

ULL

Universidad
de La Laguna

Escuela Superior de
Ingeniería y Tecnología

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Información Personalizada en Dispositivos Móviles mediante el uso de Servicios Inteligentes

*Personalized Information on Mobile Devices through the use of
Intelligent Services*

Néstor Ibrahim Hernández Jorge

La Laguna, 15 de mayo de 2018

D. **Elena Sánchez Nielsen**, con N.I.F. 42.848.599-J profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y Sistemas de la Universidad de La Laguna, como tutor

CERTIFICA

Que la presente memoria titulada:

“Información Personalizada en Dispositivos Móviles mediante el uso de Sistemas Inteligentes”

ha sido realizada bajo su dirección por D. **Néstor Ibrahim Hernández Jorge**,
con N.I.F. 54.108.285-A.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 15 de mayo de 2018

Agradecimientos

En primer lugar agradecer a mi tutora Elena Sánchez Nielsen por sus valiosas indicaciones para la realización de este trabajo.

También agradecer a mi padre, madre, abuelos y demás familiares por el constante apoyo en esta etapa.

A mi pareja por su ayuda, su compañía especialmente en los peores momentos de este periodo, por su gran apoyo y comprensión, por su constante preocupación y principalmente por todo el tiempo que le robé durante el transcurso de este trabajo. Gracias.

Y por último a mis amigos que también se preocuparon y animaron en todo este tiempo.

Licencia



© Esta obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial 4.0 Internacional.

Resumen

El presente proyecto, “Información Personalizada en Dispositivos Móviles mediante el uso de Servicios Inteligentes”, pretende desarrollar un sistema de identificación inteligente para dispositivos móviles que permita a sus usuarios consultar su información personalizada gracias a su rostro o su voz, olvidándonos del inicio de sesión tradicional y ofreciendo así una alternativa más moderna, rápida y segura.

Para ello se parte de las APIs de Inteligencia Artificial “Face API” y “Speaker Recognition” de Microsoft, con las que se desarrollan dos servicios inteligentes que confeccionan un sistema que puede ser incluido en cualquier proyecto de aplicación móvil Android.

El sistema desarrollado permite a organizaciones o empresas disponer de una aplicación móvil donde sus integrantes acceden a su contenido personalizado de una forma novedosa y desde cualquier dispositivo, independientemente de cual sea el propietario de este.

Palabras clave: Identificación inteligente, inteligencia artificial, aplicaciones móviles, Android, API.

Abstract

The present project, "Personalized Information in Mobile Devices through the use of Intelligent Services", intends to develop an intelligent identification system for mobile devices that allows its users to consult their personalized information thanks to their face or voice, forgetting about the login traditional and thus offering a more modern, fast and secure alternative.

This is part of the Artificial Intelligence APIs "Face API" and "Speaker Recognition" from Microsoft, with which two intelligent services are developed that make a system that can be included in any Android mobile application project.

The system developed allows organizations or companies to have a mobile application where their members access their personalized content in a novel way and from any device, regardless of who owns it.

Keywords: *Intelligent identification, artificial intelligent, mobile applications, Android, API.*

Índice General

Capítulo 1

Introducción

1.1 Objetivo	5
1.2 Alcance	5
1.3 Antecedentes	6
1.4 Destinatarios	7

Capítulo 2

Estudio Previo

2.1 APIs Inteligencia Artificial	8
2.1.1 Sistemas Identificación por Rostro	8
2.2.2 Sistema de Identificación por Voz	12
2.2 Análisis de Herramientas	19
2.3 Lenguajes de programación	21
2.4 Entornos de desarrollo	21

Capítulo 3

Diseño e Implementación

3.1 Diseño General del Proyecto	23
3.2 Diseño de Servicios de Identificación Inteligente	24
3.2.1 Identificación por Reconocimiento de Rostro	24
3.2.2 Identificación por Voz	30
3.3 Diseño de Entorno para los Servicios de Identificación	33
3.3.1 Diseño General de la Aplicación	34
3.3.2 Diseño de Bases de Datos para los Servicios	34
3.3.3 Diseño de Entorno de Administración de los Servicios	36
3.3.4 Diseño de entorno del usuario	40

Capítulo 4

Estudio de los Resultados Obtenidos

4.1 Estudio de Resultados en Servicio de Identificación por Rostro	42
--	----

Capítulo 5

Conclusiones y Líneas Futuras

5.1 Conclusiones	47
5.2 Líneas Futuras	47

Capítulo 6

Conclusions and Future Lines

6.1 Conclusions	49
6.2 Future Lines	49

Capítulo 7

Presupuesto

51

Bibliografía

52

Índice de Figuras

Figura 1. Componentes de TouchID.	6
Figura 2. Funcionamiento de FaceID.	7
Figura 3: Ejemplo de petición para creación de grupos.	9
Figura 4: Ejemplo de los parámetros JSON pasados a la petición de creación de grupos.	9
Figura 5: Ejemplo de respuesta de la API para la creación de una persona	10
Figura 6: Ejemplo del cuerpo de una petición para identificación de una persona a partir de un ID de cara.	11
Figura 7: Ejemplo de objeto JSON retornado por una petición de identificación de cara.	11
Figura 8: Ejemplo de petición para la acción de creación de perfil de la API Speaker Recognition.	12
Figura 9: Ejemplo de objeto JSON insertado en el cuerpo de una petición de creación de perfil para la API Speaker Recognition.	13
Figura 10: Idiomas permitidos por la API Speaker Recognition.	13
Figura 11: Ejemplo de resultado de la acción crear perfil de la API Speaker Recognition.	13
Figura 12: Ejemplo de petición para la acción de crear una inscripción para un perfil de voz en la API Speaker Recognition.	14
Figura 13: Especificaciones requeridas para el archivo de audio incorporado en una petición de creación de inscripción en la API Speaker Recognition.	14
Figura 14: Ejemplo de petición URL para el borrado de un perfil de voz en la API Speaker Recognition.	15
Figura 15: Ejemplo de petición URL para la identificación de personas en la API Speaker Recognition.	16
Figura 16: Objeto JSON obtenido del resultado de la acción Obtener Estado de Operación cuando la operación aun no ha comenzado.	16
Figura 17: JSON del resultado de una operación que aun sigue en ejecución en Speaker Recognition.	17

Figura 18: JSON del resultado de una operación que sufrió algún fallo mientras se completaba en la API Speaker Recognition.	17
Figura 19: JSON resultado de una operación de identificación por voz satisfactoria en la API de reconocimiento por voz.	18
Figura 20: JSON resultado de una operación de identificación por voz que no identificó ningún perfil en la API de reconocimiento por voz.	18
Figura 21: Captura de pantalla del entorno de desarrollo Android Studio.	22
Figura 22: Diseño tomado para el proyecto.	23
Figura 23: Jerarquía de clases del paquete que contiene el Sistema Inteligente de Identificación.	25
Figura 24: Enum ApiConsults implementado en el lenguaje Kotlin.	26
Figura 25: Enum JsonConsults implementado en el lenguaje Kotlin.	27
Figura 26: Ejemplo de una de las clases del paquete components.	28
Figura 27: Implementación del método execute de la clase Controlador del paquete faceAPI.	30
Figura 28: Paquete speakerRecognitionAPI	31
Figura 29: enum ApiConsults del paquete speakerRecognitionAPI en lenguaje Kotlin.	31
Figura 30: enum JsonConsults del paquete speakerRecognitionAPI implementado en el lenguaje Kotlin.	32
Figura 31: Diagrama de Entidad-Relación de la base de datos local.	35
Figura 32: Aspecto de la ventana de grupos.	36
Figura 33: Aspecto del menú de grupos desplegado.	37
Figura 34: Menú flotante de ventana de grupos desplegado.	37
Figura 35: Ventana de creación de grupos.	38
Figura 36: Método que se ejecuta cuando se acciona el botón de la toolbar de la ventana de creación de grupos.	38
Figura 37: Ventana de personas.	39
Figura 38: Pantalla de inicio de sesión.	40

Índice de Tablas

Tabla 1: Datos de confianza obtenidos por el servicio de identificación por rostro.	45
Tabla 2: Presupuesto del proyecto	51

Capítulo 1

Introducción

Actualmente, a pesar de los grandes avances tecnológicos, seguimos aceptando el sistema de inicio de sesión tradicional (email/username y contraseña) como algo ya establecido y definitivo. Esto ocurre incluso en aplicaciones móviles, donde nuestros teléfonos disponen de diversos sensores, como cámara, micrófono, etc. que podrían ser utilizados, ya sea de forma individual o de forma conjunta, para realizar inicios de sesión más avanzados, modernos y seguros. Y si además, los valores obtenidos por los sensores los fusionamos con técnicas de Inteligencia Artificial, podríamos obtener resultados asombrosos que incluso fuesen capaces de aprender por ellos mismos y ser cada vez más potentes y seguros.

1.1 Objetivo

El objetivo del presente proyecto es solventar ese “problema” y obtener un sistema que nos muestre nuestra información personalizada, lo que se podría traducir a realizar un inicio de sesión, mediante sistemas inteligentes, utilizando APIs de campos de la Inteligencia Artificial como el de reconocimiento automático de rostros o de voz. Y con estas medidas, conseguir un sistema que facilite la vida al usuario, siendo mucho más cómodo para él y ahorrándonos problemas como el frecuente olvido de contraseñas, ya que nuestra cara o nuestra voz siempre va a acompañarnos.

1.2 Alcance

Para dar respuesta a las necesidades planteadas en el apartado anterior, fue implementado un sistema que hace uso de APIs de Inteligencia Artificial, más concretamente de las APIs llamadas *FACE API* y *SPEAKER RECOGNITION*, desarrolladas por *Microsoft*. Este sistema es capaz de reconocer al usuario que lo está utilizando gracias a su rostro o voz, y para ser testeado y utilizado, fue implantado en una aplicación móvil para sistemas operativos *Android* que simularía el entorno de una aplicación empresarial, donde un operario inicia sesión con su cara o su voz y puede acceder a todos sus datos laborales. Cabe a destacar que este sistema funciona para un número ilimitado de personas por dispositivo, dando la capacidad a todos los usuarios del sistema a consultar su información personalizada desde la misma aplicación y

dispositivo.

Este sistema funciona de forma automática en la aplicación, abstrayendo al usuario de esta tarea. Dependiendo del contexto, la aplicación lanzará unas u otras tareas para que el inicio de sesión y la muestra de información personalizada sea de la forma más transparente. Así el usuario será beneficiado de estas tecnologías sin a penas darse cuenta, evitándole cargas extras como ya sean la de adaptarse a nuevos entornos, dificultades a la hora del uso de este nuevo sistema, etc.

1.3 Antecedentes

Durante la fase de investigación de este proyecto se han analizado diversas herramientas que, mediante los aspectos cognitivos de los usuarios, son capaces de mostrar información personalizada o iniciar sesión en cualquier sistema, principalmente en aplicaciones móviles. De todas las herramientas analizadas, cabe destacar estas, por su innovación y tecnología puntera que es reforzada con sistemas muy potentes de Inteligencia Artificial:

- **TouchID:** Herramienta desarrollada por *Apple*. Trata de un sistema que levantó muchas expectativas por su fiabilidad y robustez. La función principal de *TouchID* es el desbloqueo de los dispositivos *Apple* mediante el escaneo y reconocimiento biométrico de huellas dactilares, además de poder ser utilizado como un método seguro de autenticación en diversos servicios, compras y aplicaciones.

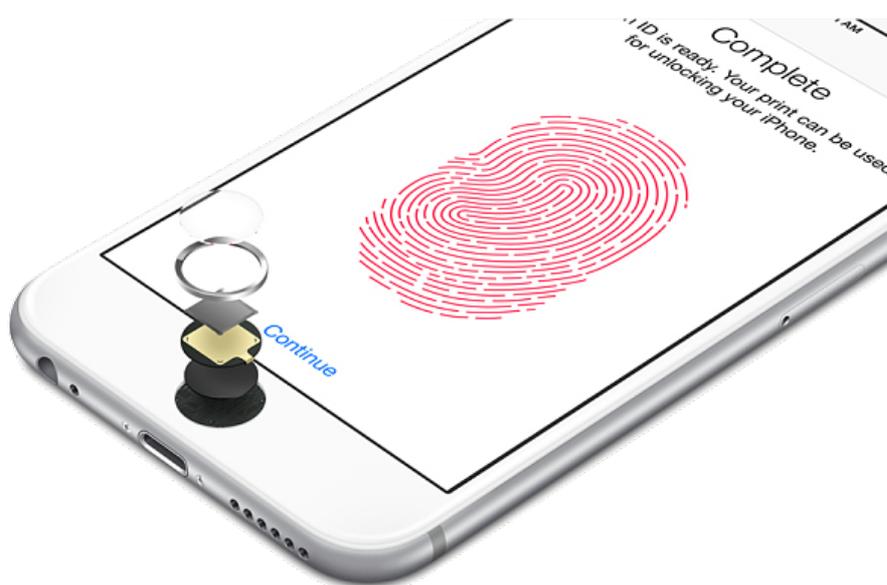


Figura 1. Componentes de *TouchID*.

- **FaceID:** Sistema también desarrollado por *Apple*, esta herramienta tiene mayores similitudes con este proyecto debido a que utiliza también el rostro del usuario para la autenticación. Aunque utiliza novedosas tecnologías de mallas de puntos mediante rayos infrarrojos para detectar un rostro, la finalidad es la misma. Este sistema tiene el inconveniente frente a este proyecto de que sólo reconoce a una persona por dispositivo, mientras este proyecto reconoce a un número ilimitado de personas por aplicación, es decir, *FaceID* sólo comprueba si la persona que posee en ese instante el dispositivo es el usuario propietario del dispositivo, y si es así, accede al contenido. Este proyecto, busca al usuario poseedor dentro del grupo de personas registradas en el sistema, si este usuario existe, se le muestra su contenido personalizado.



Figura 2. Funcionamiento de *FaceID*.

1.4 Destinatarios

Este trabajo está destinado al entorno empresarial, indistintamente del campo de esta.

