

Desarrollo de Aplicaciones Multiplataforma

# Trabajo de Fin de Grado

GMQ

IGNACIO CAMPO MARTIN y NURIA CALABRESE  
VILAGUT  
1-1-2021



## **Índice de contenidos:**

### **1- Resumen de contenidos**

### **2- Introducción**

### **3- Objetivos**

### **4- Tecnologías usadas**

#### **I. Microsoft Azure**

#### **II. Twilio**

#### **III. Android**

#### **IV. Spring**

#### **V. PostgresSql**

### **5- Detalles técnicos**

#### **I. Explicación del código del back-end**

#### **II. Explicación del código del front-end**

#### **III. Base de Datos**

### **6- Manuales para el usuario**

#### **I. Manual de creación de los servicios de Azure**

### **7- Bibliografía**

# 1-Resumen

Nuestro proyecto, consiste en la creación de un aplicativo creado para Android, pensado para un uso empresarial, donde los empleados tengan desde sus *smartphones* acceso a datos relacionados con su puesto de trabajo (información personal, nóminas...) así como, la función que consideramos más importante implementada, la posibilidad de poder fichar desde el propio teléfono móvil mediante un reconocimiento facial a la par que la geolocalización del empleado, usando así tecnologías que actualmente se está incrementando su uso en este tipo de aspectos como son el uso de servicios cognitivos.

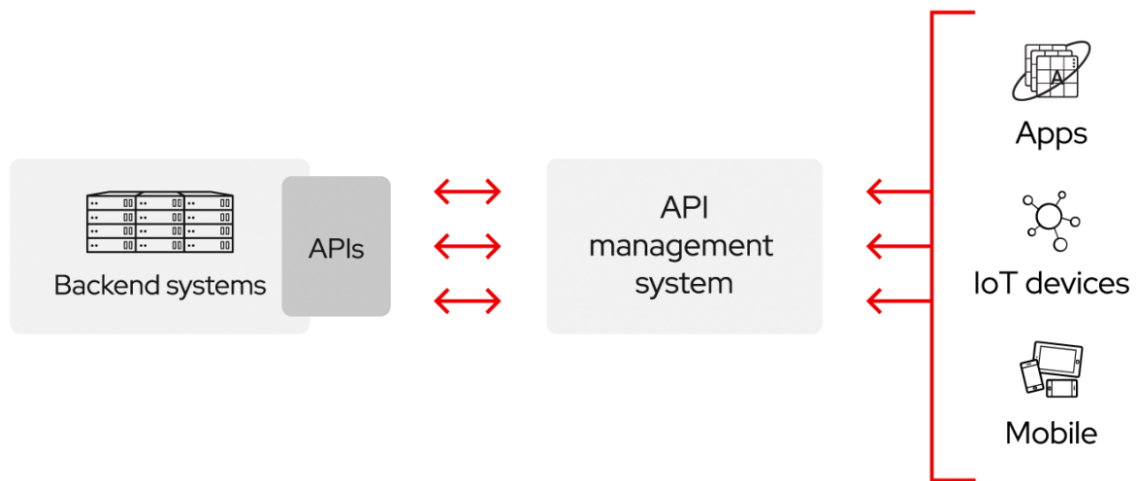
## 2-Introducción

Creemos que la realización de esta aplicación es una ventaja empresarial para que, teniendo en cuenta la importancia que se le ha dado al contacto debido a la alarma sanitaria que está viviendo el ser humano actualmente, podamos realizar tareas tan cotidianas como el fichaje de un empleado absteniéndonos de entrar en contacto con objetos físicos de forma innecesaria (como puede ser un terminal donde fichen todos los empleados a la entrada, o un ordenador donde los empleados visualicen sus datos).

Además, creemos que este modelo de fichaje, al ser una forma más “extravagante” y moderna a la que la gente no está acostumbrada, llamará la atención de los empleados para aceptar el uso de esta debido a la comodidad y la seguridad que genera el poder acceder desde tu móvil a tus datos, sin que tenga que ser un ordenador común donde puedas dejar tus credenciales guardadas por algún error y pueda acceder a ellas algún otro empleado.

Entrando más en detalle en la realización del proyecto, podemos decir que el aplicativo está dividido en tres apartados: nuestro *front-end* (parte visible de la aplicación para el empleado) desarrollado en el lenguaje **java** mediante el IDE de desarrollo de Android Studio. Usamos para la parte *back-end* (parte donde queda integrado el desarrollo de las funcionalidades del aplicativo) realizado también en lenguaje **java** desde el IDE Spring, el cual es una “extensión” del IDE de *Eclipse Enterprise Edition*. Por último, encontramos nuestra *Base de Datos*, donde almacenamos la información de cada empleado mediante PostGresSql.

Para entender bien nuestro aplicativo, debemos definir y entender también que es una API, ya que principalmente nuestra aplicación utiliza APIS para consumir servicios externos como son los de Microsoft Azure o Twilio, así como para conectar nuestro front-end con nuestro back-end. Una API (*Interfaz de Programación de Aplicaciones*) es un conjunto de servicios y protocolos utilizados en el desarrollo de software. Una API permite conectar varios servicios entre sí sin necesidad de conocer como están implementados. Estas otorgan simplicidad en el diseño, administración y uso de las aplicaciones, proporcionando mayor facilidad a la hora de implementar cambios en una aplicación.



*Ilustración 1 Explicación del funcionamiento de una API*

Explicaremos un poco más en detalle cada uno de estos entornos en el apartado 4.

### 3-Objetivos

El objetivo principal de este proyecto es plasmar, tanto los conocimientos adquiridos en nuestros años en el grado, así como los conocimientos en estos entornos tan innovadores como son el uso de servicios cognitivos y escáneres biométricos, queriendo demostrar así el provecho que hemos sacado durante las practicas, así como las utilidades que podemos realizar gracias a los conocimientos.

## 4-Tecnologías usadas

A continuación, explicaremos el uso de cada una de las tecnologías usadas, haciendo hincapié en la tecnología de Microsoft Azure, la que consideramos más importante en nuestra aplicación.

### a – Microsoft Azure

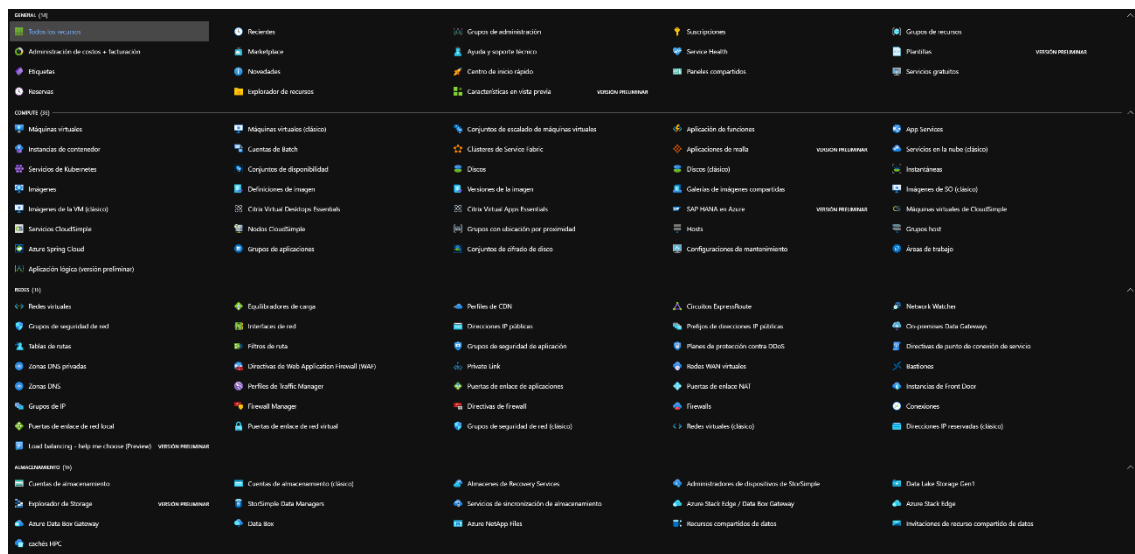
*¿Qué es y para qué sirve?*

Azure es un servicio en nube en el que el usuario paga por uso, es decir, en función de la cantidad de recursos que consumas, pagarás más o podrá llegar a ser gratuito. No todos los servicios son de pago, por ejemplo, en el caso del servicio que usamos para la detección y reconocimiento facial, no tiene ningún coste, en cambio cuando usamos los storages de Azure, si podemos llegar a tener que pagar en el momento en que excedamos el límite de memoria gratuito.

Ofrece una gran cantidad de servicios, desde máquinas virtuales en nube de Linux y Windows con gran capacidad, como bases de datos securizadas y centralizadas. Nos otorga también un almacenamiento en nube de gran capacidad, diferentes recursos de inteligencia artificial y aprendizaje automático (en nuestro caso, usando servicios cognitivos), herramientas de análisis muy útiles en el ámbito empresarial y un largo etcétera que mostraremos a continuación mediante una serie de capturas del portal de Microsoft Azure.

Puede considerarse una herramienta de gran ayuda para la transformación digital en empresas, haciendo intuitivo y más veloz el paso de ciertos mecanismos a un formato digital actualizado.

Cabe destacar también la importancia que tiene Microsoft en el ámbito de la privacidad y seguridad de sus clientes, creando así una mayor confianza en el uso de sus recursos.



