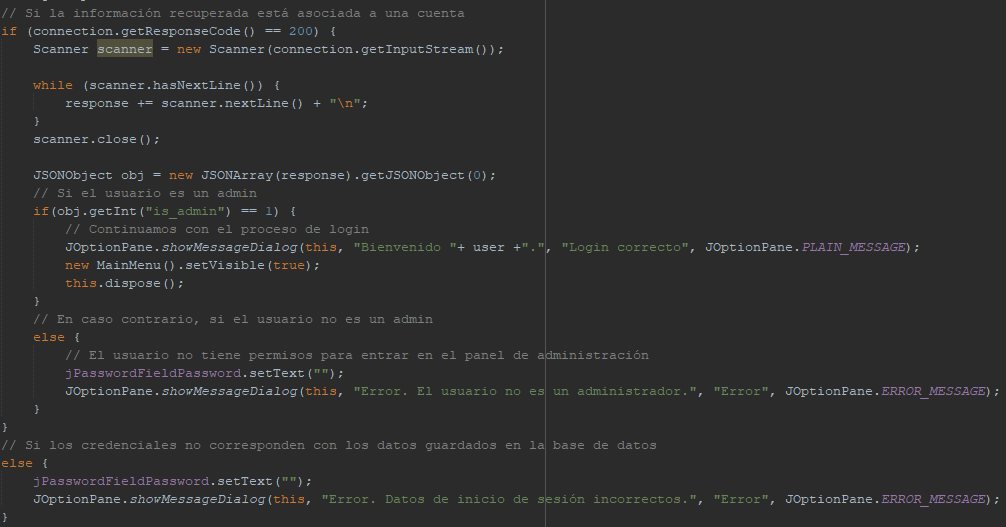
En cuanto a la clase Main, la clase implementa un “TabbedPane” para separar las funcionalidades de cada tabla. La lógica de la clase simplemente se encarga de establecer listeners para los eventos de pulsar los botones.

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:



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:



# Manual de usuario

Asdasd

## Aplicación Android

Asdasd

### Empleado

Asdasd

### Usuario administrador

Asdasd

## Panel de Administración

Asdasd

# Bibliografía

<https://en.wikipedia.org/wiki/Near-field_communication>

<https://www.bluebite.com/nfc>

<https://www.smart-tec.com/en/auto-id-world/nfc-technology>

<https://developer.android.com/guide/topics/connectivity/nfc>

<https://developer.android.com/reference/android/nfc/tech/Ndef>

<https://developer.android.com/reference/android/nfc/tech/NfcA>

NFC Forum: <https://nfc-forum.org/>

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

Diagramas hechos con la herramienta de draw.io: <https://app.diagrams.net/>