Puede ser una herramienta útil para toda empresa que disponga de una aplicación móvil donde sus operarios/clientes tengan que iniciar sesión para acceder a su espacio personal.

Capítulo 2

Estudio Previo

A continuación se estudiarán las APIs de Inteligencia Artificial, las posibles herramientas y técnicas para el desarrollo del proyecto.

2.1 APIs Inteligencia Artificial

En este apartado se pasará a detallar las distintas alternativas que se tuvieron en cuenta para conformar los dos sistemas desarrollados en este proyecto.

2.1.1 Sistemas Identificación por Rostro

Existen diversas APIs en el mercado que pueden satisfacer las necesidades de estos sistema. Las tres candidatas para ello han sido:

- **Amazon Rekognition** [1]: API desarrollada por *Amazon* a la que basta con suministrarle una imagen o vídeo y el servicio identificará objetos, personas, texto, escenas y actividades, además de detectar contenido inapropiado.

- **OpenFace** [2]: *OpenFace* es un software desarrollado por un grupo de investigadores de la Universidad Carnegie Mellon de Pensilvania. Su funcionamiento es simple, partir de una imagen se intenta detectar una cara, si ésta es detectada, recorta esta imagen quedándose simplemente con la región de interés de la imagen, es decir con la fracción de imagen que contiene únicamente la cara, desechando toda la información sobrante. Una vez obtenida la imagen resultado, ésta pasa a una red neuronal con aprendizaje profundo (previamente entrenada con las personas que vamos a detectar), y mediante técnicas de *clustering* y clasificación, es capaz de identificar la persona en cuestión. Estas tareas las realiza en unos segundos con un 87% de precisión. Este software, aunque va en una buena dirección, le queda mucho por recorrer, principal razón por la que esta opción fue descartada, a pesar de que contaba con el atractivo de ser software libre, estando alojado en su completitud en la plataforma *GitHub*.

- **FaceAPI** [3]: La última API estudiada para reconocimientos de rostro era la *FaceAPI* de *Microsoft*. La principal candidata debido a sus características y opciones. Su filosofía consiste en la creación de grupos de personas, y

cuando se trata de identificar una persona no se realiza en la red completa, sino directamente en un grupo, que computacionalmente podemos verlo como una *subred*.

Mediante peticiones *HTTP* se pueden realizar todas las acciones que nos permite la API. Estas acciones son [4]:

- **Creación de grupos:** Para esta acción hay que realizar una consulta a la que hay que pasarle como parámetros dos datos. Estos datos son elegidos por el usuario y son:

1. *ID de grupo:* Cadena de letras minúsculas, números, '-', '_' y no mayor de 64 caracteres. Servirá para identificar al grupo en las distintas acciones de la API, por lo que no podrán existir dos grupos con un mismo *ID*.

2. *Nombre de grupo:* Cadena que contendrá el nombre del grupo, es un dato simplemente informativo por lo que no podremos realizar acciones con él.

Un ejemplo de esta petición podría ser:

```
Request URL

https://[location].api.cognitive.microsoft.com/face/v1.0/persongroups/{personGroupId}
```

Figura 3: Ejemplo de petición para creación de grupos.

```
application/json

{
  "name": "group1",
  "userData": "user-provided data attached to the person group."
}
```

Figura 4: Ejemplo de los parámetros JSON pasados a la petición de creación de grupos.

- **Creación de personas:** Con esta acción crearemos una persona dentro de un grupo. La petición web es muy similar a la de creación de grupos, también necesita dos datos para ser pasados como parámetros a la consulta, el *ID* del grupo en el que la persona será insertada y el nombre de la persona, que al igual que el nombre del grupo, es un dato informativo. Para identificar a las personas utilizaremos el *ID* de la persona, este identificador

nos lo devolverá esta petición web.

Un ejemplo de la respuesta de esta API sería esta:

```
application/json
{
  "personId": "25985303-c537-4467-b41d-bdb45cd95ca1"
}
```

Figura 5: Ejemplo de respuesta de la API para la creación de una persona

- **Añadir cara a persona**: Con esta acción asociaremos una cara a una persona ya creada. Para ello, añadiremos como parámetros a la petición el *ID* de la persona a la que se le insertará la cara y el *ID* del grupo al que pertenece la persona. Además, como cuerpo de la petición añadiremos la imagen que contendrá la cara que será añadida a la persona. Esta petición devuelve el *ID* de la cara, que podemos utilizar para acciones como la eliminación de una cara de una persona.

- **Entrenamiento de un grupo**: Una vez se insertan personas, y a su vez, caras en un grupo, es indispensable entrenar este grupo para poder identificar personas dentro de él. Esto se realiza con otra petición a la que únicamente hay que pasarle como parámetro el *ID* del grupo que deseamos entrenar.

- **Identificación de una persona**: Para identificar una persona dentro de un grupo ya entrenado es necesario partir de una imagen de esta persona y la realización de dos acciones:

1. *Detección de cara*: Primero hay que detectar una cara en esta imagen. Si en esta existe una cara, la consulta nos devolverá un *ID* de cara y mucha información a partir de esta, como el género de la persona, edad aproximada, etc. En este proyecto únicamente necesitaremos el *ID*.

2. *Identificación*: Una vez detectada la cara, pasamos a la acción de identificar a la persona. Para ello, existe otra consulta web a la que hay que añadir como cuerpo de petición un objeto *JSON* que contendrá el *ID*

del grupo en el que estamos trabajando y el *ID* de la cara que obtuvimos en la acción anterior. Además, podemos añadirle dos datos opcionales, el máximo de número de candidatos que nos devolverá la petición, que por defecto es 1 y un umbral de confianza, para controlar el grado de precisión que queremos para nuestra petición, por defecto es 0,5.

Un ejemplo de objeto *JSON* que deberá acompañar a la petición podría ser este:

```
application/json
{
  "largePersonGroupId": "sample_group",
  "faceIds": [
    "c5c24a82-6845-4031-9d5d-978df9175426",
    "65d083d4-9447-47d1-af30-b626144bf0fb"
  ],
  "maxNumOfCandidatesReturned": 1,
  "confidenceThreshold": 0.5
}
```

Figura 6: Ejemplo del cuerpo de una petición para identificación de una persona a partir de un *ID* de cara.

El objeto *JSON* que devolverá esta petición contendrá un array de candidatos, este array estará vacío si la persona de la imagen no pertenece al grupo, si existe alguna o algunas personas que pueden ser identificadas con esta cara el tamaño del array dependerá del número de candidatos máximos que le hayamos asignado a la consulta, poniendo en primer lugar el resultado de mayor confiabilidad. Además, este *JSON* también contendrá el porcentaje de confiabilidad de la identificación.

Un ejemplo de respuesta podría ser este:

```
{
  "faceId": "c5c24a82-6845-4031-9d5d-978df9175426",
  "candidates": [
    {
      "personId": "25985303-c537-4467-b41d-bdb45cd95ca1",
      "confidence": 0.92
    }
  ]
}
```

Figura 7: Ejemplo de objeto *JSON* retornado por una petición de identificación de cara.

Además de estas, existen varias acciones simples como son las de borrado de grupos, personas o caras, a las que únicamente hay que pasarles como parámetros el *ID* del elemento que queremos borrar.

Una vez analizadas todas estas opciones, la más adecuada para este proyecto es la *FaceAPI*, pues su filosofía y la gran variedad de acciones que nos permite es la más que encaja con la funcionalidad que tendrá el sistema desarrollado en este trabajo. Además, a pesar de ser una opción no libre, podemos usarla de forma gratuita.

2.2.2 Sistema de Identificación por Voz

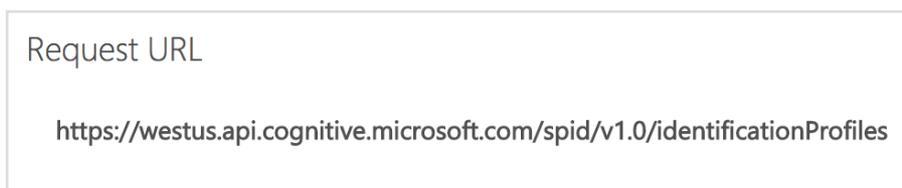
Para este tipo de identificación no existe tantas opciones como para la identificación por rostros, es por eso que el estudio no fue tan extenso resumiéndose todo a una única API estudiada. Esta es la API *Speaker Recognition* de *Microsoft* [5].

Esta API en primera instancia cumplió con creces las necesidades del proyecto, y a pesar de compartir desarrollador con la API elegida para la identificación por voz su filosofía cambia bastante, alejándose del modelo grupo-persona. Es por ello que se tuvo que recurrir a distintas estrategias para traducir su modelo al modelo planteado para esto proyecto y así este encaje a la perfección.

Las acciones que se pueden realizar con esta API mediante peticiones *HTTP* son [6]:

- **Creación de Perfil**: Con esta acción crearemos un nuevo perfil de identificación por voz con un lenguaje especificado en el cuerpo de la petición. Esta es la primera acción que habría que realizar para dar de alta a un usuario en el sistema de identificación por voz.

La *URL* de petición para la siguiente acción no necesita ningún argumento, esta petición es de método *POST*:



```
Request URL  
  
https://westus.api.cognitive.microsoft.com/spid/v1.0/identificationProfiles
```

Figura 8: Ejemplo de petición para la acción de creación de perfil de la API *Speaker Recognition*.

En el cuerpo de esta petición, como se explicó anteriormente, va insertado un objeto *JSON* que contendrá el lenguaje del usuario para el que fue creado el perfil. Esto sería un ejemplo de un cuerpo para un usuario de habla inglesa (inglés americano):

```
application/json
{
  "locale": "en-us"
}
```

Figura 9: Ejemplo de objeto *JSON* insertado en el cuerpo de una petición de creación de perfil para la API *Speaker Recognition*.

Los idiomas permitidos por esta API y su nomenclatura para insertarlos en el cuerpo de la petición son:

- **es-ES (Castilian Spanish)**
- **en-US (American English)**
- **fr-FR (Standard French)**
- **zh-CN (Mandarin Chinese)**

Figura 10: Idiomas permitidos por la API *Speaker Recognition*.

Si todo ha salido perfectamente y no ha habido errores de la API o de conexión, esta API nos devolverá un objeto *JSON* que contendrá una cadena que representa el *ID* de identificación de perfil. Este sería un ejemplo del objeto *JSON* retornado:

```
application/json
{
  "identificationProfileId": "49a36324-fc4b-4387-aa06-090cfbf0064f",
}
```

Figura 11: Ejemplo de resultado de la acción crear perfil de la API *Speaker Recognition*.

· **Crear Inscripción:** El paso posterior a la creación de un perfil es el de creación de una inscripción. Con esta acción añadiremos una voz a un perfil previamente creado.

Para la consulta es necesario pasarle como argumento a la *URL* el *ID* del perfil al que le queremos crear la inscripción. Además, también le podemos pasar como parámetro opción una variable booleana identificada como *shortAudio*. Esta variable está por defecto a *false*, y esto equivale a que el archivo de voz incorporado en esta petición tiene que tener mínimo 15 segundos de duración. Si queremos forzar a la API a que realice esta acción con archivos de audios sin un mínimo de duración tenemos que añadir esta variable a la *URL* igualada al valor *true*. Un ejemplo de la *URL* de esta petición, que también tiene como método *POST*, podría ser:

```
Request URL

https://westus.api.cognitive.microsoft.com/spid/v1.0/identificationProfiles/{identificationProfileId}/enroll[?shortAudio]
```

Figura 12: Ejemplo de petición para la acción de crear una inscripción para un perfil de voz en la API *Speaker Recognition*.

Para el cuerpo de la petición de esta acción, como es previsible, hay que incorporarle un archivo de audio limpio que únicamente contenga la voz del usuario para el que fue creado el perfil y para el que estamos creando la inscripción. Este audio tiene que tener las siguientes especificaciones:

Container	WAV
Encoding	PCM
Rate	16K
Sample Format	16 bit
Channels	Mono

Figura 13: Especificaciones requeridas para el archivo de audio incorporado en una petición de creación de inscripción en la API *Speaker Recognition*.

Si esta petición *HTTP* devuelve *202* como código de respuesta quiere decir que la inscripción para el perfil de voz se ha realizado de forma correcta.

· **Borrar Perfil**: Con esta acción podemos borrar un perfil de voz. Esta acción borra automáticamente las inscripciones que contenga el perfil que estamos borrando.

La petición *HTTP* para esta acción únicamente necesita como parámetro el *ID* del perfil que deseamos borrar. Este sería un ejemplo de petición para esta acción, a la que tenemos que indicar, en el momento de crear la petición que estamos realizando una petición con método *DELETE*:

Request URL

```
https://westus.api.cognitive.microsoft.com/spid/v1.0/identificationProfiles/{identificationProfileId}
```

Figura 14: Ejemplo de petición *URL* para el borrado de un perfil de voz en la API *Speaker Recognition*.

Esta petición no requiere de ningún cuerpo, y únicamente devolverá un código de respuesta *200* si la acción se llevó a cabo de forma satisfactoria.

· **Identificar un Perfil**: Para la identificación de un perfil de voz a partir de un archivo de audio es necesaria la ejecución de dos acciones. Esta acción es una de ellas y ejercerá una función muy importante, ya que es la que realizará las operaciones de cómputo necesarias para la identificación. Más tarde habrá que recoger el resultado de estas operaciones e interpretarlos con otra acción que será detallada más adelante.

Esta acción tiene un claro inconveniente ya que la identificación sólo se puede realizar dentro de un grupo de 10 perfiles de voz, los cuales hay que incluir en la *URL* de la petición mediante su *ID*. Esto presenta una limitación al sistema frente a la API de reconocimiento por rostro, donde la identificación se realiza dentro de un grupo el cual tiene un número ilimitado de personas. Una consulta, por ejemplo sería esta, donde los *IDs* de los perfiles entre los cuales haremos la identificación van separados por comas:

Request URL

```
https://westus.api.cognitive.microsoft.com/spid/v1.0/identify?identificationProfileIds={identificationProfileIds}[&shortAudio]
```

Figura 15: Ejemplo de petición *URL* para la identificación de personas en la API *Speaker Recognition*.

