

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](#_Toc48574238)

[Introducción 3](#_Toc48574239)

[NDEF 4](#_Toc48574240)

[Type 4 tags 5](#_Toc48574241)

[Sistema desarrollado 6](#_Toc48574242)

[Introducción 6](#_Toc48574243)

[Arquitectura del sistema 6](#_Toc48574244)

[Base de datos 7](#_Toc48574245)

[ReST API y servidor HTTP 7](#_Toc48574246)

[Aplicación Android 7](#_Toc48574247)

[Aplicación de escritorio para administradores 7](#_Toc48574248)

[Bibliografía 7](#_Toc48574249)

# 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 tarjeta:** el dispositivo móvil emula la funcionalidad de una tarjeta 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 tarjetas 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.

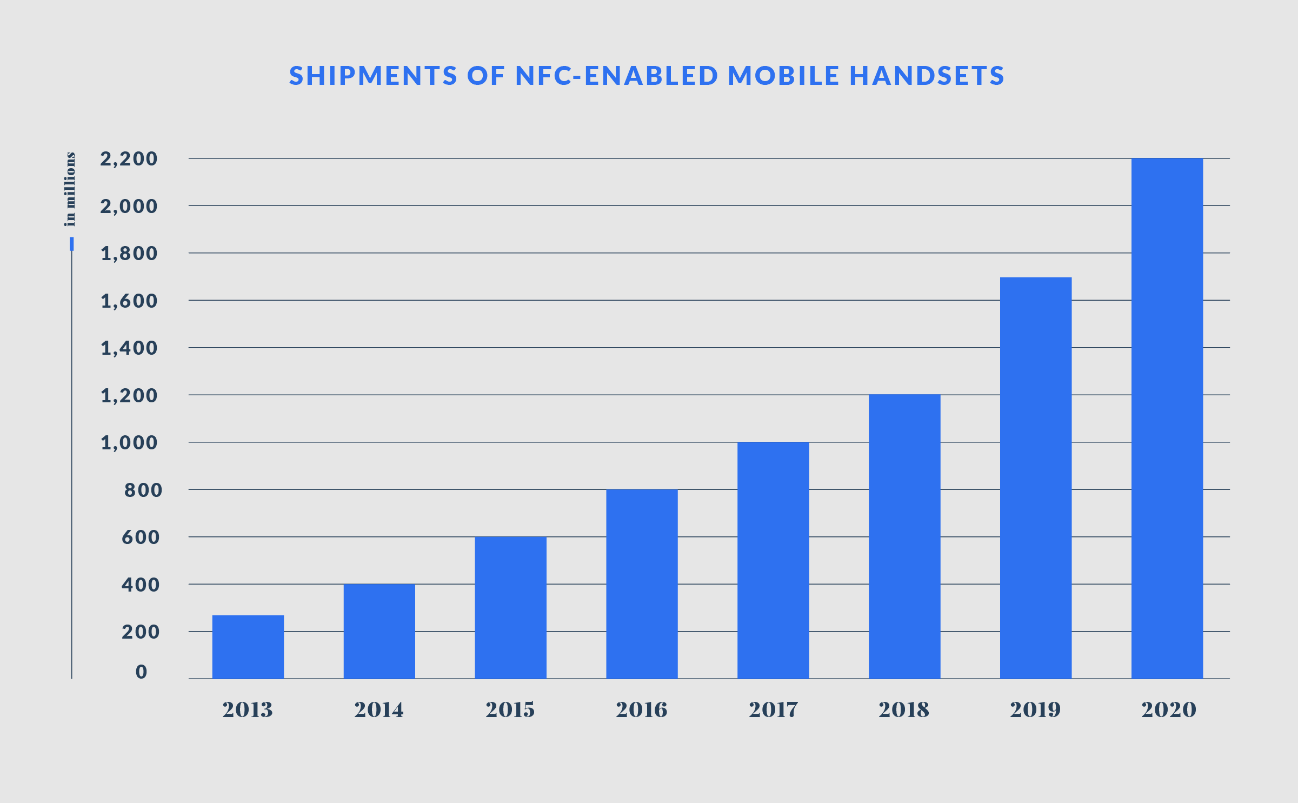
La aplicación a desarrollar contará con los dos primeros modos de operación descritos.