*Ilustración 2 Servicios que proporciona Microsoft Azure*

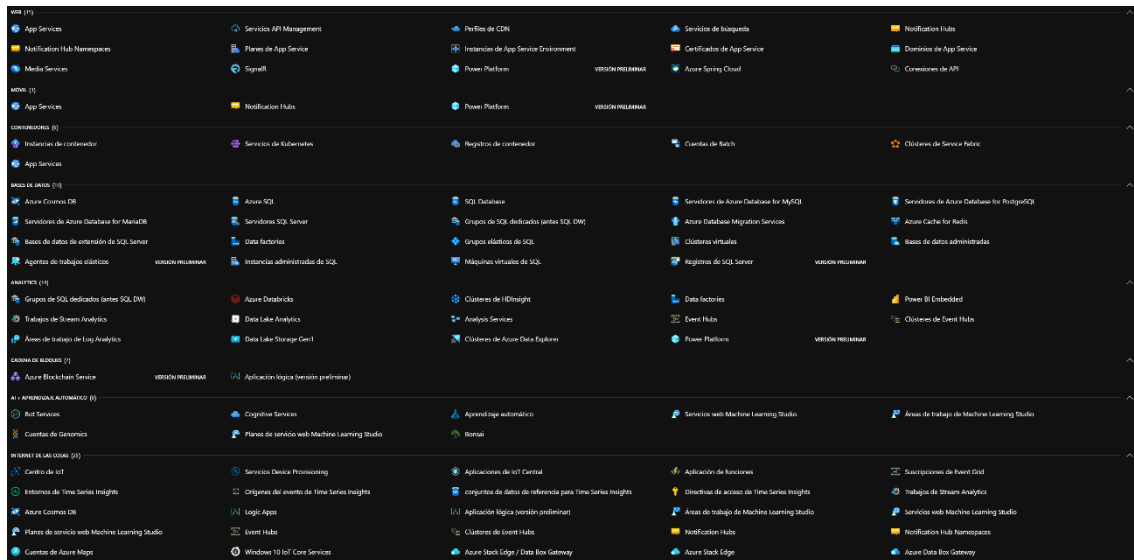


Ilustración 3 Servicios que proporciona Microsoft Azure

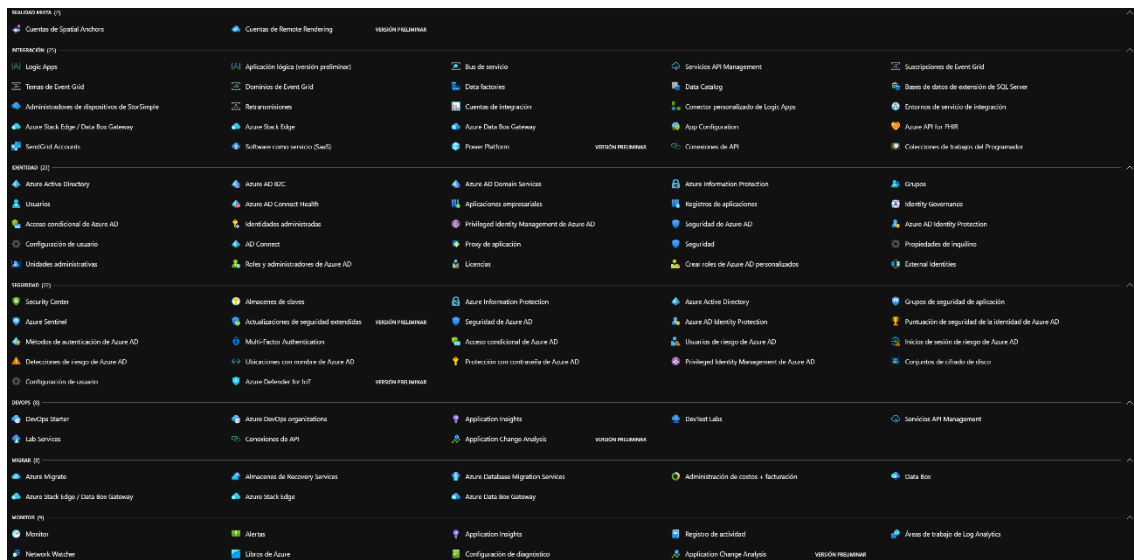


Ilustración 4 Servicios que proporciona Microsoft Azure

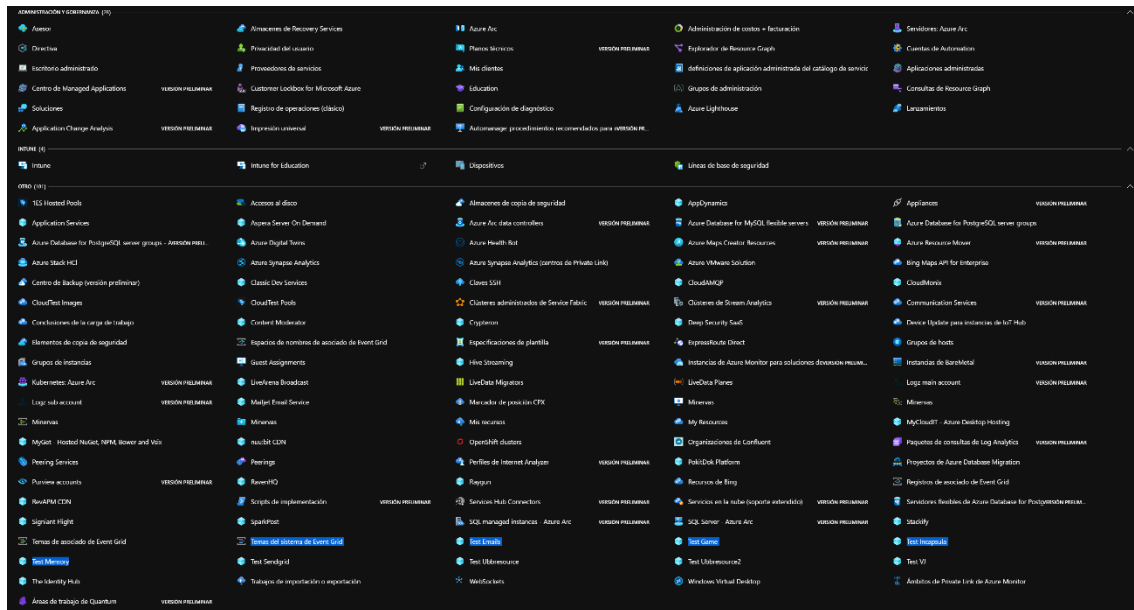


Ilustración 5 Servicios que proporciona Microsoft Azure

Pasemos más en detalle a los servicios que usaremos nosotros de Microsoft Azure: *Cognitive Service (servicios cognitivos)* y *Biblioteca de Azure Storage*.

## SERVICIOS COGNITIVOS

Este servicio, nos permite usar la Inteligencia Artificial en nuestra aplicación sin necesidad de tener unos conocimientos amplios en este ámbito, así como tampoco necesitamos saber respecto al *machine learning* (mecanismo en el que, mediante algoritmos, hacemos “predicciones” de los datos que enviamos), simplemente, tenemos que llamar a la API de Microsoft Azure desde nuestro aplicación y ya podremos usarla.

Detallemos los servicios cognitivos que nos proporciona Microsoft Azure:

-Decisiones: permite tomar decisiones de forma más rápida:

- *Anomaly Detector*: identificar probables problemas de una forma más temprana.
- *Content Moderator*: identifica contenido ofensivo o no deseado.
- *Metrics Advisor*: supervisión de métricas y diagnosticar problemas (es una versión preliminar)
- Permite también crear experiencias personalizadas para cada usuario.



-Lenguaje: consigue significados de textos mal estructurados:

- *Immersive Reader*: ayuda a los lectores a comprender el texto mediante el uso de anotaciones por audio y visuales.
- *Language Understanding*: desarrollar en un lenguaje natural y comprensible en aplicaciones, bots y servicios IoT.
- *QnA Maker*: cree una conversación de preguntas y respuestas sobre sus datos
- *Text Analytics*: detecta sentimientos, frases clave y entidades con nombre.
- *Translator*: detecta y traduce más de 60 lenguajes.

-Habla (*speech*): integra los procesos de habla en aplicaciones y servicios:

- *Speech to text*: transcribe un audio a texto legible.
- *Text to Speech*: convierte texto a audio para interfaces naturales.
- *Speech Translation*: integra traducciones en tiempo real en las aplicaciones.
- *Speaker Recognition*: identifica y verifica a personas hablando mediante el audio.

-Vision: identifica y verifica contenido de imágenes y videos:

- *Computer Vision*: analiza contenido de imágenes y videos.
- *Custom Vision*: personaliza el reconocimiento de imágenes adaptable a las necesidades de negocio.
- *Face*: detecta e identifica personas y emociones en imágenes.
- *Form Recognizer*: extrae texto y pares clave-valor de documentos.
- *Video Indexer*: analiza tanto la parte visual como de audio y nunte su contenido.

Pasemos a entrar en detalle del servicio cognitivo usado en nuestro aplicativo: *Vision Face*, el cual usamos tanto para la detección de las caras en las imágenes, como para la comparación de estas.

### ***Vision: Face***

De este servicio, usaremos dos de todas las funciones que nos presta:

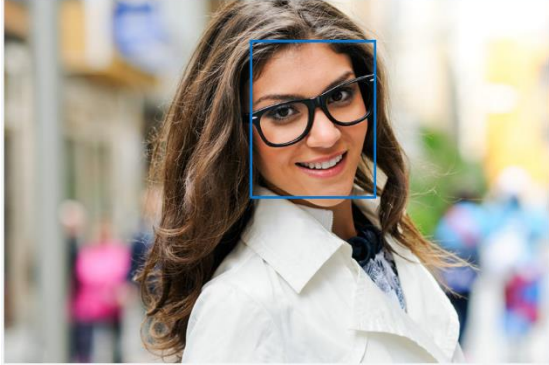
-***Detection***: en esta primera llamada al servicio, nosotros enviamos una imagen al servidor de Azure, para que la analice y detecte una cara, devolviéndonos cierta información de esta (como pueden ser, posiciones de los ojos, nariz boca, si la persona lleva gafas, si tiene pelo o no, etcétera) así como un Id único para cada cara. Este Id, lo reconocerá Azure durante 24h hasta eliminarlo de su base de datos.

Para nuestro aplicativo, el único valor que necesitamos es el Id de la cara detectada en la imagen.

Face detection
Face verification
Perceived emotion recognition

### Face detection

Detect one or more human faces along with attributes such as: age, emotion, pose, smile, and facial hair, including 27 landmarks for each face in the image.



```

Detection result:
detection_02
JSON:
{
  "faceId": "e2c8c617-a501-4bc3-85df-3bb45290a048",
  "faceRectangle": {
    "top": 76,
    "left": 446,
    "width": 226,
    "height": 284
  },
  "faceAttributes": null,
  "faceLandmarks": null
}

```

Ilustración 6 Ejemplo del resultado del servicio Face-Detection

-**Verification:** a continuación, usando el servicio de verificación, lo que hacemos es leer dos, imágenes de las cuales obtenemos un Id para cada una, y enviando estos Id's al servicio de Microsoft Azure Face-Verification para que se encargue de comparar ambas imágenes, devolviéndonos un valor comprendido entre 0 y 1 sobre su porcentaje de similitud que ha encontrado entre ambas imágenes.

### Face verification

Check the likelihood that two faces belong to the same person and receive a confidence score.


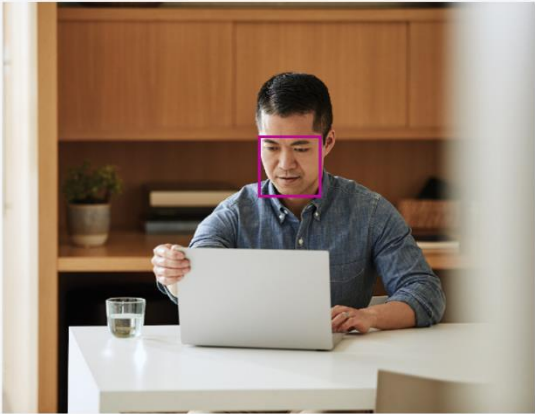



Image URL
Submit
Browse
Image URL
Submit
Browse

Verification result: The two faces belong to the same person. **Confidence is 0.93468.**

Ilustración 7 Ejemplo del resultado del servicio Face-Verification

*Nota: cabe destacar que lo comentado simplemente es a modo introductorio para entender cómo funcionan estos servicios, ya que para nuestro aplicativo, nosotros llamamos a los servicios mediante la API que nos proporciona Microsoft Azure, la cual entraremos más en detalle a la hora de explicar nuestro código, explicando correctamente como pasamos las imágenes para que las reconozca, así como explicaremos como llamamos a los servicios cognitivos de Azure y como tratamos sus valores de respuesta, dados en formato JSON (añadir el punto de donde lo explicaremos).*

## **BIBLIOTECA DE AZURE STORAGE**

Comentaremos brevemente cómo funciona la subida de archivos al Storage de Microsoft Azure en nube, ya que su funcionamiento es intuitivo y similar al de otros servicios de almacenamiento en nube.

Explicaremos brevemente los pasos que realizar para crear un recurso y como crear un contenedor donde almacenar archivos. Lo veremos más en detalle sobre su configuración y su uso en el apartado 6.a, donde comentamos como crear los recursos.

A continuación, dejo un enlace del propio Microsoft para crear una cuenta de almacenamiento en el portal de Microsoft Azure.

<https://docs.microsoft.com/es-es/azure/storage/common/storage-account-create?tabs=azure-portal>

La facilidad que nos aporta el Storage proporcionado por Azure, es la gran cantidad de objetos de datos que almacena (blobs, archivos, colas, tablas y discos) con una fácil accesibilidad a estos a través de peticiones HTTP o HTTPS, así como seguros y con una gran escalabilidad.

## **b- Twilio Service**

Twilio es una plataforma de desarrollo con la que podemos construir aplicaciones de comunicación en nube y sistemas Web, permitiendo agregar de una forma rápida y eficaz servicios de mensajería de voz, videollamadas, mensajes de texto, etcétera mediante APIS.

En nuestro caso, usamos la integración de Twilio en la mensajería de nuestra aplicación, mediante la API de SMS que nos ayudan a enviar y recibir mensajes SMS, así como proporcionándonos un sistema de autenticación basado en mensajes, verificando contraseñas OTP en aplicaciones móviles y web.

## **c- Android**

Hemos decidido usar el Sistema Operativo para móvil Android dada las facilidades para desarrollar software en este que hay, así como en lenguaje en el que se puede programar es Java. Creemos que la capacidad de usuarios que podemos abarcar con esta primera versión “beta” del aplicativo es mayor en este sistema operativo que si hubiéramos hecho el aplicativo en IOS (Sistema Operativo de Apple) considerando así que tendremos un mayor target de usuarios para poder usarla en un primer lanzamiento de esta forma.

Establecemos la versión mínima de uso en la 26, que abarca en torno a un 61% de los dispositivos actualmente comercializados en el mercado mundial.

#### **d- Spring**

Spring es un framework de código abierto pensado para la realización de aplicaciones empresariales con *Java*. De esta definición podemos sacar en claro que su uso orientado a la creación de proyectos grandes orientados a un ambiente de negocios o pensados para satisfacer las necesidades de una empresa, donde tendremos numerosas funcionalidades distintas con opción de ser escalable en caso de que aumente el número de usuarios, productos...