El cuerpo de la petición es igual que para la acción de crear un inscripción, en el que se inserta un archivo de audio con la voz a identificar (con las mismas especificaciones dadas anteriormente) y poniendo a true el parámetro *shortAudio* en la *URL* si no queremos un mínimo estricto de 15 segundos en el archivo de voz enviado.

Si la acción se lleva a cabo de forma correcta (código de respuesta de la petición *HTTP* igual a 202) creará automáticamente una petición para obtener el estado de la operación. La *URL* de esta petición creada se encuentra en los campos del encabezado del resultado. Una vez extraída se procederá a crear una petición con esa *URL* y de ella obtener el resultado de la identificación.

· **Obtener Estado de Operación:** Esta acción será ejecutada una vez hayamos realizado una acción de identificación y con ella obtendremos el estado de la operación. Será la encargada de darnos el *ID* del perfil que hayamos identificado en el archivo de audio insertado en el cuerpo de la petición de identificación anteriormente explicada.

El atributo necesario para realizar la petición es el *ID* de la operación, que nos lo da automáticamente la acción de identificación.

Se pueden dar varias opciones como resultado, todas ellas en formato *JSON*. Estas opciones son:

▶ **No empezado:** Obtendremos este resultado si la operación que estamos consultando aun no ha empezado. Ejemplo:

```
Case 1 - not started
HTTP/1.1 200 Ok
Content-Type: application/json
{
  "status": "notstarted",
  "createdDateTime": "2015-09-30T01:28:23Z",
  "lastActionDateTime": "2015-09-30T01:29:23Z"
}
```

Figura 16: Objeto *JSON* obtenido del resultado de la acción Obtener Estado de Operación cuando la operación aun no ha comenzado.

► Ejecutando: Si aun la operación está en ejecución nos encontraremos en esta situación. El *JSON* obtenido en el resultado que nos alertará de ello será:

```
Case 3 - running
HTTP/1.1 200 Ok
Content-Type: application/json
{
  "status": "running",
  "createdDateTime": "2015-09-30T01:28:23Z",
  "lastActionDateTime": "2015-09-30T01:32:23Z",
}
```

Figura 17: *JSON* del resultado de una operación que aun sigue en ejecución en *Speaker Recognition*.

► Operación Fallida: Si en la operación ocurrió algún error obtendremos este estado. Esto principalmente puede suceder por fallos de conexión, aunque también podríamos encontrarnos con problemas tales como clave de API caducada o fallo de la API. El *JSON* resultado es algo tal que así:

```
Case 4 - failed
HTTP/1.1 200 Ok
Content-Type: application/json
{
  "status": "failed",
  "createdDateTime": "2015-09-30T01:28:23Z"
  "lastActionDateTime": "2015-09-30T01:35:23Z"
  "message": "Some failure info"
}
```

Figura 18: *JSON* del resultado de una operación que sufrió algún fallo mientras se completaba en la API *Speaker Recognition*.

► Operación Completa e Identificación Satisfactoria: Nos encontramos en este escenario si la operación de identificación se llevo a cabo sin ningún problema y esta identificó un perfil a partir del archivo de voz y los *IDs* de los perfiles enviados. El *JSON* que devolverá como resultado será con este formato:

```

Case 5.2.1: Identification Result: in this case, the identification result would be returned inline in "processingResult" field. It successfully identified one of the provided profiles.
HTTP/1.1 200 Ok
Content-Type: application/json
{
  "status": "succeeded",
  "createdDateTime": "2015-09-30T01:28:23Z",
  "lastActionDateTime": "2015-09-30T01:37:23Z",
  "processingResult":
  {
    "identifiedProfileId" : "111f427c-3791-468f-b709-fcef7660fff9",
    "confidence" : "Normal" //[Low | Normal | High]
  }
}

```

Figura 19: *JSON* resultado de una operación de identificación por voz satisfactoria en la API de reconocimiento por voz.

Como vemos en la Figura 19, el objeto *JSON* retornado contiene otro objeto *JSON* llamado *processingResult* que contiene el *ID* del perfil que ha sido identificado en ese archivo de voz.

► Operación Completa e Identificación Fallida: En esta ocasión, la operación se ha completado correctamente pero la API no ha identificado en el archivo de voz adjunto ninguno de los 10 *IDs* incorporados en la consulta *HTTP* de la petición. En este caso, el *JSON* devuelto será así:

```

Case 5.2.2: Identification Result: in this case, the identification result would be returned inline in "processingResult" field. It cannot identify the audio among the provided profiles.
HTTP/1.1 200 Ok
Content-Type: application/json
{
  "status": "succeeded",
  "createdDateTime": "2015-09-30T01:28:23Z",
  "lastActionDateTime": "2015-09-30T01:37:23Z",
  "processingResult":
  {
    "identifiedProfileId" : "00000000-0000-0000-0000-000000000000",
    "confidence" : "Normal" //[Low | Normal | High]
  }
}

```

Figura 20: *JSON* resultado de una operación de identificación por voz que no identificó ningún perfil en la API de reconocimiento por voz.

En este caso, como se aprecia en la Figura 20, el *ID* devuelto es un *ID* reservado ("00000000-0000-0000-0000-000000000000") para indicar que no se pudo identificar ningún perfil en ese archivo de audio adjuntado a la petición de identificación.

2.2 Análisis de Herramientas

A continuación, en este apartado se pasa a detallar el análisis que fue llevado a cabo para la elección de las herramientas que fueron utilizadas en el diseño y desarrollo de este trabajo. También, se explicarán individualmente las herramientas escogidas.

Para evitar continuas llamadas a la API y hacer el sistema más veloz, la mayoría de datos con los que trabajaremos de forma persistente, como los *ID* de los grupos o personas, son almacenados localmente en una base de datos. Para ello, debido a que el trabajo es desarrollado para *Android*, se eligió la herramienta *SQLite*. Por su rapidez y pequeño tamaño es ideal para aplicaciones móviles, convirtiéndose casi en un estándar para el desarrollo de estas.

Todos los datos asociados a la API fueron almacenados en *Microsoft Azure*, los servicios nube de *Microsoft*.

Pero según se avanzaba en el desarrollo de este trabajo, apareció la necesidad del uso de algún servicio de nube o servidor externo, pues había información que se necesitaba almacenar de forma no local y no podía ser almacenada en los servicios nube de la API al no ser datos de esta. Estos datos son los necesitados por la aplicación *Android* desarrollada y que son independientes a la API pero son necesarios para la correcta demostración de la funcionalidad del sistema. Ellos son, la contraseña del usuario (para permitir iniciar sesión de forma tradicional) y las tareas asociadas al usuario. Para solventar este problema, en lugar de recurrir a un servicio externo, se procedió al diseño e implementación de un servidor propio, para así evitar costos y poner de manifiesto conocimientos adquiridos en asignaturas del grado. Para ello, se utilizó como hardware una *Raspberry Pi* en la que se programó un servidor básico en el lenguaje *NodeJS* y un script en la *Bash* para que siempre que esta esté encendida, el servidor esté en ejecución y se puedan hacer peticiones a él. Esta *Raspberry Pi* también cuenta con un servidor de base de datos, gestionado por el sistema de gestión *MySQL*, elegido por su gran popularidad en entornos de desarrollo web. La función del servidor *NodeJS* es la de realizar consultas a este servidor *MySQL*, añadiendo y consultando la información de usuarios que se requiera.

Para el desarrollo de la aplicación móvil en la que se probó el sistema de identificación inteligente se utilizó la herramienta *Android Studio*, pues es la más adecuada al ser la herramienta oficial para el desarrollo de aplicaciones de este entorno.

Y finalmente, como en cualquier proyecto de software fue necesario llevar un control de versiones, para ello se utilizó la tecnología *Git* debido a su

modernidad y robustez, donde se utilizaron varias ramas para la implementación que finalmente se unieron para obtener como resultado el proyecto final. El código de estas versiones fue alojado en el servicio de alojamiento *BitBucket*, ya que este servicio permite repositorios privados de manera gratuita.

Con este análisis podemos concluir definiendo las herramientas seleccionadas:

- *SQLite* [7]: Sistema de gestión de bases de datos, contenida en una pequeña librería escrita en C. Trata de un proyecto de dominio público desarrollado por Richard Hipp. Debido a su pequeño tamaño, SQLite es muy adecuado para los sistemas integrados, como teléfonos móviles.
- *Microsoft Azure* [8]: Lanzado en febrero de 2010 por *Microsoft*, *Microsoft Azure* es un servicio en la nube que está alojado en los *Data Center* de *Microsoft*. Es una plataforma general que tiene diferentes servicios para aplicaciones, desde servicios que alojan aplicaciones en alguno de los centros de procesamiento de datos de *Microsoft* para que se ejecute sobre su infraestructura hasta servicios de comunicación segura y federación entre aplicaciones.
- *Raspberry Pi* [9]: Es un computador de placa reducida de bajo costo desarrollado en el Reino Unido por la Fundación *Raspberry Pi* con el objetivo de estimular la enseñanza de la ciencia de la computación en las escuelas. Se trata de un proyecto de hardware libre y software libre, ya que su sistema operativo oficial es una versión adaptada de la distribución de *Linux*, *Debian*. A la que denominan como *Raspbian*.
- *NodeJS* [10]: Es un un entorno en tiempo de ejecución multiplataforma, de código abierto, fue creado con el enfoque de ser útil en la creación de programas de red altamente escalables, como por ejemplo, servidores web. Fue creado por Ryan Dahl en 2009. La gran ventaja que nos aporta es que no hace falta aprender un lenguaje de programación para escribir el comportamiento del servidor como *Ruby* o *PHP*, sino que nos permite desarrollar una aplicación web al completo únicamente con *JavaScript*. Además cuenta con *NPM (Node Package Manager)* que nos da la posibilidad de instalar infinidad de módulos que podemos incluir en nuestras aplicaciones.

- MySQL [11]: Es un sistema de gestión de bases de datos relacional. Está considerada como la base de datos de código abierto más popular del mundo, sobre todo para entornos de desarrollo web. Su popularidad como aplicación web está muy ligada a *PHP*, que a menudo aparece en combinación con *MySQL*.
- Git [12]: Es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente. Su propósito es llevar registro de los cambios en archivos de computadora y coordinar el trabajo que varias personas realizan sobre archivos compartidos.
- BitBucket [13]: Es un servicio de alojamiento basado en web, para los proyectos que utilizan el sistema de control de versiones *Mercurial* y *Git*. *Bitbucket* ofrece planes comerciales y gratuitos. El servicio está escrito en Python y su propietario es la empresa *Atlassian*.

2.3 Lenguajes de programación

Este proyecto, como trata de un sistema que dará soporte a aplicaciones móviles, en este caso de dispositivos con sistema operativo *Android*, se desarrolló en su totalidad en el lenguaje de programación *Kotlin*, que fue elegido frente a *Java* debido a que este se trata del nuevo lenguaje oficial para el desarrollo de aplicaciones en este sistema operativo.

Kotlin, es un lenguaje de programación de tipado estático, que tiene una sintaxis muy parecida a lenguajes como *Ruby* o *Swift*. Este corre sobre la máquina virtual de *Java* y también puede ser compilado a código fuente de *JavaScript*. Es desarrollado principalmente por *JetBrains*, empresa situada en San Petersburgo, Rusia. El nombre proviene de la Isla de Kotlin, situada cerca de San Petersburgo. Desde 2017 es el lenguaje oficial de *Android*, existiendo la posibilidad de interoperar con Java y las librerías antiguas de *Android* sin problema.

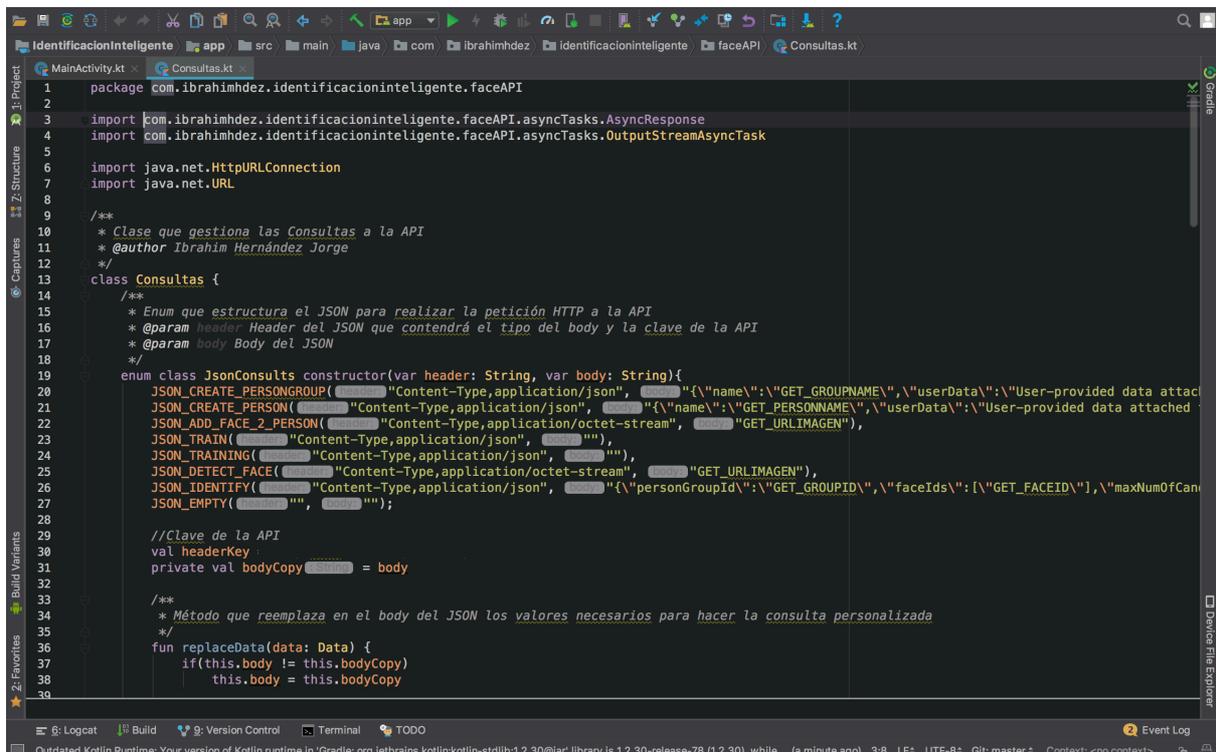
2.4 Entornos de desarrollo

Como se explica en el capítulo 2.2, el entorno de desarrollo elegido para la realización de la aplicación móvil de este proyecto es *Android Studio*.