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 especifica 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.

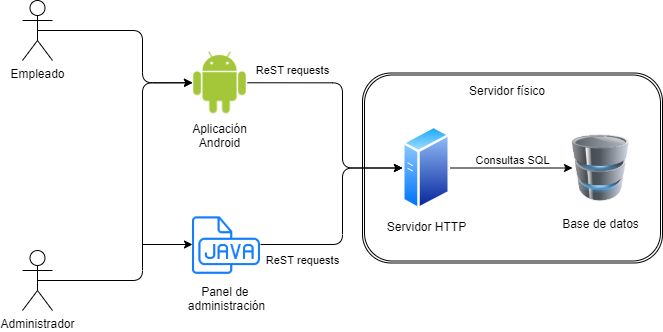
La base de datos guarda la información sobre empleados, horarios de empleados, y los registros de asistencia de todos los días de trabajo de cada empleado.

## 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

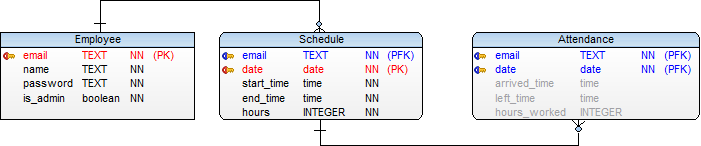
Para poder gestionar los datos referentes al control de asistencia de una organización, se ha creado el siguiente modelo de base de datos basado en tres tablas:

* **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:

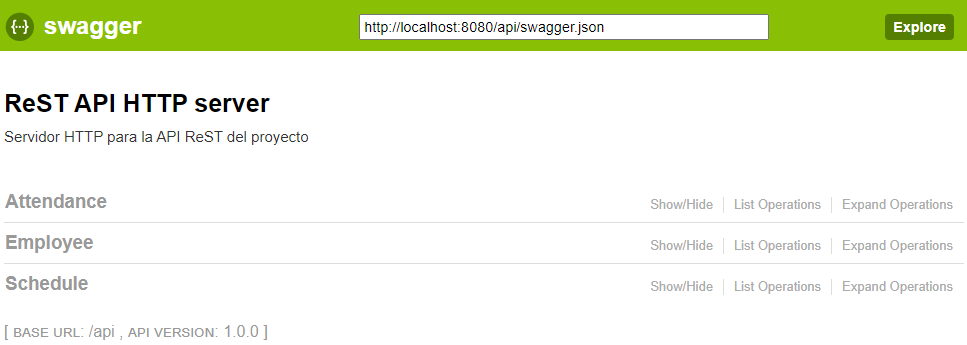
*return* make\_response(response\_str, 200)

*else*:

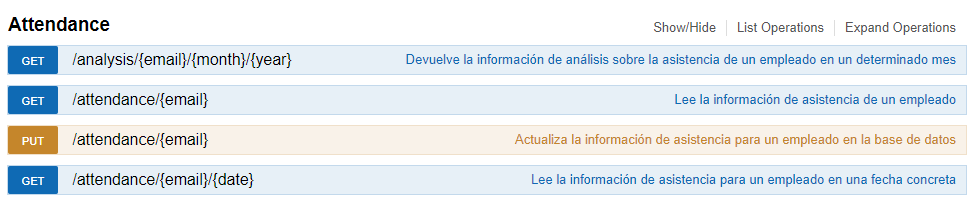
*return* make\_response(response\_str, 400)

El ultimo 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.