Desde este framework, podemos acceder a nuestra Base de Datos, podemos crear aplicaciones con esquemas clásicos de seguridad o incluso realizar aplicaciones de escritorio.

#### **e- PostgresSql**

Es un sistema de gestión de bases de datos relacional y orientado a objetos.

Permite que a la vez que procesos escriban en una tabla a la par que otros consultan datos de esta sin bloquearse, la cual tiene las siguientes ventajas frente a otros sistemas de bases de datos: restricciones en el dominio, posee integridad referencial, conexión a un sistema de gestión de base de datos.

## **5. Detalles técnicos**

Ahora, vamos a entrar en detalle en cómo está compuesta cada parte de nuestro aplicativo, explicando en detalle, las clases que hacen alusión a los servicios de Microsoft Azure y explicando de una manera menos detallada cada funcionalidad del aplicativo.

Volvemos a recordar que está compuesto de tres “partes” principales: un front-end en Android donde representamos las vistas de cara al usuario, así como recogemos los datos de este para enviárselos al back-end para que los trate, y desde este, pasemos la información necesaria a la base de datos estableciendo los valores pertinentes.

En nuestro back-end, trataremos estos datos, y los analizaremos mediante programación orientada a objetos, pasando ciertos estados a nuestra base de datos.

Por último en la base de datos, tenemos diferentes tablas con información respecto a los empleados, centros etcétera, registrando en esta estados de los empleados.

## I. EXPLICACIÓN DEL CÓDIGO DEL BACK-END

Entrando en detalle en el código de nuestro back-end, comencemos resumiendo brevemente que encontraremos en nuestra aplicación de Spring.

Para organizar nuestro código siguiendo la “*praxis*” que se rigen en el ámbito empresarial, encontramos nuestro aplicativo dividido en los siguientes paquetes:

- Paquete *controller*: en él recogemos las clases donde realizaremos los códigos para crear los controladores, que son las clases encargadas de recoger las llamadas API REST del front-end así como establecer las rutas y llamadas a otras clases del código.
- Paquete *dao*: en él estableceremos los *pojos* (*Plain Old Java Object*) donde crearemos las llamadas a la base de datos en función de la entidad a la que relacione. Por defecto, cada uno de nuestros *pojos* extienden del *CrudRepository*, repositorio ya proporcionado por las librerías de Spring donde ya tenemos métodos predefinidos para hacer consultas a la base de datos como mostrar todos los datos, buscar por un Id, etcétera.
- Paquete *entity*: donde creamos las clases simples de nuestro aplicativo. Establecemos así una serie de características básicas en los atributos de las clases en referencia a entidades que usaremos. Un ejemplo de estas clases para entender cómo funciona, pongamos que es la clase Cliente, donde establecemos valores para el cliente como nombre, apellidos, email, factura, etcétera.
- Paquete *service*: en él creamos los servicios que llaman tanto a nuestras clases de los paquetes *entity*, como del *dao*, para poder hacer uso de estas desde nuestro controlador y así tener un código menos denso y optimizado.
- Paquete *serviciosazure*: dentro de este paquete irán todas las clases relacionadas con la utilización de recursos de Microsoft Azure, es decir, abarca todas las clases relacionadas con el reconocimiento facial.

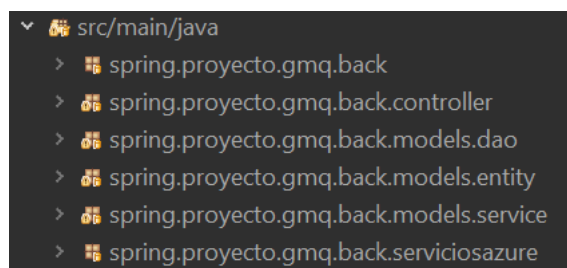


Ilustración 8 Paquetes del back-end

La finalidad de dividir el código en tantos paquetes es tener un código más limpio, donde cada paquete reúne clases con características similares, de forma que, en caso de que haya que cambiar algún apartado específico de algún componente, su cambio no afecte en gran medida al resto de clases, dejando así un código mas optimizado a la par que ordenado. Pasemos pues a entrar en detalle en cada paquete.

### Paquete entity

Comencemos definiendo este paquete para saber que entidades vamos a utilizar en nuestro aplicativo.

Cada una de las clases, vendrá acompañada de dos anotaciones imprescindibles para poder relacionarlas a las columnas que tendremos en la base de datos. La primera es la notación *@Entity*, la cual indica en cada clase que es una entidad relacionada con una base de datos. Luego encontraremos la notación *@Table*, donde especificamos la tabla de la que proviene, indicando dentro de esta notación el nombre de la tabla idéntico al nombre que hemos puesto a la tabla en nuestra base de datos (Ejemplo: *@Table(name = "nombre\_tabla")*). Dentro de cada una de estas clases, debemos definir una serie de variables en función de las columnas que tenemos en nuestra base de datos, y realizar los correspondientes *"getters & setters"* de cada una de estas variables, para poder, tanto recoger, como establecer nuevos datos dentro de nuestra tabla.

Dentro de este paquete, situaremos cinco clases que definirán a cada una de nuestras entidades:

- Centros: en ella tendremos el nombre del centro, un número identificativo (será el Id en la Base de Datos), la latitud y la longitud del centro para la posterior geolocalización del empleado.
- Departamentos: encontraremos el número de departamento (corresponde con el Id de la Base de Datos) así como el nombre de departamento.
- Fichajes: en esta clase encontramos como clave primaria o Id el número de fichaje, acompañado del id del empleado (será la clave foránea para poder relacionar cada fichaje a un empleado específico), la fecha del fichaje y el estado de este.
- Nominas: volvemos a encontrar el número de nómina como clave primaria de esta, el id de empleado como clave foránea para asociarla, una URL que será la perteneciente a la dirección de descarga del archivo PDF de la nómina y la fecha, para saber a qué mes pertenece cada nómina.
- Empleados: nuestro Id relacionado a la base de datos será el Id de empleado, acompañado de su nombre, apellidos, dirección, número de teléfono, encontramos dos claves foráneas para asociarlo a un centro y departamento específicos, que serán el número de centro y el número de departamento respectivamente. Encontramos también una URL relacionada con su imagen de empleado y un token, que será el token enviado por SMS.

A continuación, a modo de ejemplo, dejamos una imagen de la clase Empleados para tomarla como modelo y ver como se realizan.

```
package spring.proyecto.gmq.back.models.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
```

Ilustración 9 Librerías Empleados.java

```
@Entity
@Table(name = "empleados")
public class Empleados {

    @Id
    @Column(name="id_empleado")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id_empleado;

    private String nombre;

    private String apellidos;

    private String direccion;

    private String telefono;

    private int n_departamento;

    private int n_centro;

    private String url_storage;

    private int token;
```

Ilustración 10 Variables Empleados.java

```
//Getters & Setters
public Long getId_empleado() {
    return id_empleado;
}

public void setId_empleado(Long id_empleado) {
    this.id_empleado = id_empleado;
}

public String getNombre() {
    return nombre;
}

public void setNombre(String nombre) {
    this.nombre = nombre;
}

public String getApellidos() {
    return apellidos;
}

public void setApellidos(String apellidos) {
    this.apellidos = apellidos;
}

public String getDireccion() {
    return direccion;
}
```

Ilustración 11 Getters&Setters Empleados.java

```
public void setDireccion(String direccion) {
    this.direccion = direccion;
}

public String getTelefono() {
    return telefono;
}

public void setTelefono(String telefono) {
    this.telefono = telefono;
}

public int getN_departamento() {
    return n_departamento;
}

public void setN_departamento(int n_departamento) {
    this.n_departamento = n_departamento;
}

public int getN_centro() {
    return n_centro;
}

public void setN_centro(int n_centro) {
    this.n_centro = n_centro;
}

public String getUrl_storage() {
    return url_storage;
}
```

Ilustración 12 Getters&Setters Empleados.java

```
public void setUrl_storage(String url_storage) {
    this.url_storage = url_storage;
}

public int getToken() {
    return token;
}

public void setToken(int token) {
    this.token = token;
}
```

Ilustración 13 Getters&Setters Empleados.java

## Paquete dao

En este paquete vamos a definir las interfaces pertinentes para relacionar, las entidades mencionadas con nuestra base de datos mediante estas interfaces, y así comprobar que los datos que buscamos o que queremos insertar en esta es posible. Para ello, estas interfaces extenderán de la interfaz *CrudRepository* (indicando en esta el nombre de la entidad relacionada a la tabla de la base de datos y el tipo de dato de la clave primaria), la cual nos permite realizar cualquier petición CRUD genérica sin necesidad de realizar ninguna línea de código. Esta interfaz nos aporta métodos básicos para hacer peticiones a la base de datos como mostrar todos los datos de esta (*findAll*), buscar por Id (*findById*) y otras tantas. Cabe destacar también que todas estas interfaces vendrán acompañadas de la anotación *@EnableJpaRepositories*, el cual permite a las interfaces heredar del *CrudRepository* y que “comprenda” los métodos que esta provee.

A parte de las que vienen predefinidas, en nuestro caso, necesitamos para ciertas interfaces hacer consultas específicas las cuales crearemos nosotros. Para realizarlas utilizaremos la notación *@Query*, realizando una consulta común de base de datos indicando en lugar del nombre de la tabla, el nombre de la entidad de la que obtendremos el dato.

Las interfaces que tendremos en este paquete serán las mismas que las entidades, con distinto nombre, indicando por praxis una vez más que todas empiezan por una “I” para indicar que son interfaces, y acaban con la terminación “dao” para indicar que son pertenecientes a este paquete.

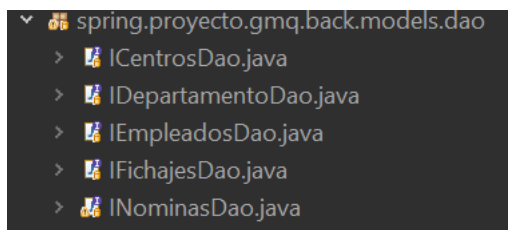


Ilustración 14 Clases del paquete dao



A continuación, mostramos las interfaces de Fichajes, Nominas y Empleados, las cuales tienen algunos ejemplos en los que usamos consultas propias en lugar de heredadas.

```
@EnableJpaRepositories
public interface IFichajesDao extends CrudRepository<Fichajes, Long>{

    @Query("select u from Fichajes u where u.id_empleado=?1")
    public List<Fichajes> findAllFichajesEmpleado(Long id);

    @Query("select u from Empleados u where u.telefono=?1")
    public Empleados buscarPorTfno(String telefono);
}
```

Ilustración 17 IFichajesDao.java

```
@EnableJpaRepositories
public interface IEmpleadosDao extends CrudRepository<Empleados, Long>{

    @Query("select u from Empleados u where u.telefono=?1")
    public List<Empleados> findByTelefono(String telefono);
}
```

Ilustración 16 IEmpleadosDao.java

```
import java.util.Date;
import java.util.List;

import org.springframework.data.jpa.repository.Query;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.data.repository.CrudRepository;

import spring.proyecto.gmq.back.models.entity.Nominas;

@EnableJpaRepositories
public interface INominasDao extends CrudRepository <Nominas, Long>{

    //Necesitamos una query para mostrar todas las nominas de un empleado
    @Query("select u from Nominas u where u.id_empleado=?1")
    public List<Nominas> findNominaByNumEmpleado(int id);

    //Metodo para seleccionar la url de la bbdd
    @Query("select u from Nominas u where to_char(u.fecha, 'MM')=?1")
    public Nominas findNominaMes(String fecha);
}
```

Ilustración 15 INominasDao.java

## Paquete service

En este paquete, encontramos todas las clases de “servicios” las cuales implementan métodos de llamada a las clases del paquete *dao* para así tratar los datos que recibimos desde el front-end. Estas clases, serán las que utilizemos desde nuestras clases del paquete *controller* para realizar métodos con los datos recibidos.

Como forma de optimizar el código una vez más, dividimos los códigos de cada clase en interfaces, donde especificamos el nombre de cada método que vamos a usar, y clases que implementan dichas interfaces, donde tendremos la lógica de cada una de estas funciones. Cabe destacar que todas las clases llevarán incluida la notación *@Service* para indicar de esta forma que son servicios.