Este reemplazó a *Eclipse* como el *IDE* oficial para el desarrollo de aplicaciones *Android* y está basado en el software *IntelliJ IDEA* de *JetBrains*.

Sus principales características son:

- Renderizado en tiempo real.
- Contiene consola de desarrollador con consejos de optimización, ayuda para la traducción, estadísticas de uso, etc.
- Soporte para construcción basada en *Gradle*.
- Refactorización específica de *Android* y arreglos rápidos.
- Editor de diseño enriquecido que permite arrastrar y soltar componentes de la interfaz de usuario.
- Herramientas para detectar problemas de rendimiento, usabilidad, compatibilidad de versiones y otros problemas.
- Plantillas para crear diseños comunes de *Android* y otros componentes.
- Un dispositivo virtual de *Android* que se utiliza para ejecutar y probar actualizaciones.



```
1 package com.ibrahimhdez.identificacioninteligente.faceAPI
2
3 import com.ibrahimhdez.identificacioninteligente.faceAPI.asyncTasks.AsyncResponse
4 import com.ibrahimhdez.identificacioninteligente.faceAPI.asyncTasks.OutputStreamAsyncTask
5
6 import java.net.HttpURLConnection
7 import java.net.URL
8
9
10 /**
11  * Clase que gestiona las Consultas a la API
12  * @author Ibrahim Hernández Jorge
13  */
14 class Consultas {
15     /**
16      * Enum que estructura el JSON para realizar la petición HTTP a la API
17      * @param header Header del JSON que contendrá el tipo del body y la clave de la API
18      * @param body Body del JSON
19      */
20     enum class JsonConsults constructor(var header: String, var body: String){
21         JSON_CREATE_PERSONGROUP("Content-Type,application/json", "{\n\"name\":\n\"GET_GROUPNAME\", \"userData\":\n\"User-provided data attac
22         JSON_CREATE_PERSON("Content-Type,application/json", "{\n\"name\":\n\"GET_PERSONNAME\", \"userData\":\n\"User-provided data attached
23         JSON_ADD_FACE_2_PERSON("Content-Type,application/octet-stream", "GET_URLIMAGEN"),
24         JSON_TRAINING("Content-Type,application/json", ""),
25         JSON_DETECT_FACE("Content-Type,application/octet-stream", "GET_URLIMAGEN"),
26         JSON_IDENTIFY("Content-Type,application/json", "{\n\"personGroupId\":\n\"GET_GROUPID\", \"faceIds\": [\n\"GET_FACEID\", \n\"maxNumOfCan
27         JSON_EMPTY("","");
28
29     //Clave de la API
30     val headerKey :
31     private val bodyCopy = body
32
33     /**
34      * Método que reemplaza en el body del JSON los valores necesarios para hacer la consulta personalizada
35      */
36     fun replaceData(data: Data) {
37         if(this.body != this.bodyCopy)
38             this.body = this.bodyCopy
39     }
```

Figura 21: Captura de pantalla del entorno de desarrollo *Android Studio*.

Capítulo 3

Diseño e Implementación

En este capítulo se expondrán las diferentes fases de diseño e implementación transcurridas para todos los componentes de este proyecto.

3.1 Diseño General del Proyecto

Para el proyecto, se ha tomado el siguiente diseño después de un estudio realizado para decidir cual es el más adecuado para este sistema que será utilizado en una aplicación móvil:

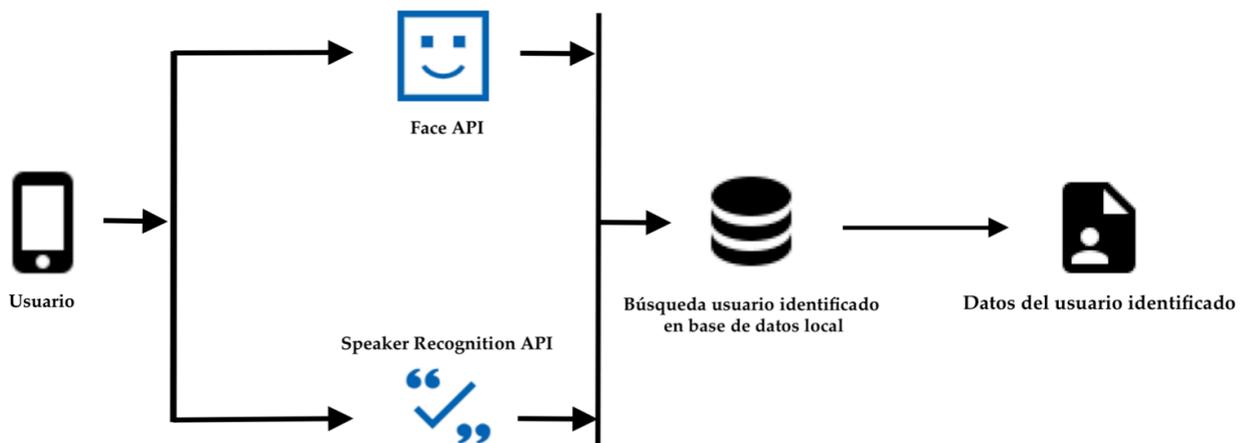


Figura 22: Diseño tomado para el proyecto.

En este diseño, en primera instancia, el usuario es detectado por una de las dos APIs implementadas utilizando los sensores necesarios del teléfono móvil. Una vez detectada la persona, el sistema pasa a identificarla buscándola en su base de datos local. Si esta persona está registrada en el sistema se le mostrará su información.

3.2 Diseño de Servicios de Identificación Inteligente

Los servicios de identificación inteligente se pueden definir como los componentes más importantes de este proyecto ya que son los que moderniza y resuelve los problemas inicialmente citados.

Estos servicios fueron englobados en paquetes individuales que contiene diferentes clases, funcionando a modo de API. Bastaría con incluir estos paquetes en un proyecto de aplicación móvil y realizando las oportunas llamadas a la clase controlador de los paquetes podríamos beneficiarnos de sus funcionalidades, todo esto de una forma bastante sencilla.

Para su diseño e implementación, en primera instancia, fue necesario el estudio exhaustivo de las documentaciones de las APIs pertinentes y realizar numerosos ejemplos utilizándola para así familiarizarse con ellas y tener una mejor base a la hora de diseñar cuál sería la estructura en la que se realizarían las peticiones.

Una vez estudiada las APIs, se procedió a diseñar la composición de estos paquete.

Cabe destacar que se eligió separar el diseño de estos dos servicios por separado aunque no distan mucho uno del otro. Estos dos servicios perfectamente podrían haber sido diseñados para que su implementación fuese en un solo servicio y paquete, pero así, obligaríamos a otros desarrolladores que quieran incluir este proyecto en el suyo a que implementen los dos tipos de identificaciones obligatoriamente. Con este diseño se podría implementar uno de los servicios de identificación sin problemas.

3.2.1 Identificación por Reconocimiento de Rostro

El resultado del diseño para el servicio de reconocimiento de rostro utilizando la API *FaceAPI* fue un paquete con dos subpaquetes y 11 clases, quedando así la jerarquía del paquete:

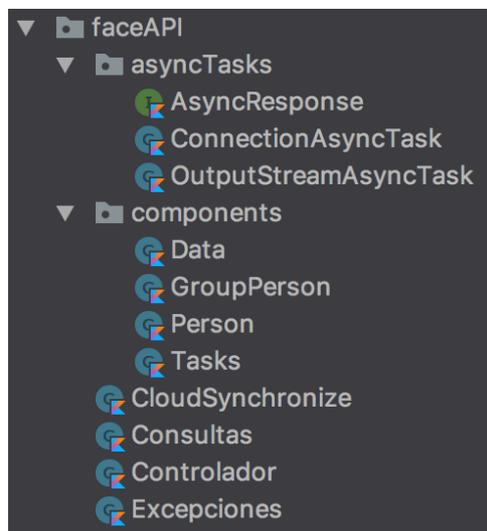


Figura 23: Jerarquía de clases del paquete que contiene el Sistema Inteligente de Identificación.

❖ **Clase Consultas:** Es una de las clases más importantes del paquete. En ella se estructuran las peticiones a la API. Para ello se utilizan dos *enum*. En uno, llamado *ApiConsults*, se construye la consulta con sus parámetros y en el otro, llamado *JsonConsults*, el objeto *JSON* que contendrá los datos para construir el cuerpo de la petición.

ApiConsults [Figura 24] contiene una entrada por cada acción que realiza el sistema sobre la API, esto es: creación de grupos, creación de personas, añadido de caras, entrenamiento de grupos, etc.

Los parámetros que se le pasan para la creación de cada objeto del *enum* son:

- ▶ *requestMethod*: *String* que indica el método de la petición HTTP. Este valor depende de la acción, y puede tener los valores: *PUT*, *POST*, *GET* y *DELETE*.

- ▶ *modulo*: Módulo de la petición web, nos indica si estamos, por ejemplo, tratando con grupos de personas. Este atributo se utiliza para la construcción de la *URL* para la petición a la API.

- ▶ *groupId*: *String* que también se utiliza para la construcción de la *URL*. Puede tener como valor una cadena vacía o "*GET_GROUPID*" que indica que para construir la *URL* necesitamos el *ID* del grupo con el que estamos trabajando. Este valor más adelante con las funciones del *enum* es reemplazado por el *ID*.

- ▶ *accion*: También es un *String* que sirve para la utilización de la

URL. Indica la acción que vamos a hacer, que por ejemplo puede ser la de entrenar un grupo.

▶ *personId*: Cumple las mismas funciones que *groupId* pero este atributo es reemplazado por el *ID* de la persona.

▶ *parametros*: String que complementa la *URL* de la petición con más parámetros si estos son necesarios.

▶ *respuestaJSONArray*: Valor booleano que nos indica si el resultado que devuelve la API es de tipo objeto *JSON* o *array JSON*. Útil a la hora de tratar este resultado.

▶ *json*: Objeto del otro *enum* que contiene esta clase, *JsonConsults*. Contendrá toda la información para construir el cuerpo de la petición. Puede ser *null* si la petición no necesita un cuerpo.

```
/**
 * Enum que estructura las peticiones HTTP
 * @param requestMethod Método de la petición (Puede tener los valores: PUT, POST, GET y DELETE)
 * @param modulo Módulo de la petición web
 * @param groupId ID asociado al GroupPerson con el que queremos trabajar
 * @param accion acción de la petición
 * @param personId ID asociado a la Person con la que queremos trabajar
 * @param parametros Últimos parámetros de la petición web
 * @param json JSON que acompaña a la petición HTTP
 */
enum class ApiConsults constructor(var requestMethod: String, private var modulo: String, private var groupId: String, private var accion: String, private var personId: String, private var parametros: String, var json: String) {
    CONSULT_CREATE_PERSONGROUP("PUT", "persongroups/", "GET_GROUPID/", "", "", "", false, JsonConsults.JSON_CREATE_PERSONGROUP),
    CONSULT_CREATE_PERSON("POST", "persongroups/", "GET_GROUPID/", "persons/", "", "", false, JsonConsults.JSON_CREATE_PERSON),
    CONSULT_ADD_FACE_2_PERSON("POST", "persongroups/", "GET_GROUPID/", "persons/", "GET_PERSONID/", "persistedFaces", false, JsonConsults.JSON_ADD_FACE_2_PERSON),
    CONSULT_TRAIN("POST", "persongroups/", "GET_GROUPID/", "train", "", "", false, JsonConsults.JSON_TRAIN),
    CONSULT_TRAINING("GET", "persongroups/", "GET_GROUPID/", "training", "", "", false, JsonConsults.JSON_TRAINING),
    CONSULT_DETECT_FACE("POST", "detect", "", "", "", true, JsonConsults.JSON_DETECT_FACE),
    CONSULT_IDENTIFY("POST", "identify", "", "", "", true, JsonConsults.JSON_IDENTIFY),
    CONSULT_GET_PERSON("GET", "persongroups/", "GET_GROUPID/", "persons/", "GET_PERSONID/", "", true, JsonConsults.JSON_EMPTY),
    CONSULT_GET_LIST_PERSONGROUPS("GET", "persongroups/", "", "", "", true, JsonConsults.JSON_EMPTY),
    CONSULT_GET_LIST_PERSONS("GET", "persongroups/", "GET_GROUPID/", "persons/", "", true, JsonConsults.JSON_EMPTY),
    CONSULT_DELETE_PERSONGROUP("DELETE", "persongroups/", "GET_GROUPID/", "", "", false, JsonConsults.JSON_EMPTY),
    CONSULT_DELETE_PERSON("DELETE", "persongroups/", "GET_GROUPID/", "persons/", "GET_PERSONID/", "", false, JsonConsults.JSON_EMPTY),
    CONSULT_DELETE_FACE("DELETE", "persongroups/", "GET_GROUPID/", "persons/", "GET_PERSONID/", "persistedFaces/GET_PERSISTEDFACEID", false, JsonConsults.JSON_EMPTY);
}
```

Figura 24: Enum *ApiConsults* implementado en el lenguaje Kotlin.

El otro *enum*, *JsonConsults* [Figura 25], también contiene una entrada por cada acción. Y para construir las instancias se le pasan por parámetro a su constructor los siguientes atributos:

▶ *header*: String que contendrá la clave de la API para poder realizar las peticiones y el tipo de petición que vamos a hacer. Este tipo es para saber si el cuerpo de la petición es un *JSON* o un fichero, en este caso imagen.