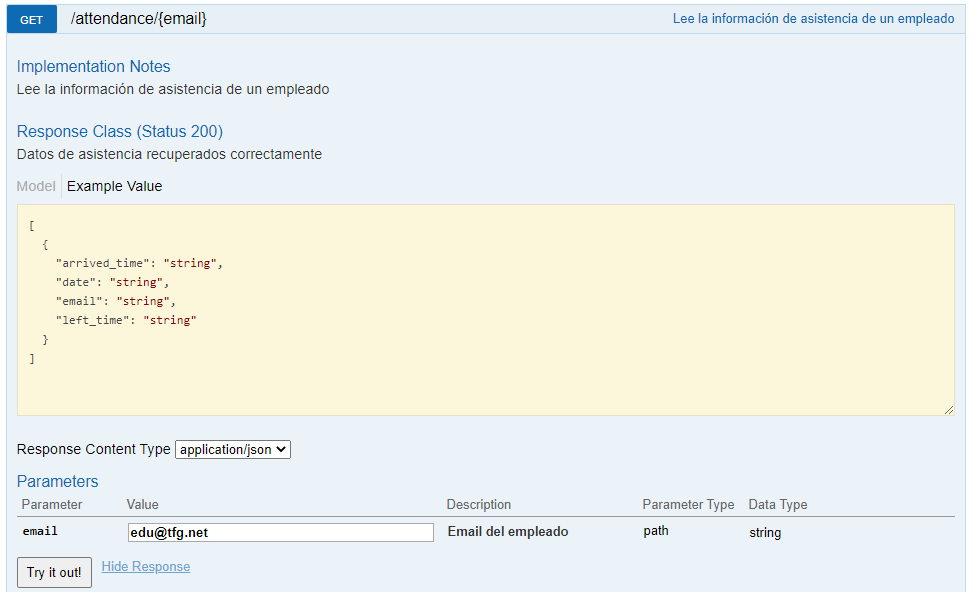
Para acceder a ella simplemente tenemos que abrir un navegador, acceder a la IP y puerto del servidor y dirigirnos a “/api/ui/”:

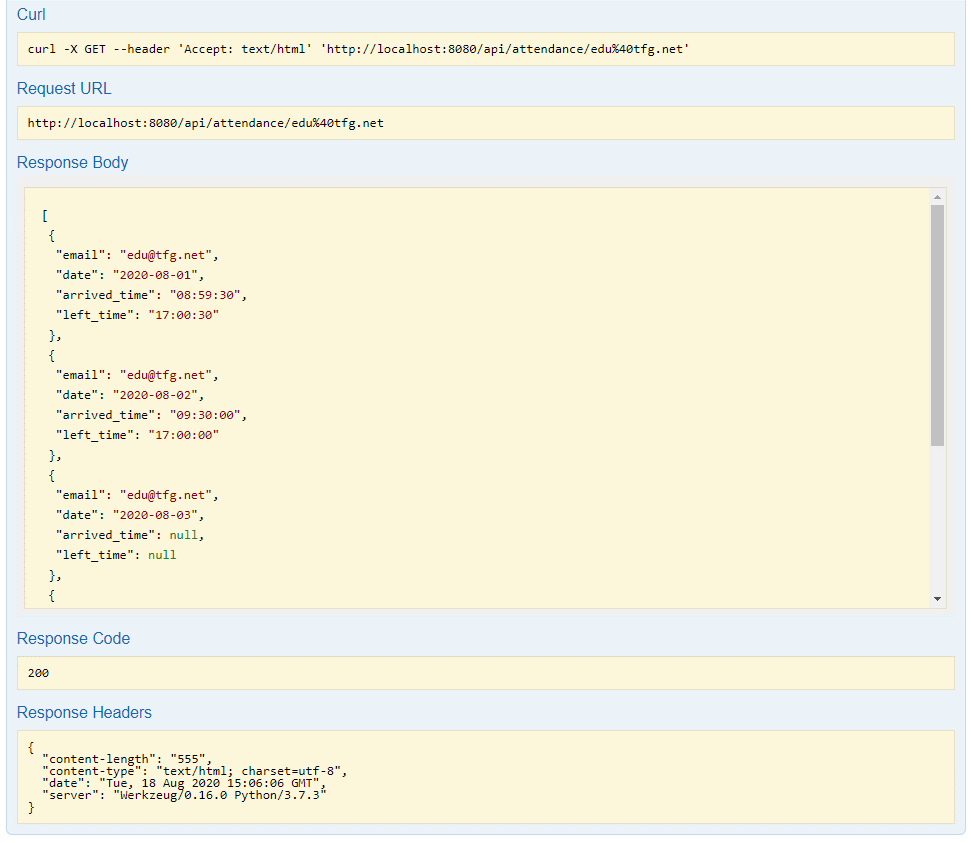


Si accedemos a Attendance, se nos muestra todas las llamadas ReST registradas bajo la etiqueta Attendance:



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

## Aplicación de escritorio para administradores

# 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/>