En este caso, encontraremos las siguientes clases dentro de nuestro paquete:

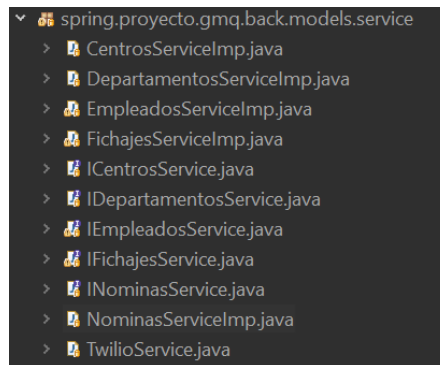


Ilustración 18 Clases del paquete service

A continuación, mostraremos como ejemplo la clase de *FichajesServiceImp.java* así como la interfaz correspondiente, *IFichajesService.java* para observar cómo funcionan:

```
import java.util.List;

@Service
public interface IFichajesService {

    public List<Fichajes> findAllFichajesEmpleado(Long id);

    public Fichajes guardarFichaje(Fichajes fichaje);
}
```

Ilustración 19 Interfaz IFichajesService.java

```
@Service
public class FichajesServiceImp implements IFichajesService{

    @Autowired
    IFichajesDao dao;

    @Override
    public List<Fichajes> findAllFichajesEmpleado(Long id) {
        // TODO Auto-generated method stub
        return dao.findAllFichajesEmpleado(id);
    }

    public Empleados buscarPorTfno(String telefono) {
        return dao.buscarPorTfno(telefono);
    }

    /*
     * Metodo para comprobar las caras
     */
    public Boolean comprobarCaras(String imagen1, String imagen2) {
        Map<String, Object> response = new HashMap<>();

        String respuesta = CompararCaras.returnIdentical(imagen1, imagen2);
        if (Double.parseDouble(respuesta) < 0.8) {
            response.put("Respuesta: ", "Estado -> no son la misma persona");
            return false;
        } else {
            response.put("Respuesta", "Estado -> son la misma persona");
            return true;
        }
    }

    @Override
    public Fichajes guardarFichaje(Fichajes fichaje) {
        // TODO Auto-generated method stub
        return dao.save(fichaje);
    }
}
```

Ilustración 20 Clase FichajesServiceImp.java

Enseñemos también cómo funciona el servicio de Twilio, para entender su funcionamiento:

```
@Service
public class TwilioService {
    // Find your Account Sid and Token at twilio.com/console
    public static final String ACCOUNT_SID = "ACdaad24f7337c3c94ac635b8f747c49f5";
    public static final String AUTH_TOKEN = "d2fa53eb1d97386bfe8a3b2769695144";
    static String fecha = "dd-MM-yyyy";
    static SimpleDateFormat formatear = new SimpleDateFormat(fecha);
    /*----- SMS -----*/
    public static void sms(Empleados empleado) {

        Twilio.init(ACCOUNT_SID, AUTH_TOKEN);
        Message message = Message.creator(
            new com.twilio.type.PhoneNumber("+34"+empleado.getTelefono()),
            new com.twilio.type.PhoneNumber("+14758897855"),
            "Su contraseña es: "+empleado.getToken()
        ).create();

        System.out.println(message.getSid());
    }
}
```

Ilustración 21 Clase TwilioService

### Paquete controller

En este paquete tendremos tres clases con los métodos pertinentes para, desde el front, obtener ciertos parámetros que trataremos dando así ciertos valores a nuestros empleados. Es por ello, que necesitaremos, un controlador para las nóminas, otro para realizar los fichajes, y otro para el empleado, donde pueda visualizar información suya, loggarse, etcétera.

Todas estas clases vienen con dos anotaciones, `@RestController`, para indicar que es un controlador de una petición *REST*, y la notación `@RequestMapping`, que nos sirve para crear nuestra ruta principal de acceso a los métodos del controlador, un ejemplo de como debe generarse este nombre para la ruta es: `@RequestMapping("/nombre_de_la_ruta")`.

En función de que método usemos, debemos indicar antes de cada uno la notación pertinente a lo que realizaremos, los métodos más conocidos para peticiones REST: *GET*, *POST*, *PUT* y *DELETE*. En nuestro caso, los métodos son siempre GET y POST, ya que simplemente queremos u obtener valores, o introducirlos, pero en ningún momento modificamos valores de cada usuario. Cuando situemos estas anotaciones en los métodos, debemos indicar una ruta en cada una (no es obligotario, pero para no tener ningún conflicto en la ruta lo hemos creído conveniente), y en caso de que pasemos algún parámetro en la ruta, debemos indicarlo con llaves, por ejemplo: `@GetMapping("/nombre_ruta/{parámetro}")`.

Este parámetro, dentro del método, deberemos indicar el tipo de valor y añadirle la notación `@PathVariable` para indicar que pertenece a la ruta. Para los valores que enviamos en el cuerpo, la notación que usaremos es `@RequestBody`, para indicar que necesitamos obtener un cuerpo en la respuesta de la petición.

A continuación, como ejemplo, enseñaremos como son nuestras clases controller. Creemos conveniente enseñar las tres ya que, junto a las clases relacionadas con Microsoft Azure, consideramos que son las más importantes del código, ya que son aquellas que reciben los datos de Android, los tratan, y devuelven una respuesta a Android, lo que implica que una mala realización de estas influye en una mala ejecución del código.

```
@RestController
@RequestMapping("/nominas")
public class NominasController {

    @Autowired
    INominasService nomService;
    @Autowired
    INominasDao dao;
    //Metodo para mostrar todas las nominas de un empleado
    @GetMapping("/verNominas/{id}")
    public List<Nominas> verNominas(@PathVariable int id) {
        return nomService.findNominaByNumEmpleado(id);
    }

    //Metodo para descargar nomina por mes
    @GetMapping("/descargarNomina/{hola}")
    public String descargarUltimaNomina(@PathVariable String hola) {
        Nominas nom = new Nominas();

        nom = nomService.findNominaMes(hola);

        String url = nom.getUrl();
        System.out.println(url);
        return url;
    }
}
```

Ilustración 22 NominasController.java

```
@RestController
@RequestMapping("/fichajes")
public class FichajesController {

    @Autowired
    IFichajesService fichService;
    @Autowired
    FichajesServiceImp serviceImp;
    @Autowired
    CentrosServiceImp centroService;

    @GetMapping("/verFichajes/{id}")
    public List<Fichajes> listarFichajes(@PathVariable Long id) {
        return fichService.findAllFichajesEmpleado(id);
    }

    @PostMapping("/hacerFichaje/{telefono}/{latitud}/{longitud}")
    public boolean comprobarCera (@PathVariable String telefono,
        @PathVariable String latitud,
        @PathVariable String longitud,
        @RequestBody String imagen){

        Map<String, Object> response = new HashMap<>();
        //Buscamos primero el empleado por telefono
        Empleado emp = serviceImp.buscarPorTfno(telefono);
        //Buscamos el centro mediante el empleado
        Centros centro = centroService.findCentroById((long) emp.getId_centro());
        //Buscamos la latitud y la longitud
        double lat = centro.getLatitude();
        double longi = centro.getLongitude();
        //Creamos un nuevo fichaje para establecer a true o false
        Fichajes fichaje = new Fichajes();

        Fichajes fichajeNuevo = null; //Nuevo fichaje que guardaremos

        System.out.println(emp.getId_storage());

        //Almacenamos la imagen en un String
        String imagen1 = emp.getId_storage();

        boolean result = serviceImp.comprobarCeras(imagen1, imagen);

        Date date = new Date();
        //date.getTime();
        java.sql.Timestamp fecha = new java.sql.Timestamp(date.getTime());

        if (result == true && lat == Double.parseDouble(latitud) && longi == Double.parseDouble(longitud)) {

            System.out.println("\n\nLas caras son iguales");
            fichaje.setFecha(fecha);
            fichaje.setEmpleado(emp.getId_empleado());
            fichaje.setEstado(true);
            //System.out.println(d);

            fichajeNuevo = fichService.guardarFichaje(fichaje);
            response.put("Resultado: ", fichaje.isEstado());
            return true;

        } else {
            System.out.println("Ha entrado aqui...");
            fichaje.setFecha(fecha);

            fichaje.setEmpleado(emp.getId_empleado());
            fichaje.setEstado(false);
            fichajeNuevo = fichService.guardarFichaje(fichaje);
            return false;
        }
    }
}
```

Ilustración 23 FichajesController.java

```

@RestController
@RequestMapping("/empleados")
public class EmpleadosController {

    @Autowired
    EmpleadosServiceImp empService;

    @Autowired
    IEmpleadosService service;

    @Autowired
    ICentrosService centService;

    @Autowired
    IDepartamentosService depService;

    //Metodo para listar empleados
    @GetMapping("/listar")
    public List<Empleados> listar() {
        return empService.listar();
    }

    //Metodo para listar un empleado por su id
    @GetMapping("/listar/{id}")
    public ResponseEntity<> show (@PathVariable Long id) {

        Map<String, Object> response = new HashMap<>();
        Empleados empleado = service.findById(id); //Buscamos al empleado por su id
        Centros centro = centService.findCentroById((long) empleado.getN_centro()); //Buscamos el centro
        Departamentos dep = depService.findDepById((long) empleado.getN_departamento()); //Buscamos la mis

        /*
        try {
            empService.findById(id);
        } catch (DataAccessException e){
            response.put("mensaje", "Error al realizar la consulta en la base de datos");
            return new ResponseEntity<Map<String, Object>>(response, HttpStatus.INTERNAL_SERVER_ERROR);
        }*/

        if (empleado == null) {
            response.put("mensaje", "La visita con ID: " + id.toString() + " no existe en la base de datos");
            return new ResponseEntity<Map<String, Object>>(response, HttpStatus.NOT_FOUND);
        } else {
            response.put("Ver info: ", empleado);
            response.put("Bienvenido de nuevo: ", empleado.getNombre() + " " + empleado.getApellidos());
            response.put("Tu centro es: (centro numero: " + centro.getN_centro() + ") "
                    + centro.getNombre() + " ", empleado.getN_centro() + " ", empleado.getDireccion());
            response.put("Departamento: ", dep.getNombre());

            return new ResponseEntity<Map<String, Object>>(response, HttpStatus.OK);
        }
        //return new ResponseEntity<Empleados>(empleado, HttpStatus.OK);
    }

    //Metodo para solicitar el token
    @GetMapping("solicitar/{telefono}")
    public List<Empleados> solicitarPos(@PathVariable String telefono){
        return service.findById(telefono);
    }

    @PostMapping("token/{telefono}")
    public Boolean solicitarToken(@PathVariable String telefono){
        return service.solicitarToken(telefono);
    }

    //Metodo para solicitar el token
    @PostMapping("login/{telefono}")
    public Boolean getLogin (@PathVariable String telefono, @RequestBody String token){
        System.out.println("he entrado aqui");
        return service.getLogin(telefono, token);
    }
}

```

Ilustración 24 EmpleadosController.java

## Paquete serviciosazure

En este paquete recogemos todas las clases que utilizamos para la detección de caras, así como la comparación de ellas. Cabe destacar que en nuestras pruebas previas, también creamos una clase para subir al Storage de Azure la imagen desde el dispositivo, pero decidimos quitarlo debido a que, a modo empresarial, es preferible que la primera imagen la almacenen desde la propia empresa para evitar que el empleado realice la imagen con cualquier objeto como gorra o mascarilla, lo que a posteriori, puede dificultarnos la probabilidad de que detecte que son la misma persona.

Por ello, expliquemos para tener una idea general que clases vamos a encontrar dentro de esta carpeta:

- **GuardarImagenStorage:** esta clase, la vamos a dejar a modo ejemplo para mostrar cómo podríamos subir la imagen al storage desde la aplicación Android. Pese a ello, hemos decidido que la primera foto de nuestro empleado será almacenada directamente su URL en la base de datos, ya que creemos conveniente que esta primera imagen no la maneje el usuario, para mejorar la veracidad de que cada empleado es quien dice ser.
- **DetectarCaraArray:** es la clase encargada de detectar la cara desde un array de bytes. Esta clase, la usamos, tanto para comprobar al hacer cada foto, que se está reconociendo al individuo asegurándonos así que siempre se pueda detectar una

cara, así como, a la hora de comparar las caras, la que nos analice la segunda imagen.

- **DetectarCaraUrl:** es la clase encargada de detectar la cara desde una URL, URL que pertenecerá a la imagen del usuario que hemos almacenado en el storage.
- **CompararCaras:** esta clase, se encarga de comparar ambas caras y devolviéndonos el resultado de si son idénticas, o no.