▶ *body*: String con los datos que vamos a realizar la petición. Estos datos puede ser, ejemplo, el *ID* de la cara que queremos identificar.

```

/*
 * Enum que estructura el JSON para realizar la petición HTTP a la API
 * @param header Header del JSON que contendrá el tipo del body y la clave de la API
 * @param body Body del JSON
 */
enum class JsonConsults constructor(var header: String, var body: String){
    JSON_CREATE_PERSONGROUP("Content-Type,application/json", "{ \"name\": \"GET_GROUPNAME\", \"userData\": \"User-provided data attached to the PersonGroup.\" }"),
    JSON_CREATE_PERSON("Content-Type,application/json", "{ \"name\": \"GET_PERSONNAME\", \"userData\": \"User-provided data attached to the PersonGroup.\" }"),
    JSON_ADD_FACE_2_PERSON("Content-Type,application/octet-stream", "GET_URLIMAGEN"),
    JSON_TRAIN("Content-Type,application/json", ""),
    JSON_TRAINING("Content-Type,application/json", ""),
    JSON_DETECT_FACE("Content-Type,application/octet-stream", "GET_URLIMAGEN"),
    JSON_IDENTIFY("Content-Type,application/json", "{ \"personGroupId\": \"GET_GROUPID\", \"faceIds\": [\"GET_FACEID\"], \"maxNumOfCandidatesReturned\": 1, \"confidenceThreshold\": 0.5 }"),
    JSON_EMPTY("", "");
}

```

Figura 25: Enum JsonConsults implementado en el lenguaje Kotlin.

- **Clase Excepciones:** Esta clase controlará cualquier error que pueda ocurrir con la API. Esto puede ser: clave de API caducada, creación de un grupo ya existente, tamaño de imagen subida demasiado grande, etc.

Una vez la petición falla, se llama a una de las funciones, dependiendo del contexto, de esta clase y esta mostrará al usuario un mensaje con el error ocurrido.

- **Clase CloudSynchronize:** Esta clase, actualizará la versión local de la aplicación con la última versión que se encuentre alojada en la nube de la API. Esto sirve para tener todos los grupos y personas en los distintos dispositivos que usen una misma clave de API. Así, si por ejemplo se quiere añadir un nuevo operario al grupo de la empresa, no hay que añadirlo en todos los dispositivos de la empresa, sino únicamente en uno de ellos y ya el resto de dispositivos se actualizarán automáticamente. Esta clase no solo comprueba si faltan datos por incluir en la versión local, sino también si los datos ya alojados han sufrido cambios.

Para ello, se utiliza el siguiente algoritmo:

1. Petición a API para obtener todos los grupos.
2. Si alguno de esos grupos no existe en local, se añade con todos sus datos (personas y caras).
3. Si existe un grupo en local y no está en estos grupos descargados quiere decir que se borró desde otro dispositivo, por lo que se elimina de local.
4. Si un grupo local existe en los descargados quiere decir que aun el grupo existe. Pero este puede tener una versión que no sea la más actualizada, por lo que se pasa a actualizar si fuese el caso.
5. Realización de acciones 1, 2, 3 y 4 pero con personas.
6. Realización de acciones 1, 2, 3 y 4 pero con caras.

- **Paquete components:** Este paquete incluye clases con un funcionamiento muy básico, el de crear objetos de grupos, de personas, de tareas y de datos. Estos datos son los del grupo, persona o caras actuales. Necesarios a la hora de elaborar las consultas para las peticiones a la API. Los atributos que contienen las clases son los *IDs* o nombres de los componentes.

```

/**
 * Clase que gestiona las Persons
 * @author Ibrahim Hernández Jorge
 */
class Person {
    var name: String
    var personId: String? = null
    var persistedFaceId: String? = null
    var description: String? = null

    constructor(name: String){
        this.name = name
    }

    constructor(id: String, name: String) {
        this.name = name
        this.personId = id
    }

    constructor(name: String, personId: String, persistedFaceId: String) {
        this.name = name
        this.personId = personId
        this.persistedFaceId = persistedFaceId
    }

    override fun toString(): String {
        return "Nombre: $name \nPersonID: $personId \nPersistedFaceID: $persistedFaceId"
    }
}

```

Figura 26: Ejemplo de una de las clases del paquete components.

- **Paquete asyncTasks:** En él se incluyen todas las clases relacionadas con las tareas asíncronas. En el desarrollo de las aplicaciones *Android*, todas las operaciones que utilizan la red, ya sean de descarga o subida de datos, o la carga, de por ejemplo una imagen en un *buffer*, requieren un tipo de clase especial. Esta es la clase abstracta *AsyncTask*, y en este proyecto, las clases que desempeñan funciones de red y de carga de *buffers*, heredan de esta. Estas clases se ejecutan en segundo plano, mientras el hilo principal de la aplicación sigue con la ejecución de esta. Con esto evitamos que la aplicación se congele cuando estamos descargando o subiendo paquetes de gran tamaño:

❖ **Interface AsyncResponse:** *Interface* sencilla que actúa a modo de evento-delegado con las operaciones de red. Todas las clases que necesitan de conexiones web la implementarán. Dispone de dos métodos, uno que se ejecutará si la petición a la API se desarrolla de forma correcta (método *processFinish*) y otro que se ejecutará en caso de que ocurra un error, generalmente de fallo de conexión web, mientras se realiza la petición (método *processFailed*).

❖ **Clase ConnectionAsyncTask:** Clase para las peticiones web a la API, realiza la consulta y recoge el resultado que esta devuelve. Como hereda de *AsyncTask* y esta es abstracta, implementa sus funciones:

- ***doInBackground*:** Método que ejecutaremos para poner en funcionamiento la tarea asíncrona. Este realiza la petición,

recibe como parámetro una consulta (*ApiConsult*) y devuelve como resultado la consulta que estamos ejecutando. Mientras se ejecuta, comprobamos que la petición se ejecuta correctamente y no hay fallos de conexión. El resultado de este método lo recibe la función *onPostExecute*.

▸ *onPostExecute*: Método que se ejecutará automáticamente (no necesitamos invocarlo desde el código) cuando termina de ejecutarse el método *doInBackground*. Este método, en primer lugar, comprueba si hubo algún fallo de conexión que impidió la correcta llamada a la API, si lo hubo, ejecutará el método de la *interface processFailed*, si no lo hubo, ejecutará *processFinish*.

❖ **Clase OutputStreamAsyncTask**: Esta clase servirá para insertar el contenido del cuerpo de la petición *HTTP* que realiza la petición a la API. Su estructura es la misma que la clase *ConnectionAsyncTask*. Donde el método *doInBackground* recibe la consulta y distingue de dos situaciones:

▸ Inserción de imagen en la petición HTTP: Nos encontraremos en esta situación si la consulta trata de una detección de rostro en una imagen o de un añadido de cara a una persona. En este caso, insertaremos una imagen, la cual todas sus bytes están almacenados en un vector (objeto *ByteArray*), en el flujo de salida de la petición *HTTP* (atributo *outputStream* del objeto *URLConnection*).

▸ Inserción de objeto JSON en la petición HTTP: En este escenario estaremos tratando las demás acciones salvo las consultas con método de petición *GET* y *DELETE*. Estas acciones son las de obtener listados de elementos, ya sea obtener todos los grupos creados en la API o todas las personas pertenecientes a un grupo (*GET*) y todas las acciones relacionadas con borrado de elementos, como borrar una persona, grupo, etc (*DELETE*). Estas acciones no se incluyen debido a que no necesitan un cuerpo de petición.

• **Clase Controlador**: Esta clase será la encargada de comunicar el sistema con las demás clases de la aplicación. Cuenta con diversas funciones, las cuales realizan las acciones de la API. Lo interesante de esta clase es la

concatenación que realiza de las demás clases del sistema para abstraer al programador, por lo que, únicamente con esta clase podemos acceder a todas las funciones de la API. Esta clase, además de ejecutar las diferentes acciones de la API, almacena el resultado que estas arrojan. Esto lo hace con dos de sus atributos, los cuales son: un objeto *JSON* para las consultas que devuelven su resultado en este tipo y un objeto *JSONArray* para el otro tipo de consultas. Sabemos el tipo del objeto devuelto gracias al atributo booleano *respuestaJSONArray* del *enum ApiConsults*. Accediendo a estos atributos desde las otras clases del proyecto podemos obtener y tratar el resultado de la petición. Por otro lado, su método más importante es *execute*, este método recibe como parámetros la consulta que se quiere ejecutar (objeto del *enum ApiConsults* de la clase *Consultas* de este paquete), un objeto *Data* con los datos necesarios para realizar esta consulta y un objeto *AsyncResponse*, para llamar a su delegado una vez las acciones de red hayan finalizado.

```
/**
 * Método que ejecuta una consulta
 * @param consulta Objeto con la consulta a ejecutar
 */
fun execute(consulta: Consultas.ApiConsults, data: Data, asyncResponse: AsyncResponse) {
    //Preparamos los datos y delegados para ejecutar la consulta web
    this.data = data
    this.mainAsyncResponse = asyncResponse
    //Guardamos en el atributo de la clase la consulta API para ejecutarla cuando este preparada
    this.consulta = consulta
    //Ponemos como delegado de la consulta esta clase, así cuando termine la tarea asíncrona se ejecutará el método finishProcess de esta clase
    consulta.delegate = this
    //Reemplazamos los datos dados por la interfaz en la consulta, cuando termine se ejecutará el finishProcess
    consulta.replaceData(data)
}
```

Figura 27: Implementación del método *execute* de la clase Controlador del paquete *faceAPI*.

3.2.2 Identificación por Voz

El diseño final para este servicio, que se basa en la API de *Microsoft Speaker Recognition*, es muy parecido a la estructura implementada para el servicio explicado anteriormente. También se parte de esa filosofía de tratar al paquete que contendrá todas las clases y subpaquetes del sistema como una API, a la cual las demás clases que conformen el proyecto se comunican con ella a través de una clase controlador que unirá todas las clases del paquete, obteniendo así la sensación de tratar con una única clase para acceder a todas las funciones del sistema.

Es por esto que el paquete del servicio queda muy similar al de identificación por reconocimiento de rostro. Este paquete y sus clases son las siguientes:

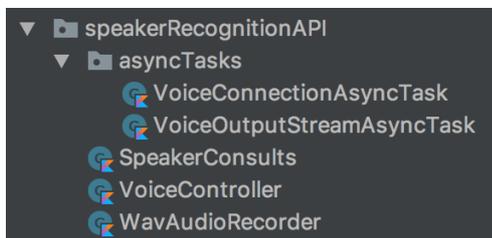


Figura 28: Paquete *speakerRecognitionAPI*

- **Clase *SpeakerConsults***: Esta clase sería la equivalente a la clase *Consultas* del paquete *faceAPI*. En ella se estructuran las llamadas a la API, esto se hace gracias también a los dos *enum ApiConsults* [Figura 24] y *JSONConsults* [Figura 25] pero a diferencia de los anteriores vistos, estos implementan las acciones necesarias para conformar un sistema de reconocimiento por voz siguiendo las especificaciones de la documentación de esta API.

Por un lado, en el *enum ApiConsults* [Figura 29] de este paquete tenemos las acciones de crear un perfil de voz, crear una inscripción para un perfil de voz, borrar un perfil de voz, identificar un perfil de voz y por último, la acción para obtener el estado de una operación de identificación.

Este *enum*, recibe como parámetros para construir todas sus instancias los siguientes atributos:

- ▶ *requestMethod*: *String* que indica el método de la petición HTTP. Este valor depende de la acción, y puede tener los valores: *PUT*, *POST*, *GET* y *DELETE*.
- ▶ *modulo*: Módulo de la petición web, nos indica si estamos, por ejemplo, tratando con grupos de personas. Este atributo se utiliza para la construcción de la *URL* para la petición a la API.
- ▶ *profileId*: *String* que es utilizado para la construcción de la *URL*. Puede tener como valor una cadena vacía o "*GET_PROFILEID/*" que indica que para construir la *URL* necesitamos el *ID* del perfil de voz con el que estamos trabajando. Este valor más adelante con las funciones del *enum* es reemplazado por el *ID*.
- ▶ *json*: Objeto del *enum JsonConsults*. Contendrá toda la información para construir el cuerpo de la petición.

```
enum class ApiConsults constructor(var requestMethod: String, private var modulo: String, private var profileId: String, var json: JsonConsults) {
    CALL_CREATE_PROFILE("POST", "identificationProfiles", "", JsonConsults.JSON_CREATE_PROFILE),
    CALL_CREATE_ENROLLMENT("POST", "identificationProfiles", "/GET_PROFILEID/enroll?shortAudio=true", JsonConsults.JSON_CREATE_ENROLLMENT),
    CALL_IDENTIFY("POST", "identify?", "identificationProfileIds=GET_GROUPPROFILES&shortAudio=true", JsonConsults.JSON_IDENTIFY),
    CALL_DELETE_PROFILE("DELETE", "identificationProfiles", "/GET_PROFILEID", JsonConsults.JSON_DELETE_PROFILE);
}
```

Figura 29: *enum ApiConsults* del paquete *speakerRecognitionAPI* en lenguaje Kotlin.

Y por otro lado tenemos el *enum* *JsonConsults* [Figura 30], que construirá el cuerpo de las peticiones *HTTP* de las acciones. En este caso tendremos una entrada en el *enum* por cada acción implementada en el *ApiConsults*.

Al igual que en el *JsonConsults* del paquete *faceAPI* [Figura 25] tenemos los atributos *header* y *body*, que desempeñan exactamente la misma función.

```
enum class JsonConsults constructor(var header: String, var body: String) {
    JSON_CREATE_PROFILE("Content-Type,application/json", "{\"locale\":\"es-Es\""},
    JSON_CREATE_ENROLLMENT("Content-Type,application/octet-stream", ""),
    JSON_IDENTIFY("Content-Type,application/octet-stream", ""),
    JSON_DELETE_PROFILE("Content-Type,application/json", "");
}
```

Figura 30: *enum* *JsonConsults* del paquete *speakerRecognitionAPI* implementado en el lenguaje *Kotlin*.