Pasemos a definir más en detalle cómo funciona cada clase:

### DetectarCaraArray

En esta clase, tendremos un método llamado *DetectarCara*, público, que recibe como parámetro un String con la imagen recibida desde el *front-end* de la aplicación (Android en este caso).

Primero de todo, vamos a mostrar las librerías usadas para la clase, pertenecientes a **apache** para la realización de las peticiones http y también a **JSON** para tratar la respuesta de Azure, que viene dada en este formato.

```
import java.net.URI;
import org.apache.http.HttpEntity;
import org.apache.http.HttpResponse;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.client.utils.URIBuilder;
import org.apache.http.entity.StringEntity;
import org.apache.http.impl.client.HttpClientBuilder;
import org.apache.http.util.EntityUtils;
import org.json.JSONArray;
import org.json.JSONObject;
```

Ilustración 25 Librerías de la clase

Dentro de nuestro método *DetectarCara*, transformaremos la imagen recibida como *String* a un array de bytes (*byte[]*), que es el formato que reconoce el servicio de Azure para leer la imagen. A continuación, crearemos unas variables necesarias para la realización del método, las cuales son:

- *LlaveAzure:* en ella recogemos la clave privada del servicio creado de Azure para el reconocimiento facial (este será del tipo *String*).
- *EndpointAzure:* recogemos también el endpoint de nuestro servicio, que será una URL de llamada a este (este será del tipo *String*).
- *jsonString:* creamos una variable del tipo *String* donde recogeremos el JSON de Azure en este formato para poder “tratarlo” y obtener así el id.
- *valorIdImagen1:* variable del tipo *String* que retornará el método para devolvernos el id que ha asignado el servicio de Azure a la cara reconocida en nuestra imagen.
- *httpClient:* creamos esta variable del tipo *HttpClient* donde creamos un cliente que nos permita enviar la solicitud al servidor de Azure y obtener respuesta de este, en formato *http*.

```

public class detectarCaraArray {

    /*
     * Crearemos un metodo donde llamamos a la api de Azure para detectar caras,
     * retornando este un String donde almacenamos la Id de la imagen...
     */

    public static String DetectarCara(String imagen) {
        byte[] Imagen; //Variable para transformar la imagen obtenida como string a array

        //Parseamos la imagen de String a un array de bytes
        Imagen = javax.xml.bind.DatatypeConverter.parseBase64Binary(imagen);

        //Variables de azure, guardamos las credenciales de azure
        final String LlaveAzure = "e4ee36742e6548a3a08689f6b3e9c73b"; //Clave de Azure...
        final String EndpointAzure = "https://recfacialazure.cognitiveservices.azure.com"; //endpoint de Azure...

        String jsonString = null; //Variable para recoger el JSON que devuelve la api de Azure

        String valorIDImagen1 = null; //Variable para recoger el id de la imagen

        HttpClient httpClient = HttpClientBuilder.create().build();
        //System.out.println("Hemos creado HttpClient...");
    }
}

```

Ilustración 26 Clase detectarCaraArray. Método y variables.

Una vez creadas, dentro de un try, comenzamos nuestra llamada a la API de Azure. Crearemos una variable llamada *builder* del tipo *UriBuilder* donde construimos la URL para la petición del servicio, pasando a esta variable nuestra variable *EndPointAzure* y añadiéndole al final de esta la notación *"/face/v1.0/detect"*, la cual le indica que del recurso *face* de Azure, lo que necesitamos usar en este caso es su método *detect* para que detecte la cara de la imagen.

Una vez creada la variable, establecemos los parámetros de la API, configurando, el modelo de detección que usaremos de Azure (en nuestro caso, es irrelevante usar uno que otro ya que solo necesitamos el id y no obtener información de las características del usuario, asique cogemos el modelo *detection\_01*), establecemos también como true que retorne un Id y por último establecemos el modelo de reconocimiento, para así poder usar luego la comparación de caras (*recognition\_03*).

Ya establecidos los parámetros, creamos la URI y, creamos una nueva variable llamada *request*, la cual será la respuesta http que vamos a hacer mediante un método *POST*. Falta también establecer las cabeceras de esta respuesta, indicando que el objeto que vamos a enviar es del tipo *octet-stream* (este objeto será nuestro array de bytes) y pasando también nuestra clave de acceso al servicio de Azure. De esta forma, hemos creado la petición API REST para llamar a Azure.

```

//REQUEST parametros

builder.setParameter("detectionModel", "detection_01");
builder.setParameter("returnFaceId", "true");
builder.setParameter("returnPersistedFaceId", "true");
//builder.setParameter("returnFaceLandmarks", "true");
//builder.setParameter("returnFaceAttributes", "age, gender, facialhair, emotion");
builder.setParameter("recognitionModel", "recognition_03");
//builder.setParameter("returnRecognitionModel", "true");
//builder.setParameter("returnFaceListId", "true");

System.out.println("Hemos establecido que devuelva el ID de la cara...");

//Preparamos URI para la llamada API REST
URI uri = builder.build();

HttpPost request = new HttpPost(uri);

System.out.println("Hemos enviado llamada 1 de API REST...");

//RequestHeader
request.setHeader("Content-Type", "application/octet-stream");
request.setHeader("Ocp-Apim-Subscription-Key", LlaveAzure);

```

Ilustración 27 Clase detectarCaraArray. Creación de la petición API REST



Ahora, tenemos que pasar la imagen que hemos guardado en el array de bytes a Azure, para ello, tenemos que guardar la imagen en una variable del tipo *Entity* de las librerías de apache donde la guardamos, y pasamos mediante un método de la respuesta donde creamos esta *Entity*.

Ahora hemos de ejecutar la API, guardando su respuesta en una variable del tipo *HttpResponse* y acto seguido obtenemos la *Entity* de esta respuesta.

```
ByteArrayEntity reqEntityPrueba = new ByteArrayEntity(Imagen);
request.setEntity(reqEntityPrueba);
System.out.println("Se esta leyendo la imagen 2...");

//Ejecutamos el API REST y obtenemos la respuesta
HttpResponse responde1 = httpClient.execute(request);
HttpEntity entity1 = responde1.getEntity();

System.out.println("Hemos obtenido la priemra respuesta de la imagen 2");
```

Ilustración 28 Clase *detectarCaraArray*. Obteniendo respuesta de la API de Microsoft Azure

Cuando por fin obtenemos respuesta, procedemos a tratar el JSON que nos habrá devuelto, lo haremos dentro de una sentencia *if* para comprobar que esta sea distinta de *null*. Lo primero que realizaremos será almacenar esta respuesta en la variable *jsonString* que habíamos creado previamente, pudiéndola así hacer legible en java. Al tratarse de un array nuestro JSON, creamos una variable del tipo *JSONArray* donde pasamos este String para que quede como un objeto JSON del tipo array. De él, tras hacer varias pruebas con el modelo de detección facial, sabemos que el primer campo que siempre muestra será el id de la cara, luego, seleccionamos el campo 0 del array para guardar el valor de este, siendo el valor que retornamos después.

Tanto dentro de nuestro catch como fuera del try y este, retornamos *null* para indicar que no se ha detectado ninguna cara en la imagen.

```
if (entity1 != null)
{
    System.out.println("REST Response:\n");
    jsonString = EntityUtils.toString(entity1).trim();

    JSONArray ArrayJson = new JSONArray(jsonString);
    JSONObject obj = ArrayJson.getJSONObject(0);
    valorIDImagen1 = obj.getString("faceId");

    return valorIDImagen1;
}

} catch (Exception e) {
    e.printStackTrace();
    return null;
}

return null;
}
```

Ilustración 29 Clase *detectarCaraArray*. Recogiendo el id de la imagen del JSON



## DetectarCaraURL

Esta clase, actúa prácticamente igual que *DetectarCaraArray*, cambiando solamente, la imagen, que esta vez la pasamos directamente la URL de la imagen que tenemos almacenada en el storage convertida en formato JSON, y también cambiamos la cabecera de la respuesta, que en lugar de ser del tipo *octet-stream*, indicamos que es del tipo JSON (*"application/json"*). Cabe destacar que también usaremos las mismas librerías.

```
import java.net.URI;

import org.apache.http.HttpEntity;
import org.apache.http.HttpResponse;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.client.utils.URIBuilder;
import org.apache.http.entity.StringEntity;
import org.apache.http.impl.client.HttpClientBuilder;
import org.apache.http.util.EntityUtils;
import org.json.JSONArray;
import org.json.JSONObject;

public class detectarCaraURL {

    /*
     * Creamos un método para detectar la cara de la imagen almacenada en el storage,
     * retornando de esta el id.
     * Es idéntica a la clase de detectarCaraArray, diferenciando que aquí en vez de
     * pasar un octet-stream, pasamos un application-json para que reconozca
     * la url.
     * Le pasaremos como parametro la url de la imagen almacenada en la bbdd en lugar de una imagen
     */

    public static String DetectarCaraUrl (String rutaImagen) {
        //Variables de azure, guardamos las credenciales de azure
        final String LlaveAzure = "e4ee36742e6548a3a08689f6b3e9c73b"; //Clave de Azure...
        final String EndpointAzure = "https://recfacialazure.cognitiveservices.azure.com"; //endpoint de Azure...

        /*Variable donde convertimos la ruta de la imagen en formato JSON
        * formato legible para la api de Azure...
        */
        String Imagen = "{ \"url\": \"" + rutaImagen + "\" }";

        String jsonString = null; //Variable para recoger el JSON que devuelve la api de Azure

        String valorIDImagen1 = null; //Variable donde recogemos el id de la imagen

        //Creamos un HttpClient
        HttpClient httpClient = HttpClientBuilder.create().build();
        //System.out.println("Hemos creado HttpClient...");
    }
}
```

Ilustración 30 Clase detectarCaraURL. Librerías, métodos y variables.

```

try {
    URIBuilder builder = new URIBuilder(EndpointAzure + "/face/v1.0/detect");
    //System.out.println("Hemos creado un builder con que encuentre la cara en la URL imagen...");

    //REQUEST parametros
    builder.setParameter("detectionModel", "detection_01");
    builder.setParameter("returnFaceId", "true");
    builder.setParameter("returnPersistedFaceId", "true");
    //builder.setParameter("returnFaceLandmarks", "true");
    //builder.setParameter("returnFaceAttributes", "age, gender, facialhair, emotion");
    builder.setParameter("recognitionModel", "recognition_03");
    //builder.setParameter("returnRecognitionModel", "true");
    //builder.setParameter("returnFacelistId", "true");

    System.out.println("Hemos establecido que devuelva el ID de la cara...");

    //Preparamos URI para la llamada API REST
    URI uri = builder.build();
    HttpPost request = new HttpPost(uri);

    System.out.println("Hemos enviado llamada 1 de API REST...");

    //RequestHeader
    request.setHeader("Content-Type", "application/json");
    request.setHeader("Ocp-Apim-Subscription-Key", llaveAzure);

    //Establecemos el tipo de valor a devolver
    StringEntity reqEntity = new StringEntity(Imagen);
    request.setEntity(reqEntity);
    System.out.println("Se esta leyendo la imagen1...");

    //Ejecutamos el API REST y obtenemos la respuesta
    HttpResponse response1 = httpClient.execute(request);
    HttpEntity entity1 = response1.getEntity();
    System.out.println("Hemos obtenido la priemra respuesta de la imagen 1 (URL)");

    if (entity1 != null)
    {
        // Format and display the JSON response.
        System.out.println("REST Response:\n");
        jsonString = EntityUtils.toString(entity1).trim();

        JSONArray jsonArray = new JSONArray(jsonString);
        JSONObject obj = jsonArray.getJSONObject(0);

        valorIDImagen1 = obj.getString("faceId");

        return valorIDImagen1;
    }

} catch (Exception e) {
    e.printStackTrace();

    return null;
}

return null;
}

```

Ilustración 31 Clase detectarCaraURL.  
Petición de la API REST y recogiendo el  
Id de la imagen.

## CompararCaras

En esta clase, vamos a realizar la comparación entre ambas caras mediante un método que hemos llamado *returnIdetical*, al cual le pasamos dos parámetros del tipo *String*, que serán las imágenes, siendo la primera la url del storage de azure que obtenemos de la base de datos, y la segunda, la imagen que capturamos en Android.

Volveremos a usar las mismas librerías que hemos usado en las dos clases anteriores, ya que el método para comparar ambas caras mediante la petición API REST es similar.

Como variables, vamos a tener dos del tipo *String* que serán los Id's de las imágenes, llamadas *id1* e *id2* respectivamente. También, tendremos las variables *imagen1* e *imagen2* que son del tipo *detectarCaraURL* y *detectarCaraArray* respectivamente, para, desde los parámetros recibidos, llamar a sus métodos para retornar los ids.

Esta vez, cuando creemos la URI, debemos indicar al final de ella que es del tipo *"/face/v1.0/verify"* en lugar de detect como en las clases anteriores. Nuestras cabeceras, seguirán siendo del tipo *"application/json"* ya que vamos a pasar un JSON con ambos Id's.

A continuación, recorreremos el array del JSON para recoger el valor que queremos obtener, ya que en dicho array tendremos un valor del tipo *boolean* llamado *itsIdentical* que nos comunica si las caras comparadas, las considera iguales o no, y a parte, el valor que nosotros queremos coger, *confidence*, que será del tipo *double* y nos remite un valor entre 0 y 1 del “porcentaje” de parecido que encuentra entre las caras, siendo 0 el 0% y 1 el 100%.

Retornamos este valor para reconocer si son la misma persona o no. Esta verificación, la hacemos desde la clase *FichajesServiceImp*, la cual podemos ver en la *Ilustración 20*. A continuación, mostramos el código de la clase, así como un pequeño fragmento de la consola para ver la respuesta.

```
import java.net.URI;

import org.apache.http.HttpEntity;
import org.apache.http.HttpResponse;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.client.utils.URIBuilder;
import org.apache.http.entity.StringEntity;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.util.EntityUtils;
import org.json.JSONArray;
import org.json.JSONObject;

public class CompararCaras {

    public static String id1, id2; //Variables donde recoger los string

    //Variable para reconocer la imagen de la url
    public static detectarCaraURL imagen1;

    //Variable para reconocer la imagen del array
    public static detectarCaraArray imagen2;

    //Metodo para comparar las caras
    public static String returnIdentical(String image1, String image2) {
```

*Ilustración 32 Clase CompararCaras. Librerías y variables*

```
//Metodo para comparar las caras
public static String returnIdentical(String image1, String image2) {

    HttpClient httpClient = HttpClient.createDefault();

    try
    {
        URIBuilder builder = new URIBuilder("https://recfacialazure.cognitiveservices.azure.com/face/v1.0/verify");

        URI uri = builder.build();
        HttpPost request = new HttpPost(uri);
        request.setHeader("Content-Type", "application/json");
        request.setHeader("Ocp-Apim-Subscription-Key", "e4ee36742e6548a3a08689f6b3e9c73b");

        id1 = detectarCaraURL.DetectarCaraUrl(image1);
        System.out.println("id1: "+id1);
        id2 = detectarCaraArray.DetectarCara(image2);
        System.out.println("id2: "+id2);
        StringEntity reqEntity =
            new StringEntity(
                "{ \"faceId1\": \""+id1+"\" ,\"faceId2\": \""+id2+"\"}"
            );

        /*
        * eca78dde-bc13-4fcb-b470-928720021fb9
        * cdded5e5-528f-4fc0-aa13-7bbaaa7e76dd
        */
        request.setEntity(reqEntity);

        HttpResponse response = httpClient.execute(request);
        HttpEntity entity = response.getEntity();

        if (entity != null)
        {
            String jsonString = EntityUtils.toString(entity).trim();
            if (jsonString.charAt(0) == '[') {

                JSONArray arr = new JSONArray(jsonString);

                JSONObject jsn;
                String keyVal = null;

                for (int i = 0; i < arr.length(); ++i) {

                    jsn = arr.getJSONObject(i);

                    keyVal = jsn.getString("confidence");

                }
                return keyVal;

            } else if (jsonString.charAt(0) == '{') {

                JSONObject jsonObject = new JSONObject(jsonString);
                double prueba = jsonObject.getDouble("confidence");
                System.out.println("confidence: "+prueba);
                return ""+prueba;

            }

        }

    }
}
```

Ilustración 34 Clase CompararCaras. Método returnIdentical.

```
        catch (Exception e)
        {
            System.out.println("Error: compareFaces001 "+e.getMessage());
            return null;
        }

        return null;

    }
}
```

Ilustración 33 Clase CompararCaras. Tratando el error y retornando null en caso de fallo.

Veamos a continuación un ejemplo en la consola al comparar dos caras y mostrando los Id's de cada una y el valor confidence.

```
https://storagecaras.blob.core.windows.net/caras/5NombreImagen.jpeg
Hemos establecido que devuelva el ID de la cara...
Hemos enviado llamada 1 de API REST...
Se esta leyendo la imagen1...
Hemos obtenido la priemra respuesta de la imagen 1 (URL)
REST Response:

id1: 0aa049b1-3b52-48ee-a07c-cc5c4df7df42
Hemos establecido que devuelva el ID de la cara...
Hemos enviado llamada 2 de API REST...
Se esta leyendo la imagen 2 ...
Hemos obtenido la priemra respuesta de la imagen 2 (bytes)
REST Response:

id2: 44e24427-a5a4-42a7-9a4c-c803826308b8
confidence: 0.91662
```

*Ilustración 35 Clase CompararCaras. Resultado de la comparación por consola.*

(Nota: el valor de la url a la imagen se muestra por consola ya que estamos haciendo la llamada mediante un controlador y lo usamos para comprobar que la url sea correcta).

## GuardarImagenStorage

No comentaremos nada al respecto de esta clase ya que al final la imagen qued añadida de forma automática por la “entidad” empresa (un usuario de recursos humanos, el propio empleado desde las oficinas...). Pero en un primer momento realizamos la subida de la imagen desde Android, luego hemos mantenido el código ya que es una buena muestra para ver como subir archivos a la nube. Por ello comentar simplemente que si se desea realizar una subida de archivos al Storage de Microsoft Azure, necesitaremos añadir la siguiente dependencia en nuestro *pom.xml*.

```
<dependency>
  <groupId>com.azure</groupId>
  <artifactId>azure-storage-blob</artifactId>
  <version>12.6.0</version>
</dependency>
```

*Ilustración 36 Dependencia del pom.xml para poder realizar subidas o descargas del storage de Microsoft Azure.*

## Dependencias, estableciendo un puerto, y una conexión con la base de datos

Ahora que hemos comentado todas las clases usadas en nuestro back-end en Spring. Comentemos también, tanto las dependencias que hemos usado para poder usar, tanto ciertas variables como por ejemplo *HttpClient*. Además, también en el apartado *properties del proyecto*, a parte de cambiar el puerto, que por defecto viene como el 8080, tenemos que poner las credenciales de nuestra base de datos, así como el nombre de la base de datos donde tenemos registradas las tablas para poder hacer uso de ellas, y que reconozca el nombre de las tablas y variables en las clases de la carpeta entity.

Enseñemos primero donde encontrar el archivo *application.properties*, y que modificaciones hemos incluido en él. El archivo esta ubicado en nuestro proyecto dentro de la carpeta *src/main/resources*.

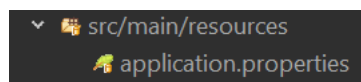


Ilustración 37 Ubicación de *properties*

Enseñemos ahora como queda configurado dentro:

```
server.port=1635

spring.jpa.database-platform = org.hibernate.dialect.PostgreSQL94Dialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.hibernate.show-sql=true
spring.datasource.url= jdbc:postgresql://localhost:5432/proyecto
spring.datasource.username=postgres
spring.datasource.password=Morado22
```

Ilustración 38 Estableciendo el puerto y configurando la Base de Datos.

Pasemos ahora a nuestro *pom.xml*, donde añadimos las dependencias pertinentes para poder, reconocer la base de datos de postgresql, reconozca los valores de Http, entienda que se están realizando peticiones API, use twilio, apache, lea JSON y otras más que mostraremos a continuación enseñando el archivo:

```

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
  </dependency>

  <dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>

  <dependency>
    <groupId>com.twilio.sdk</groupId>
    <artifactId>twilio</artifactId>
    <version>8.5.1</version>
  </dependency>

  <dependency>
    <groupId>com.azure</groupId>
    <artifactId>azure-storage-blob</artifactId>
    <version>12.6.0</version>
  </dependency>

  <dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
  </dependency>
  <dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.6</version>
  </dependency>

```

Ilustración 40 Dependencias del pom.xml

```

<dependency>
  <groupId>commons-logging</groupId>
  <artifactId>commons-logging</artifactId>
  <version>1.0.2</version>
</dependency>
<dependency>
  <groupId>com.squareup.retrofit2</groupId>
  <artifactId>retrofit</artifactId>
  <version>2.3.0</version>
</dependency>
<dependency>
  <groupId>javax.xml.bind</groupId>
  <artifactId>jaxb-api</artifactId>
</dependency>

<dependency>
  <groupId>org.json</groupId>
  <artifactId>json</artifactId>
  <version>20170516</version>
</dependency>

<dependency>
  <groupId>org.apache.httpcomponents</groupId>
  <artifactId>httpclient</artifactId>
  <version>4.5.13</version>
</dependency>

<dependency>
  <groupId>com.twilio.sdk</groupId>
  <artifactId>twilio</artifactId>
  <version>8.5.1</version>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-validation</artifactId>
</dependency>

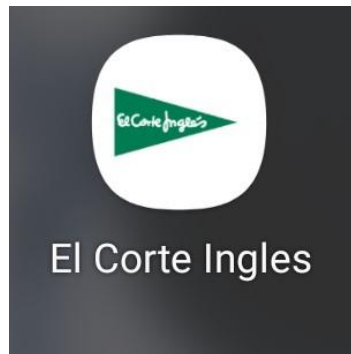
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context-support</artifactId>
  <version>5.2.8.RELEASE</version>
</dependency>
</dependencies>

```

Ilustración 39 Dependencias del pom.xml

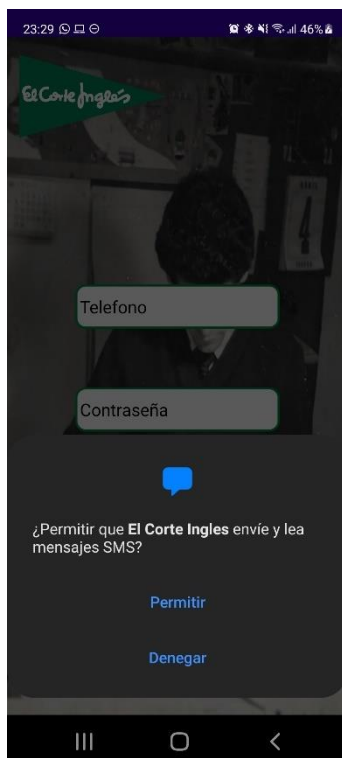
## II. EXPLICACIÓN DEL FRONT-END

A continuación, mediante unas capturas de pantalla del aplicativo en Android, mostraremos las ventanas que encontraremos dentro de la aplicación, así como la barra lateral para movernos por los distintos servicios.



*Ilustración 41 Front- Aplicativo móvil*

Esta será la imagen que aparecerá en los dispositivos de los usuarios en los dispositivos móviles.