- **Paquete *asyncTasks***: En él nos encontramos con las clases necesarias para realizar las tareas asíncronas de este sistema. Estas son las conexiones con la red para las peticiones a la API y la carga de archivos de audios en *buffers* que serán insertados en el cuerpo de las peticiones *HTTP*.

❖ **Clase *VoiceConnectionAsyncTask***: Esta clase es la encargada de las conexiones de red. Realiza las peticiones a la API y espera a que esta devuelva un resultado. Este resultado lo recoge y lo almacena en los atributos de la clase controlador del paquete. Sigue la estructura de las clases heredadas de la clase *AsyncTask* y una vez la tarea es finalizada llama a los métodos de la *interface* *AsyncResponse*, siguiendo el mismo diseño que con *ConnectionAsyncTask*.

❖ **Clase *VoiceOutputStreamAsyncTask***: Clase encargada de incluir el cuerpo en las peticiones a la API. También nos podemos encontrar con dos escenarios:

- **Inserción de Archivo de Audio**: Nos encontraremos en esta situación si estamos creando una inscripción o si estamos identificando un perfil. En estas dos acciones, es necesario la inserción de un archivo de audio en la petición *HTTP*. Esta clase carga en un *buffer* el archivo de audio que contendrá la voz del usuario que queremos dar de alta o identificar y lo inserta en la petición. Una vez haya terminado esta tarea

asíncrona se ejecutará el método *onPostExecute* de la clase y este invocará al método que corresponda del delegado *AsyncResponse*.

▸ Inserción de un objeto JSON: En este caso nos encontraremos con la acción de creación de un perfil. Por lo tanto, es necesario añadirle al cuerpo de la petición un objeto *JSON* que indique el lenguaje del usuario. También ejecutará el método del delegado una vez haya acabado esta tarea.

- **Clase WavAudioRecorder**: Esta clase es la principal diferencia con el paquete del sistema de reconocimiento de rostros. Su función es la de grabar audio con las especificaciones que requieren los archivos de audios enviados a la API. Esta clase fue necesaria ya que por defecto *Android* no graba con las especificaciones necesarias. En ella se realizan las pertinentes transformaciones y añadido de cabeceras a los archivos de audio grabados con el teléfono móvil. Devuelve un archivo de audio que es almacenado en el almacenamiento interno del teléfono, donde posteriormente se utiliza para realizar la petición a la API y borrado cuando deja de ser necesitado (por cuestiones de optimización de memoria).

- **Clase VoiceController**: Clase equivalente a la clase *Controlador* del paquete *faceAPI*. Con ella las demás clases de la aplicación pueden hacer uso de todas las funciones del sistema sin preocuparse por que clases hay que usar.

3.3 Diseño de Entorno para los Servicios de Identificación

Estos servicios de identificación fueron diseñados para que fuesen incluidos en proyectos de aplicaciones móviles. Su diseño está pensado para que sea tan fácil como incluir el o los paquetes en el proyecto de la aplicación y acceder a sus funciones mediante las clases controladores de cada paquete.

Para ello, habría que orientar la aplicación en la que se quieran incluir para que estos funcionen de la manera correcta.

Para este trabajo se implementó una aplicación que incluye estos dos servicios, por lo que en este subcapítulo se explicará el diseño que se tomó

para esta.

3.3.1 Diseño General de la Aplicación

Esta aplicación simula el entorno de una aplicación de una empresa. Donde un trabajador inicia sesión en ella para ver aspectos como las tareas que debe de desempeñar en su jornada laboral.

El diseño que se ha elegido para esta aplicación es el más adecuado para albergar estos dos servicios ya que esta contendrá dos escenarios diferentes. Uno donde el usuario normal actuará que únicamente accederá a las funciones de identificación de los servicios y otro donde el administrador de la aplicación podrá desempeñar las funciones de los servicios que van más allá de la identificación. Tales como dar de alta a nuevos usuarios, añadir sus caras, sus voces, borrar usuarios, etc.

3.3.2 Diseño de Bases de Datos para los Servicios

Para esta aplicación fue diseñada e implementada una base de datos relacional para evitar llamadas a la API en determinadas situaciones.

En ella, insertaremos los grupos, personas, caras, etc. Así, al usar la aplicación únicamente tenemos que consultar la base de datos para conocer los datos y no tener que realizar llamadas con conexiones a la red, esto hace una aplicación más rápida y dejando al usuario una mayor sensación de fluidez.

Con esto, limitaremos las llamadas a la API para únicamente realizar las acciones de:

- Creación de grupos, personas y perfiles de voz.
- Borrado de grupos, personas, perfiles de voz y caras.
- Añadido de caras y voz a personas.
- Obtener *ID* de una persona en tareas de identificación.

Tras realizar estas acciones sus resultados son guardados en la base de datos y cuando sean necesarios se consultaran en ella. Algunas de las acciones que se pueden ahorrar son:

- Consulta de grupos y toda su información (evita acción de obtener grupos de la API).
- A través de un grupo, conocer sus personas y toda su información (evita la acción de obtener personas de un grupo que contiene la API).

- Obtener la información de una cara para por ejemplo, eliminarla (evita la acción de obtener caras de la API).
- Obtener información de una persona, como por ejemplo, su nombre a partir de su *ID*. Este caso se da a la hora de la identificación, ya que la API devuelve el *ID* de la persona que identifica en una imagen. Con la base de datos podemos obtener toda la información de la persona a través de ese *ID* (evita llamada a la API para obtener la información de una persona).

El diseño de la base de datos es la siguiente:

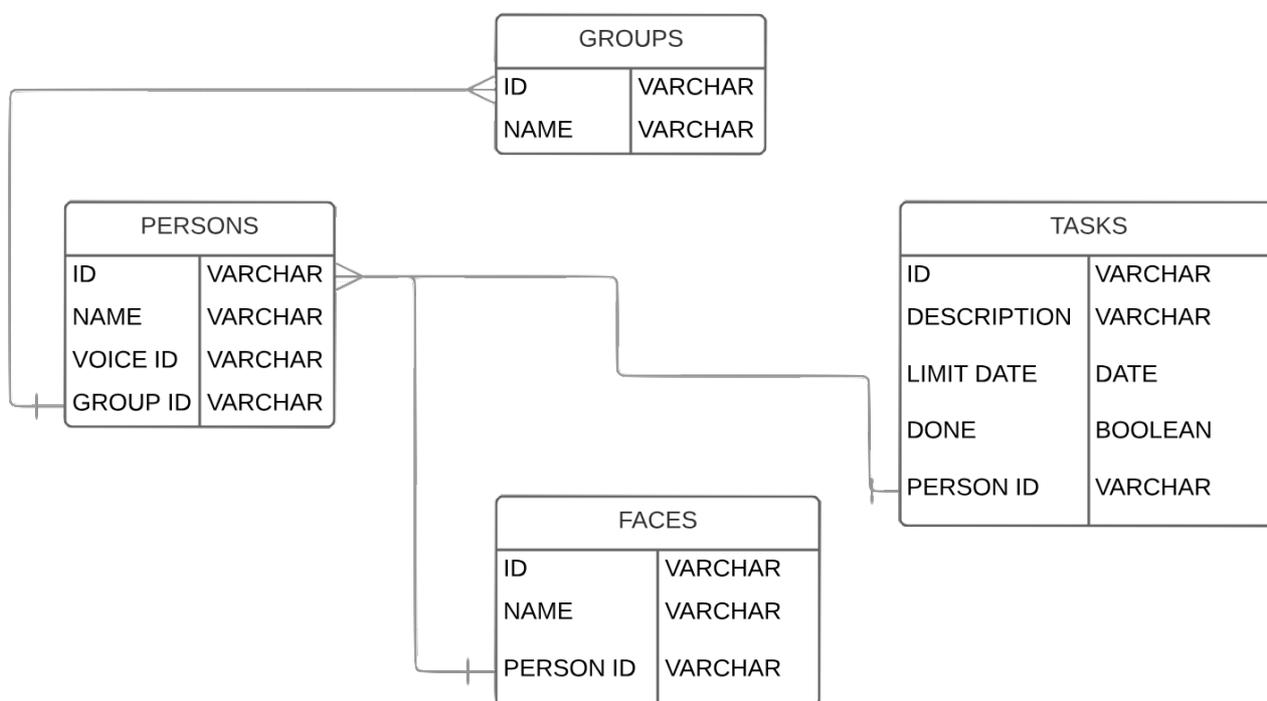


Figura 31: Diagrama de Entidad-Relación de la base de datos local.

Las tablas que nos encontramos son:

- ▶ Groups: Tabla donde serán almacenados los grupos. Clave primaria: *ID*.
- ▶ Persons: Tabla en la que se almacenan las personas. Clave primaria: *ID*. *GroupID* referencia al atributo *ID* de la tabla *Groups*.
- ▶ Face: Tabla para insertar la información relacionada a los rotros. Clave primaria: *ID*. *PersonID* referencia al atributo *ID* de la tabla *Persons*.
- ▶ Tasks: Tabla para almacenar las tareas de los trabajadores con su

correspondiente información. Clave primaria: *ID*. *PersonID* referencia al atributo *ID* de la tabla *Persons*.

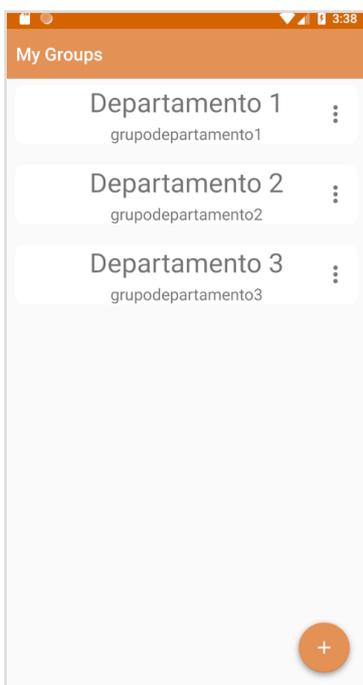
3.3.3 Diseño de Entorno de Administración de los Servicios

Como ya se explicó en capítulos anteriores, la filosofía tomada en estos servicios era la de creación de grupos en los que se insertaban personas y así aligerar la carga computacional de la API a la hora de realizar los pertinentes cálculos para la identificación de una persona. Por ejemplo, si estamos tratando con una empresa muy grande lo más normal es que el administrador de la aplicación cree un grupo por departamento en vez de un solo grupo en el que todos los trabajadores de la empresa se encuentran como integrantes. Así, a la hora de identificar a un trabajador, las comparaciones se realizan únicamente con los integrantes de su grupo, en este caso sus compañeros de departamento y no todos los trabajadores de la empresa.

Es por esto que se vio necesario incluir en esta aplicación un entorno de administración para que así exista un usuario encargado de diseñar la estrategia que tomará la empresa en cuanto a la creación de grupos y perfiles.

Este diseño consta de varias ventanas en la interfaz gráfica que pueden realizar estas acciones. Estas ventanas son:

- **Ventana de Grupos**: En ella el usuario administrador puede realizar todas las acciones pertenecientes a los grupos y se accede a ella a través de las preferencias de la aplicación. Estas acciones son, crear grupos, eliminarlos, ver sus personas y sincronizar la aplicación para obtener la última versión de grupos almacenadas en la nube de la API.



Este es el aspecto de esta ventana [Figura 32]:

Figura 32: Aspecto de la ventana de grupos.

En ella podemos observar todos los grupos existentes. Cada tarjeta representa a un grupo, donde existen 3 elementos:

- ▶ **Nombre del grupo**: Representado con una cadena de caracteres en la parte superior de la tarjeta del grupo.
- ▶ **ID del grupo**: Situado justo debajo del nombre del grupo en una cadena de caracteres con menor tamaño de fuente que este.

► Menú de tres puntos: Este menú es desplegado al ser pulsada la imagen de los 3 puntos. En él nos aparece diversas acciones relacionadas con el grupo que representa esa tarjeta. Estas acciones son:

- View persons: Si es presionada esta acción, pasaremos a otra ventana [Figura 37] similar a esta pero donde observaremos todas las personas integrantes de este grupo. Además podremos realizar diferentes acciones sobre estas personas.

- Train: Con esta acción entrenaremos el grupo. Esta acción se realiza automáticamente cada vez que borramos o insertamos una cara o persona en el grupo, pero existe esta opción manual debido a que nos podremos encontrar en la situación de que la acción de entrenamiento falle por diversos motivos.

- Delete group: Con esta acción eliminaremos el grupo actual tanto de forma local como de la nube de la API.

El menú desplegado (marcado en rojo) tiene este aspecto [Figura 33]:

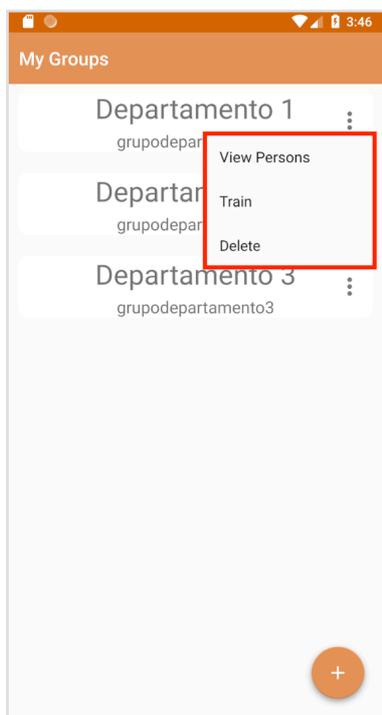


Figura 33: Aspecto del menú de grupos desplegado.

Además en esta ventana nos encontramos con un menú flotante situado en la parte inferior derecha de la ventana. Este menú al ser accionado despliega todas sus acciones [Figura 34]. Estas son:

- Create: Acción que nos pasará a otra ventana donde nos encontraremos con un formulario con el que podremos crear un nuevo grupo.

- Cloud Synchronize: Accionando esta acción sincronizaremos nuestra base de datos local con los datos más recientes alojados en la nube de la API. Se borrarán los grupos y personas que se hayan eliminado desde otro terminal y se añadirán los que hayan sido creados.

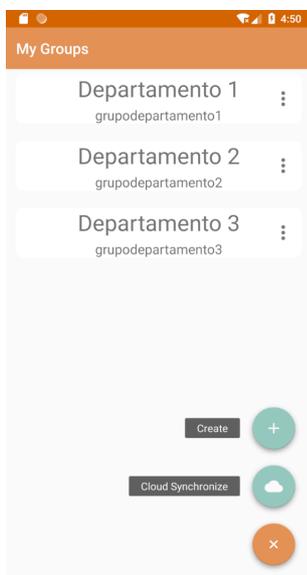
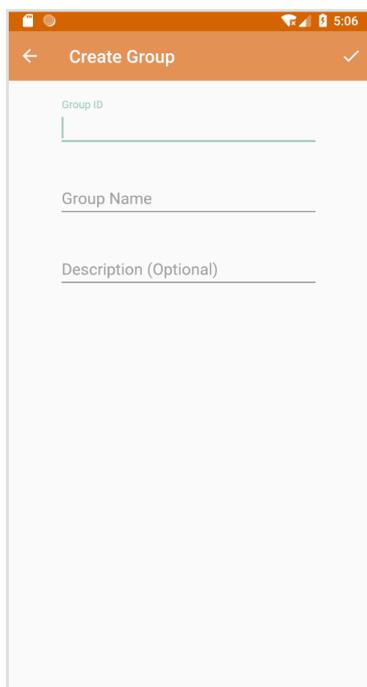


Figura 34: Menú flotante de ventana de grupos desplegado.

- **Ventana de Creación de Grupos:** Esta ventana [Figura 35], a la que se accede desde la acción *create* del menú flotante [Figura 34] de la ventana de grupos [Figura 32], contiene un formulario que habría que rellenar para la creación de un grupo. El formulario contiene los siguientes campos:

- **Group ID:** En este campo habrá que definir el *ID* del grupo que estamos creando. Este *ID* tiene que ser único del grupo, por lo que obtendremos un error por parte de la API si al crear el grupo ya existe el *ID* introducido. El *ID* no puede ser mayor de 64 caracteres.



- **Group Name:** Nombre del grupo. Esto es un valor simplemente informativo ya que únicamente utilizaremos el *ID* para realizar todas las operaciones del grupo.

- **Description:** Campo para añadir alguna descripción al grupo. Es un campo opcional.

Figura 35: Ventana de creación de grupos.

Una vez rellenados los campos, bastaría con accionar el botón que se encuentra en la parte derecha de la *toolbar* [Figura 35] para crear el grupo. Este botón está implementado para que ejecute la acción de crear grupos de la API con los datos introducidos, además, insertará en la base de datos el usuario.

```

/**
 * Método que ejecuta la petición de crear un GroupPerson
 */
private fun execute() {
    this.progress!!.show()
    Comunicador.controlador!!.execute(Consultas.ApiConsults.CONULT_CREATE_PERSONGROUP, Comunicador.data!!, syncResponse this)
}

```

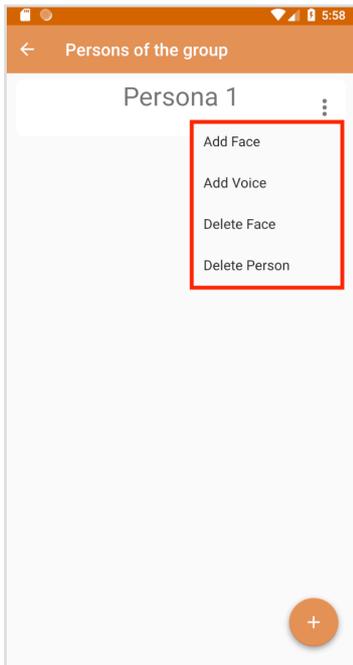
Figura 36: Método que se ejecuta cuando se acciona el botón de la *toolbar* de la ventana de creación de grupos.

- **Ventana de Personas:** Se accede a esta ventana [Figura 37] a través del menú de tres puntos encontrado en las tarjetas de los grupos en la ventana de grupos [Figura 32]. Esta ventana contiene exactamente los mismos elementos que la ventana de grupos con la diferencia de que las tarjetas contenidas en la ventana pertenecen a personas, no a grupos.

Estas tarjetas contienen los mismos elementos que la de los grupos

diferenciándose de las acciones que se despliegan si accionamos el menú de tres puntos. En este caso nos encontramos con:

- Add Face: Esta opción nos pasará a otra ventana donde tendremos la opción de añadir un rostro a la persona de la tarjeta desde donde se accionó el menú.
- Add Voice: Opción que también nos trasladará a otra ventana en la que podremos añadirle a la persona su voz. Cabe a destacar que sólo se permiten 10 personas con voz en un mismo grupo (por limitaciones de la API). Por lo que si ya 10 personas en el grupo donde nos encontramos tiene asignadas una voz esta acción no estará permitida.



- Delete Face: Opción que nos abrirá un diálogo donde veremos todas las caras que se han asignado a esta persona. Podremos seleccionarlas y borrarlas.
- Delete Person: Si pulsamos esta opción se ejecutará el método que realiza una llamada a la API para borrar a la persona del grupo. Además también se borrará de la base de datos local.

Figura 37: Ventana de personas.

Esta ventana también contiene un menú flotante que nos permite la acción de crear una persona. La ventana de creación de personas es la misma que para la de creación de grupos [Figura 35] a diferencia de que los campos del formulario nos indican que estamos rellenando datos para crear una persona y que este formulario contiene una entrada más, esta entrada es para la contraseña del usuario.

- **Ventana de Añadido de Caras**: A esta ventana se accede presionando la opción *Add face* del menú de tres puntos [Figura 37] de la tarjeta de la persona a la que queremos añadir la cara en la ventana de personas.

En esta ventana se le da la opción al administrador de añadirle el nombre a una cara, por si queremos añadirles varias al usuario y queremos distinguir unas de otras, y de añadir una foto que contenga su cara (es importante que no aparezcan más caras en esa foto). El añadido de la foto se puede sacando una foto en ese instante a la persona que quiere ser registrada o seleccionando una foto suya desde la galería.

Una vez realizado todos los pasos y el botón de la *toolbar* sea accionado internamente se van a realizar la acción de la API *FaceAPI* para que sea añadido el rostro a esa persona.

- **Ventana de Añadido de Voz:** A esta ventana podemos acceder desde el menú de los tres puntos [Figura 37] de las tarjetas que podemos encontrar en la ventana de personas. Únicamente nos encontraremos con un botón para grabar la voz del usuario y un botón situado en la *toolbar* que al ser pulsado (siempre y cuando previamente se haya grabado una voz) se desencadenarán dos acciones, la primera será crear para esta persona un perfil de voz (siempre y cuando en el grupo no existan 10 personas ya con perfil de voz creados, debido a la limitación de la API), una vez acabada esta acción, se activará el evento que realiza la siguiente petición, esta será la de crear la inscripción con la voz grabada para el perfil recientemente creado. La aplicación recogerá el archivo de voz grabado en esta ventana y lo insertará en el cuerpo de la petición *HTTP* que vamos a realizar a la API de *Speaker Recognition*, posteriormente lo borra. Si esta petición devuelve un código de respuesta satisfactorio podemos dar como concluido la tarea y este usuario podría ser identificado por este método.

3.3.4 Diseño de entorno del usuario

Para el entorno de usuario fue creada una ventana que se asimila a la tradicional ventana de inicio de sesión [Figura 38]. En ella nos encontramos con los campos para introducir usuario (nombre de la persona con la que se dio de alta) y contraseña. Además también nos encontraremos con unos botones para realizar el inicio de sesión mediante rostro y voz de forma manual.

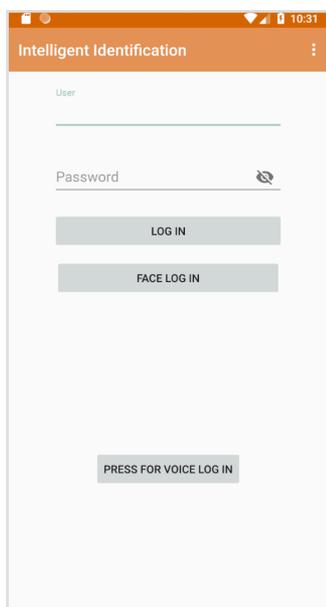


Figura 38: Pantalla de inicio de sesión.

En primera instancia, cuando se inicia la aplicación lo que esta hace es capturar nuestro rostro, y con la imagen de nuestro rostro ejecuta la acción de identificación de la API. Para ello se deben de realizar dos acciones, la primera acción a realizar es la de detección de un rostro en la imagen tomada al iniciar la aplicación. Esta imagen se inserta en el cuerpo de la consulta *HTTP* para hacer una petición a la API de detección de rostros. Una vez la API haya realizado sus

cálculos de detección de rostro en la imagen nos devolverá un *ID* de cara que es utilizado para los parámetros de la siguiente consulta.

Una vez esta acción se haya completado correctamente, un evento activará su delegado y este realizará automáticamente la siguiente acción necesaria para la identificación de la persona. Esta acción trata de una petición a la API mandando como cuerpo de la petición el resultado de la acción anterior, que trata del *ID* del rostro identificado. Si la API identifica a alguna persona dentro del grupo con el rostro detectado nos devolverá como resultado el *ID* de esa persona. El paso posterior es consultar en la base de datos a qué persona pertenece ese *ID* e iniciar su sesión mostrándole su información personalizada.

Si la acción de identificación es errónea, ya que podría darse el caso de que no exista un rostro en esa imagen, que esa persona no esté registrada en la API o que ocurra cualquier problema de conexión o de la API, el sistema avisará al usuario con un mensaje personalizado mostrando el error. El usuario en cualquier momento puede decidir si vuelve a probar la identificación por rostro accionando el botón de *Face Log In* que detecta de forma transparente para el usuario su rostro y repite el proceso anteriormente explicado para la identificación. También tiene la opción por inicio de sesión tradicional insertando su nombre y su contraseña o utilizando el sistema de identificación por voz.

Para la identificación por voz el usuario deberá de presionar el botón de *Voice Log In* y grabar su voz. Cuando vuelva a pulsar el botón para finalizar la grabación, el archivo de voz recientemente grabado es insertado en el cuerpo de la consulta *HTTP* para realizar la petición a la API. Una vez esta API realice los cálculos, nos devolverá el resultado. Si es satisfactorio se procederá a buscar en la base de datos el *ID* del perfil que hemos recibido como resultado de la operación y posteriormente, se le mostrará la información personalizada a este usuario.

Capítulo 4

Estudio de los Resultados Obtenidos

Una vez confeccionados los dos servicios, se pasó a valorar los resultados obtenidos tras varios usos del sistema. Para ello, en el servicio de identificación por rostro fue muy útil los datos devueltos por la API haciendo referencia a la confianza que se ha obtenido al identificar un usuario. Este dato viene insertado en el objeto *JSON* de respuesta de la petición acompañando al *ID* de la persona identificada. Gracias a esto se pudo realizar un profundo análisis del resultado, no siendo así en el servicio de identificación por voz, donde la API no devuelve resultados de la confianza de la identificación, por lo que el estudio se tuvo que basar al 100% en los usos prácticos del sistema.

4.1 Estudio de Resultados en Servicio de Identificación por Rostro

Aprovechando los resultados de confianza dados por la API de reconocimiento por rostro (valores entre 0.5 y 1), se sometió el sistema a pruebas en escenarios donde podríamos, a priori, intuir que los resultados no fuesen los esperados.

En primer lugar, las pruebas fueron realizadas en un escenario perfecto, así, a partir de estos resultados podremos valorar si los posteriores obtenidos en escenarios con diferentes dificultades son satisfactorios o no. Entendemos escenario perfecto como un lugar de mucha luz y donde el usuario está de cara a la cámara del teléfono móvil (tanto en la identificación como en el momento de la toma de la foto para el añadido de cara en el sistema al usuario). Tras varias pruebas, con hasta 5 personas diferentes, los resultados obtenidos en esta situación tienen como media un valor de confianza de 0.946 (94'6%).

A partir de aquí, sabiendo que 0.946 es un valor muy bueno de confianza se pasó a dificultarle la tarea al sistema para valorar su efectividad. Estos escenarios a los que se sometió el sistema fueron:

- **Compresión de las imágenes:** Al tratarse de una API a la que nos comunicamos vía red se realizaron pruebas disminuyendo la calidad de

las imágenes con la que realizábamos las peticiones, para así agilizar la subida de imágenes (acciones de añadidos de rostros e identificaciones) y con esto obtener los resultados de la API de una manera más rápida. Se fue disminuyendo la calidad de la imagen de una forma paulatina y observando los resultados hasta que llegamos al punto donde disminuyendo la calidad de la imagen un 70% obteníamos como resultado una media de 0.86 de confianza. Estos son unos resultados sorprendentes, ya que quedándonos con el 30% de información de una imagen conseguimos mantener un resultado de confianza bastante alto, donde apenas se perdieron 0.09 (9%) puntos de confianza. Una vez en conocimiento de estos datos, se procedió a llevar a cabo esta acción de comprensión de imágenes para el funcionamiento final del sistema, reduciendo notablemente el tiempo de comunicación con la API en las acciones que requerían de imágenes para ser llevadas a cabo.