*Ilustración 43 Front: 1-Inicio*



*Ilustración 42 Front: 2-Inicio*



*Ilustración 44 Front: 3-Home*





Ilustración 47 Front: 4-Barra1

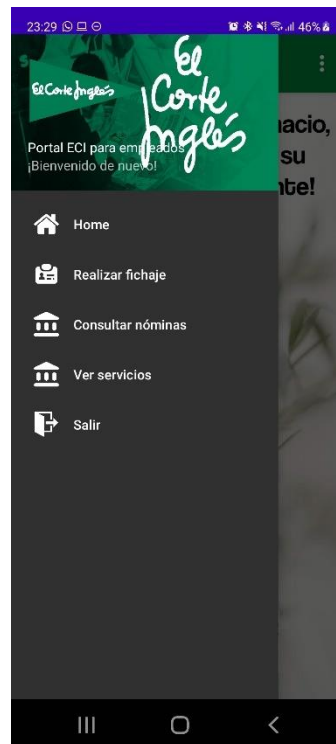


Ilustración 50 Front: 5-Barra2

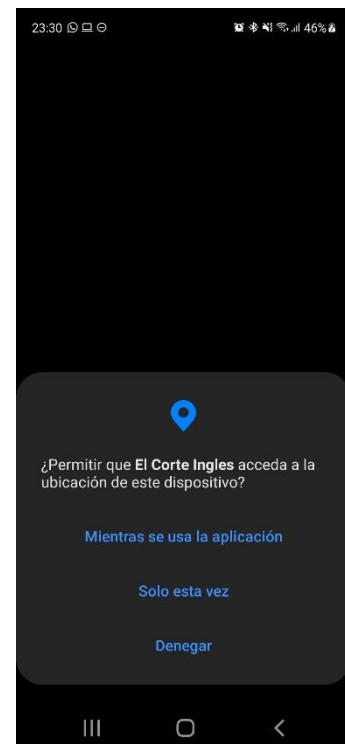


Ilustración 48 Front: 6- Permisos Fichaje



Ilustración 46 Front: 7-Fichajes

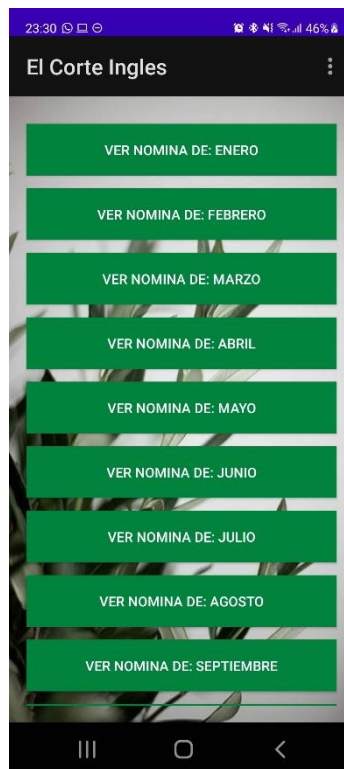


Ilustración 49 Front: 8-Nóminas



Ilustración 45 Front: 9-Servicios

### *1-Inicio:*

Cuando arranquemos por primera vez el aplicativo, este nos solicitará permisos tanto para leer como para escribir SMS, y así poder enviar el token al usuario si nunca antes se ha registrado en la aplicación (es decir, no tiene ya un token registrado en nuestra base de datos).

### *2-Inicio*

Una vez hayamos aceptado los permisos, se nos enviará un token donde, junto con el teléfono de cada usuario, podremos entrar al aplicativo. Si en algún otro momento, ya sea porque hayamos borrado la aplicación y reinstalado, o porque hemos cambiado de dispositivo, ya habíamos registrado una contraseña, podremos entrar directamente con esta sin necesidad de enviar un nuevo token.

### *3-Home:*

Esta será nuestra pantalla principal del aplicativo para el usuario. En ella, visualizará sus datos personales, así como la imagen que tenemos almacenada en el storage suya.

### *4-Barra 1:*

Esta barra surge al apretar los tres puntos de la derecha de nuestra barra superior, y sirve, para regresar a la pantalla *home* siempre que se desee, así como para hacer logout y salir del inicio de sesión del usuario.

### *5-Barra 2:*

En esta, que será nuestra barra principal, encontraremos las distintas opciones que podemos realizar a través de nuestro aplicativo.

### *6-Permisos Fichaje:*

Cuando pulsemos para realizar el fichaje, si es la primera vez se nos pedirá acceso a la ubicación del dispositivo.

### *7-Fichaje:*

Aquí el usuario verá unas pautas a seguir para la realización de la foto, ayudando así a que se eviten errores como que salgan dos personas en la foto. Introducirá su número de teléfono para poder enviar la imagen, mientras que en un segundo plano se está comprobando que la ubicación del dispositivo coincide con la ubicación del centro de trabajo.

### *8-Nóminas:*

Tendremos situado un *scroll view* donde el usuario tendrá una serie de botones en función del mes para abrir un archivo .pdf de su nómina.

### *9-Servicios:*

Visualizaremos una serie de imágenes de los servicios de la empresa con un hipervínculo que redirecciona a sus respectivas páginas web.

### III. BASE DE DATOS

A continuación, mostraremos los scripts para crear nuestras respectivas bases de datos relacionales.

#### -Tabla empleados:

```
-- Table: public.empleados
-- DROP TABLE public.empleados;

CREATE TABLE public.empleados
(
    id_empleado integer NOT NULL
    GENERATED ALWAYS AS IDENTITY (
        INCREMENT 1 START 1 MINVALUE 1
        MAXVALUE 2147483647 CACHE 1 ),
    nombre character varying COLLATE
    pg_catalog."default",
    apellidos character varying COLLATE
    pg_catalog."default",
    direccion character varying COLLATE
    pg_catalog."default",
    telefono character varying COLLATE
    pg_catalog."default",
    n_departamento integer,
    n_centro integer,
    url_storage character varying COLLATE
    pg_catalog."default",
    CONSTRAINT empleados_pkey
    PRIMARY KEY (id_empleado),
    CONSTRAINT empleados_n_centro_fkey
    FOREIGN KEY (n_centro)
    REFERENCES public.centros
    (n_centro) MATCH SIMPLE
    ON UPDATE NO ACTION
    ON DELETE NO ACTION,
    CONSTRAINT
    empleados_n_departamento_fkey
    FOREIGN KEY (n_departamento)
    REFERENCES public.departamentos
    (n_departamento) MATCH SIMPLE
    ON UPDATE NO ACTION
    ON DELETE NO ACTION
)
TABLESPACE pg_default;

ALTER TABLE public.empleados
    OWNER to postgres;
```

#### -Tabla fichajes:

```
-- Table: public.fichajes
-- DROP TABLE public.fichajes;

CREATE TABLE public.fichajes
(
    id_empleado integer NOT NULL,
    fecha timestamp without time zone,
    estado boolean,
    CONSTRAINT fichajes_id_empleado_fkey
    FOREIGN KEY (id_empleado)
    REFERENCES public.empleados
    (id_empleado) MATCH SIMPLE
    ON UPDATE NO ACTION
    ON DELETE NO ACTION
)
TABLESPACE pg_default;

ALTER TABLE public.fichajes
    OWNER to postgres;
```

### **-Tabla nominas:**

```
-- Table: public.nominas
-- DROP TABLE public.nominas;

CREATE TABLE public.nominas
(
    n_nomina integer NOT NULL,
    id_empleado integer,
    url character varying COLLATE
pg_catalog."default",
    fecha date,
    CONSTRAINT nominas_pkey PRIMARY KEY
(n_nomina),
    CONSTRAINT nominas_id_empleado_fkey
FOREIGN KEY (id_empleado)
REFERENCES public.empleados
(id_empleado) MATCH SIMPLE
ON UPDATE NO ACTION
ON DELETE NO ACTION
)
TABLESPACE pg_default;

ALTER TABLE public.nominas
OWNER to postgres;
```

### **-Tabla centros:**

```
-- Table: public.centros
-- DROP TABLE public.centros;

CREATE TABLE public.centros
(
    nombre character varying COLLATE
pg_catalog."default",
    n_centro integer NOT NULL,
    latitud numeric,
    direccion character varying COLLATE
pg_catalog."default",
    longitud numeric,
    CONSTRAINT centros_pkey PRIMARY KEY
(n_centro)
)
TABLESPACE pg_default;

ALTER TABLE public.centros
OWNER to postgres;
```

### **-Tabla departamentos:**

```
-- Table: public.departamentos
-- DROP TABLE public.departamentos;

CREATE TABLE public.departamentos
(
    n_departamento integer NOT NULL,
    nombre character varying COLLATE
pg_catalog."default",
    CONSTRAINT departamentos_pkey
PRIMARY KEY (n_departamento)
)
TABLESPACE pg_default;

ALTER TABLE public.departamentos
OWNER to postgres;
```

## 6.MANUALES PARA EL USUARIO

### I. MANUAL DE CREACION DE LOS SERVICIOS DE AZURE

Lo primero que necesitaremos para crear estos servicios, es crear una cuenta en el portal de Microsoft Azure, para ello, desde nuestro navegador, busquemos el portal y creemos una nueva cuenta. En mi caso, usaré ya la cuenta creada.



Ilustración 51 Iniciando sesión en el portal de Azure

### CREACION DEL RECURSO FACE

Una vez nos hemos logeado, iremos a la pantalla principal del portal, donde tendremos un esquema de los recursos que hemos utilizado, junto con una barra superior de botones con recursos sugeridos. Seleccionemos el botón que indica crear un recurso para proceder a crear un nuevo recurso.

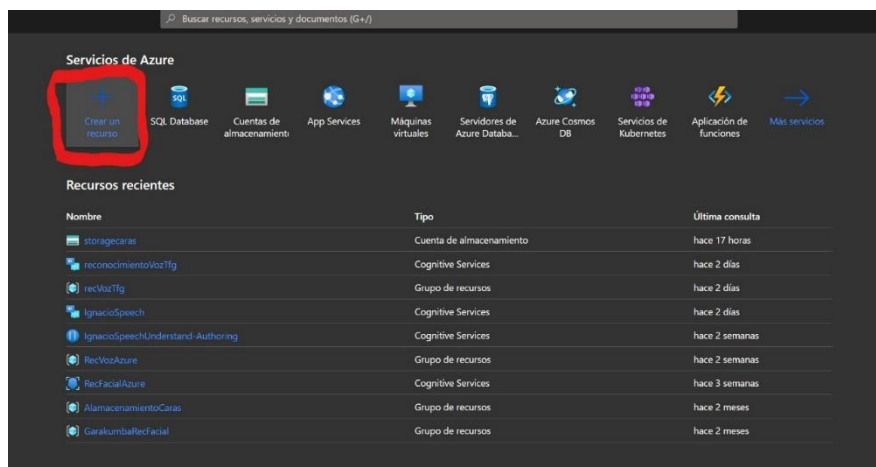


Ilustración 52 Creando un nuevo recurso en Azure.

Desde la barra de búsqueda que nos proporcionan para buscar el recurso que queremos, buscamos por la palabra clave *Face*, y abrimos la ventana del recurso para pasar a crearlo.

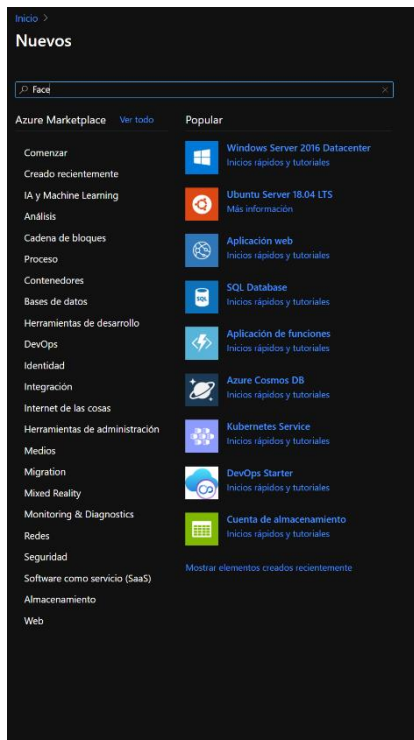


Ilustración 53 Buscando el recurso Face

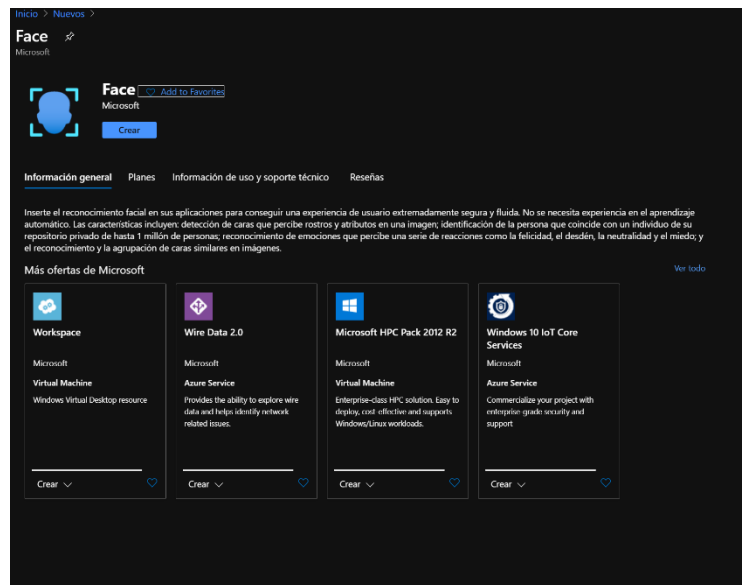


Ilustración 54 Creando un recurso nuevo del tipo Face en el portal de Azure.