- **Usuarios con gafas:** Que un usuario poseedor de gafas, tanto de sol como de vista, use este sistema es algo muy probable. Por lo tanto, se procedió a estudiar la reacción del sistema frente a estos cambios. Se probaron dos situaciones, la primera situación donde se registraba al usuario con una foto donde portaba las gafas y éste se identificaba en el sistema sin ellas y la segunda situación era justo la contraria de la primera. Estos resultados bajaban en un 10% de media los resultados obtenidos en un escenario perfecto, esto es, obteniendo como resultado al rededor de 0.81 puntos de confianza. A pesar de seguir siendo una confianza bastante alta sería recomendable, en el caso de usuarios portadores de gafas, que registrasen dos fotos a la hora de añadir su rostro, una con gafas y otra sin ellas y con esto, asegurarnos de que obtendremos una confianza que rondará los 0.90 puntos a la hora de su identificación.
- **Personas de parecido razonable:** Se procedió a registrar en el sistema a dos personas con una gran similitud facial entre ellas. En este caso no hubo problemas y el sistema fue capaz de identificar a cada uno de ellos con confianzas muy altas (entre 0,9245 y 0,935).
- **Gemelos:** Aquí se estudiaron dos casos. En primer lugar, se registraron dos gemelos en los que existían diferencias, aunque éstas muy leves. El sistema en este caso supo identificar de forma correcta a los dos individuos, aunque con una tasa de confianza muy baja, rondando los 0.70 puntos. Por lo que sería recomendable que, en estos casos, a la hora de registrar a los usuarios se haga con varias imágenes, así la API contiene un

mayor conocimiento de sus rostros y puede apurar al máximo la confianza a la hora de identificarlos. En segundo lugar, se probó el sistema con dos gemelos de apariencia idéntica. En este caso, empezaron a aparecer los problemas, donde el sistema no era capaz de identificarlos de forma correcta dando una pobre puntuación de confianza (0.509 - 0.53294). Para esta situación, lo recomendable, a parte de registrarlos en el servicio con el máximo número de imágenes posibles, sería registrarlos en el servicio de identificación por voz y cuando se detecte una confianza tan baja, realizar una doble comprobación, realizándola esta vez con su voz.

- **Ojos cerrados**: Se investigó si a la hora de identificarse podría ser un problema que el sistema nos capte con los ojos cerrados. Analizando los resultados obtenidos se comprobó que esto no resultaba un problema, pues los puntos de confianza a penas variaban frente a una situación normal donde nos encontramos con los ojos abiertos.
- **Captura de rostros en posición de perfil**: Se procedió a probar el sistema con imágenes del rostro del usuario de perfil (no estando su rostro de forma frontal hacia la pantalla del teléfono). Procedimos a probar con este tipo de situación tanto a la hora de identificación como a la de añadido de rostro, donde un usuario se registraba con una foto suya con el rostro girado y se identificaba con el rostro hacia la pantalla y también la situación contraria. En estos escenarios, el sistema acertaba a la hora de la identificación pero con una confianza menor, bajando hasta en un 20% los resultados obtenidos (0.6944 - 0.7211). Como se trata de una confianza baja, se recomendaría que a la hora de registrar a un usuario se haga con una imagen suya frontal y otra de perfil, ya que es bastante probable que a la hora de captar al usuario para identificarlo este se encuentre mirando hacia otro lado y no hacia el teléfono móvil. Basta con registrar al usuario con estas dos imágenes para que sea insignificante si a la hora de identificarlo éste está de perfil o no, pues la confianza que devolverá la API será alta (> 0.90).
- **Oscuridad**: A priori, como no disponemos de dispositivos infrarrojos a la hora de captación del usuario, sino únicamente de la cámara del teléfono, podríamos intuir que la oscuridad podría ser un problema para nuestro sistema. Pero una vez analizados los resultados, podemos afirmar que nuestro sistema funciona a la perfección en situaciones de escasa o inexistente iluminación ambiental. La media de los datos de confianza alcanza los 0.80 puntos, por lo que pueden ser categorizados como

resultados muy buenos. Esto se debe en gran parte al diseño de la ventana de identificación, donde fue elegido el blanco como color de fondo, dando al ambiente un brillo artificial mínimo (no necesario brillo de pantalla en niveles altos) para que nuestro sistema pueda funcionar de forma correcta.

Podríamos resumir los datos obtenidos en esta tabla [Tabla 1]:

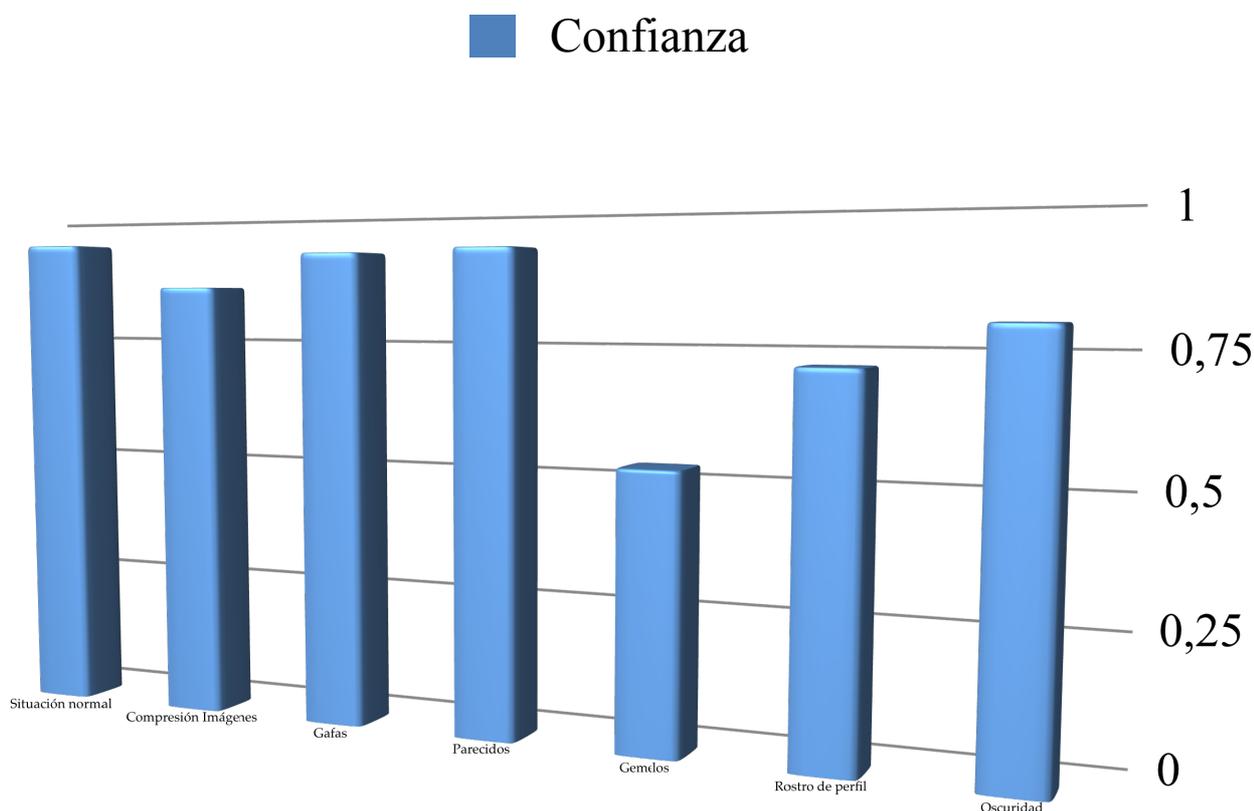


Tabla 1: Datos de confianza obtenidos por el servicio de identificación por rostro.

4.2 Estudio de Resultados en Servicio de Identificación por Voz

En este caso, el estudio no pudo ser tan extenso y preciso, pues la API no devolvía resultados de confianza a la hora de la identificación del usuario. Es por esto que se tuvo que recurrir a la valoración de los resultados por las pruebas prácticas realizadas con este servicio.

En resumidas cuentas, el sistema en situaciones normales siempre da resultados positivos, tanto en el registro de la voz como en la identificación. Entendemos situaciones normales como situaciones en las que el usuario del día a día se encontrará y donde no existe un ruido excesivo que pueda

dificultar estas tareas.

El sistema funciona correctamente en lugares donde existe ruido considerable, como por ejemplo la calle. En cambio, en situaciones donde nos encontremos, por ejemplo en una habitación con la música muy alta, al sistema le será una tarea muy difícil distinguir nuestra voz, pues en 20 pruebas sólo fue capaz de identificarnos en 4 ocasiones. Es por esto que es recomendable registrar también el rostro del usuario, así si nos encontramos en una situación similar a esta, podemos utilizar el otro servicio de reconocimiento para identificarnos y visualizar nuestra información personalizada.

Capítulo 5

Conclusiones y Líneas Futuras

5.1 Conclusiones

Tras haber cumplido los objetivos principales que fueron planteados en un principio, se puede llegar a la conclusión de que se ha alcanzado satisfactoriamente los objetivos del Trabajo de Fin de Grado, logrando crear con éxito un sistema de identificación inteligente para aplicaciones móviles.

Se han obtenido unos resultados bastante satisfactorios, obteniendo un sistema bastante estable y con buenas características teniendo en cuenta las restricciones temporales con las que se contaba.

Por otro lado, la realización de este Trabajo de Fin de Grado ha conllevado un amplio estudio sobre herramientas como las APIs de Inteligencia Artificial desarrolladas por Microsoft. Además de esto, se ha conseguido ampliar los conocimientos en herramientas ya conocidas como el desarrollo de aplicaciones Android. Permiéndome así obtener unos grandes conocimientos en estas materias tan actuales.

Durante el desarrollo del sistema, cabe a destacar el grandísimo reto personal que ha supuesto, ya que se han presentado diversas dificultades, como las propiciadas por el desconocimiento del mundo de las APIs o de la programación de aplicaciones móviles que realicen conexiones a la red. Todo esto pasando por diferentes entornos y lenguajes los cuales no se tenían conocimientos previos.

5.2 Líneas Futuras

Este proyecto puede ampliarse en distintas líneas de trabajo y nuevas funcionalidades, como por ejemplo, añadir la posibilidad de que el sistema de identificación por reconocimiento de rostros sea capaz, a la hora de dar de alta a un nuevo usuario, de avisarnos si en el grupo donde lo estamos insertando existe un usuario tan parecido a él como para existir el peligro de que el sistema falle en el momento de la identificación, dando un resultado distinto al esperado. Esto es probable que pase con gemelos de alta similitud de apariencia. En esta ocasión el sistema podría obligar a dar de alta a este usuario en el servicio de identificación por voz, y en el momento de que se identifique a uno de estos usuarios con una gran probabilidad de error a

través de su rostro, obligar a realizar una segunda confirmación, esta vez con su voz.

También se le podría incluir al sistema la capacidad de que vaya aprendiendo del rostro de un usuario, así con esto conseguir reducir la vulnerabilidad del sistema frente a cambios faciales. Para esto, cada cierto tiempo o después de cada número, previamente establecido, de identificaciones, se utilice una de las imágenes tomadas del usuario para su identificación para añadirla como nuevo rostro a este y posteriormente volver a entrenar la red, así vamos obteniendo pequeños cambios faciales y siempre teniendo una versión actual del rostro del usuario.

Capítulo 6

Conclusions and Future Lines

6.1 Conclusions

After having fulfilled the main objectives that were raised at the beginning, it can be concluded that the objectives of the Final Degree Project have been satisfactorily achieved, successfully creating an intelligent identification system for mobile applications.

We have obtained quite satisfactory results, obtaining a fairly stable system with good characteristics taking into account the temporary restrictions that were available.

On the other hand, the completion of this Final Degree Project has led to a broad study on tools such as Artificial Intelligence APIs developed by Microsoft. In addition to this, it has managed to expand knowledge in tools already known as the development of Android applications. Allowing me to obtain great knowledge in these matters.

During the development of the system, it is worth highlighting the great personal challenge that has been involved, since various difficulties have arisen, such as those caused by the ignorance of the world of APIs or the programming of mobile applications that make connections to the network. All this going through different environments and languages which had no previous knowledge.

6.2 Future Lines

This project can be extended in different lines of work and new functionalities, such as adding the possibility that the identification system by face recognition is able, at the time of registering a new user, to let us know if in the group where we are inserting there is a user so similar to him that there is a danger that the system fails at the time of identification, giving a different result than expected. This is likely to happen with twins of high similarity of appearance. On this occasion, the system could force this user to register in the voice identification service, and at the moment that one of these users is identified with a high probability of error through his face, to force the user to perform A second confirmation, this time with your voice.

The ability to learn from a user's face could also be included in the system,

thus reducing the system's vulnerability to facial changes. For this, every so often or after each number, previously established, of identifications, one of the images taken from the user is used for identification to add it as a new face to this and later retrain the network, so we get small changes Facials and always having a current version of the user's face.

Capítulo 7

Presupuesto

En este capítulo se ofrece un presupuesto para el desarrollo íntegro del proyecto desglosado en función de las horas invertidas. Suponiendo un coste de 20€/hora:

Tarea	Horas	Coste (€)
Análisis de requisitos	25	500
Documentación de las herramientas seleccionadas	50	1.000
Diseño del modelo	20	400
Implementación del sistema de identificación por rostro	200	4.000
Implementación del sistema de identificación por voz	80	1.600
Implementación de interfaz gráfica de aplicación que implementa los sistemas	50	1.000
Fase de investigación para optimizar sistemas	100	2.000
Total	525	10.500 €

Tabla 2: Presupuesto del proyecto

Bibliografía

- [1] API Rekognition de Amazon: <https://aws.amazon.com/es/rekognition/>
- [2] Repositorio de OpenFace en GitHub: <https://cmusatyalab.github.io/openface/>.
- [3] Página de la API FaceAPI de Microsoft: <https://westus.dev.cognitive.microsoft.com/docs/services/563879b61984550e40cbbe8d/operations/563879b61984550f30395244>
- [4] Documentación de la API FaceAPI de Microsoft: <https://azure.microsoft.com/es-es/services/cognitive-services/face/>
- [6] Página de la API Speaker Recognition de Microsoft: <https://azure.microsoft.com/es-es/services/cognitive-services/speaker-recognition/>
- [6] Documentación de la API Speaker Recognition de Microsoft: <https://westus.dev.cognitive.microsoft.com/docs/services/563309b6778daf02acc0a508/operations/5645c068e597ed22ec38f42e>
- [7] Página oficial de SQLite: <https://www.sqlite.org/index.html>
- [8] Página oficial de Microsoft Azure: <https://azure.microsoft.com/es-es/>
- [9] Página oficial de Raspberry Pi: <https://www.raspberrypi.org/>
- [10] Página oficial de NodeJS: <https://nodejs.org/es/>
- [11] Página oficial de MySQL: <https://www.mysql.com/>
- [12] Página de Git: <https://git-scm.com>
- [13] Página web de BitBucket: <https://bitbucket.org/>