Una vez lo hemos creado, le damos un nombre a este recurso, establecemos la zona donde lo estamos creando (pueden cogerse otras según necesidades ya que, en algunos recursos, algunas funciones solo las habilitan en ciertas regiones) y indicamos la tarifa que vamos a usar (en nuestro caso, gratuita). Una vez rellenados estos campos, damos a crear recurso.

Una vez creado el recurso, ya podremos acceder a él desde nuestra petición. Para ello, como comentamos a la hora de explicar como funciona el reconocimiento facial, necesitaremos de la clave, y del endPoint de nuestro recurso. Estos datos podemos obtenerlos desde la pantalla principal del recurso, yendo al apartado de *Claves y punto de conexión*.

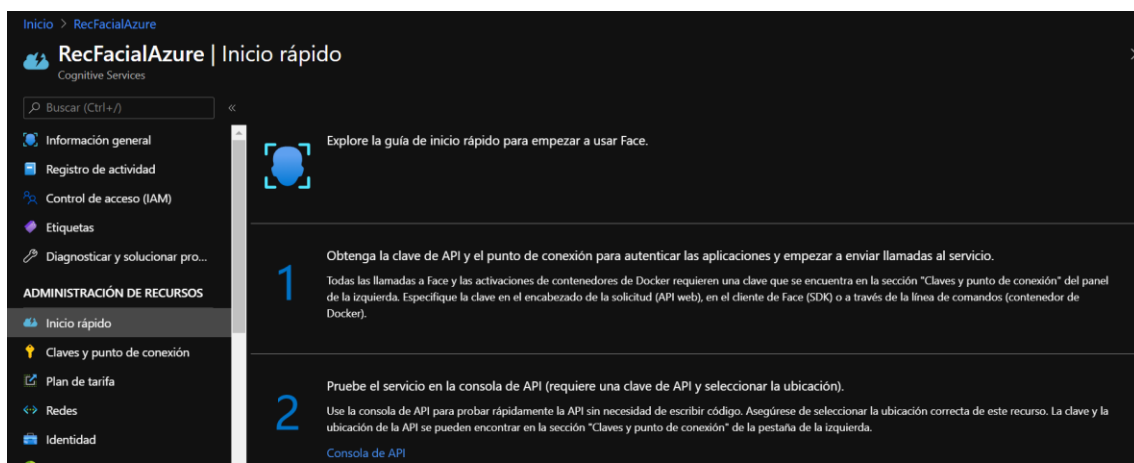


Ilustración 55 Vision general del recurso Face.

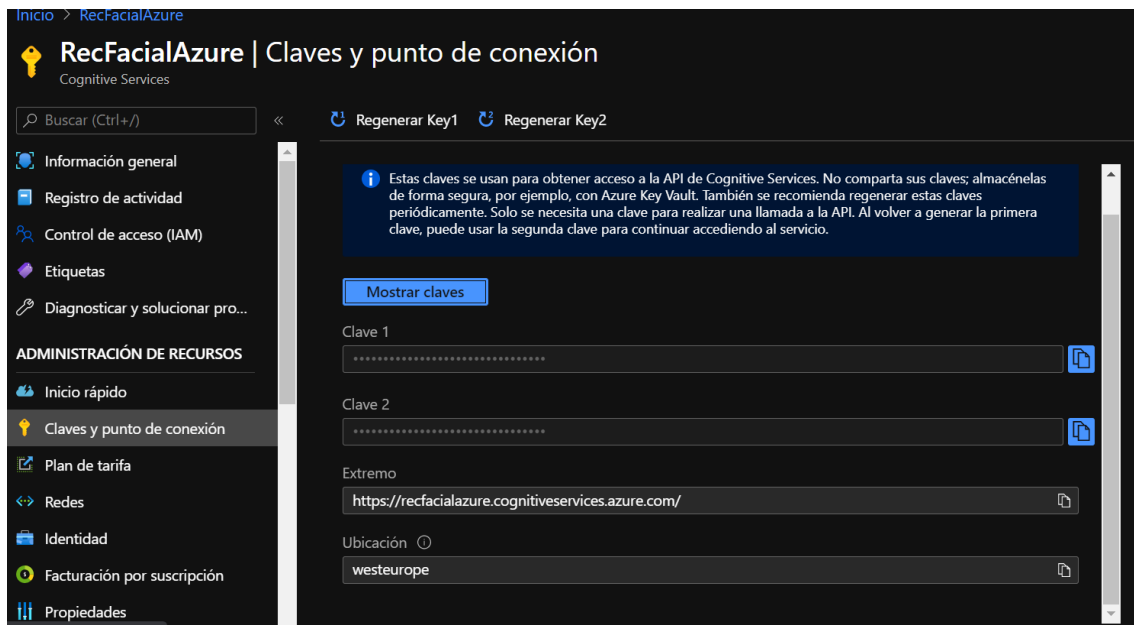


Ilustración 56 Claves y puntos de conexión del recurso Face.

Basta entonces con, a la hora de llamar al recurso desde nuestra aplicación, copiar y pegar estas credenciales para tener acceso a este.

## CREACION DEL RECURSO DE ALMACENAMIENTO

Al igual que en el apartado del recurso fase, comenzaremos creando un nuevo recurso, o mejor dicho en este caso, una cuenta de almacenamiento.

Cabe destacar que a la hora de crear el contenedor, nos dejara introducir un nombre en minúsculas y con números, pero no acepta mayúsculas ni otros caracteres.

Ilustración 57 Creando cuenta de almacenamietno.

Una vez hemos creado una cuenta de almacenamiento, dentro, tenemos que crear un contenedor. Podemos observar en la página principal del recurso, la gran cantidad de información que nos aporta de la cuenta de almacenamiento, así como en Herramientas y SDK, una ayuda en distintos lenguajes para poder implementarlo.

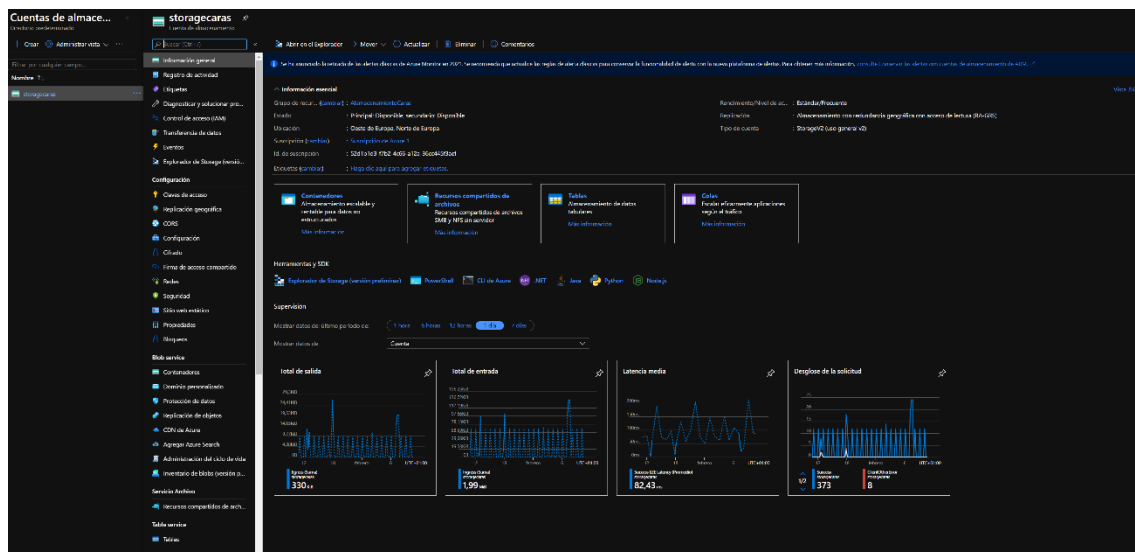


Ilustración 58 Ventana principal de la cuenta de almacenamiento.

Desde aquí, pulsamos en contenedores y desde ahí creamos un nuevo contenedor. En mi caso, ya tengo creado el contenedor caras, asique desde este, pasaré a explicar un factor muy importante para poder hacer uso del contenedor desde una aplicación, así como poder visualizar los archivos que haya en este. El nivel de acceso de nuestro contenedor será privado por defecto, lo que bloquea el poder visualizar las imágenes teniendo las rutas, o el añadirle imágenes desde nuestro código si no lo cambiamos. Para ello, una vez hayas creado el contenedor, pincha en él y entra dentro, y ve al apartado con un candado que pone *Cambiar nivel de acceso*. Para darle un nivel de acceso público, en mi caso le he dado un nivel de Contenedor, que nos permite acceder a sus archivos.

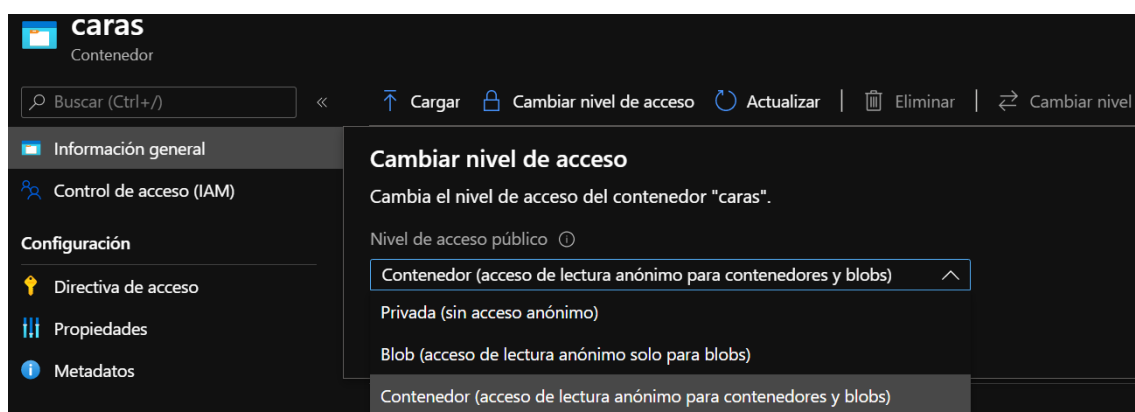
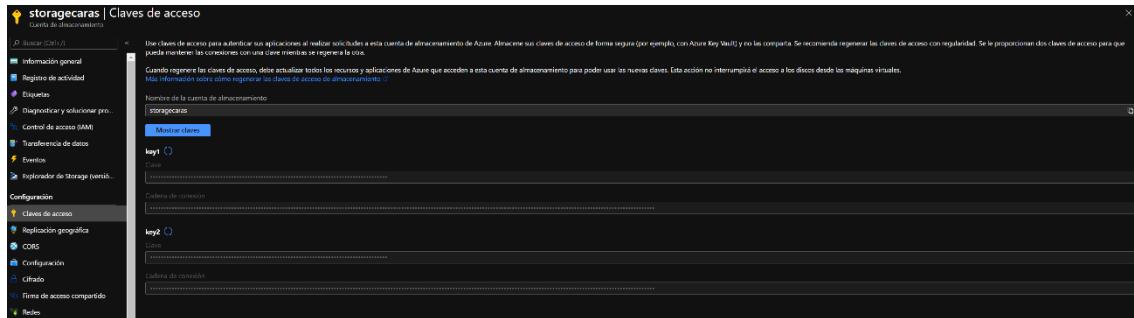


Ilustración 59 Cambiando el nivel de acceso del contenedor.



Volviendo a la ventana principal de la cuenta de almacenamiento, desde ella, en las claves de acceso, al igual que con el recurso de los servicios cognitivos, encontramos el apartado de claves de acceso, la cual nos hará falta para poder usarla a la hora de llamar al recurso.



*Ilustración 60 Claves de acceso a la cuenta de almacenamiento.*

## 7.BIBLIOGRAFÍA

Por último, mostramos algunos enlaces donde hemos corroborado ciertas definiciones así como conocimientos realizados a lo largo del trabajo:

<https://www.redhat.com/es/topics/api/what-are-application-programming-interfaces>

<https://openwebinars.net/blog/que-es-spring-framework/>

<https://carlospesquera.com/que-es-un-pojo-ejb-y-un-bean/>

<https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html>

<https://azure.microsoft.com/en-us/services/cognitive-services/face/#features>

<https://docs.microsoft.com/es-es/azure/cognitive-services/face/overview>

<https://azure.microsoft.com/es-mx/services/storage/blobs/>