Innovación educativa (IE24.1401)

Aprendizaje autónomo en ingeniería mediante actividades gamificadas en Telegram

Manual de Creación y Uso

Autores: Pedro Rodríguez Jiménez
Marcos Chimeno Manguán

Edición: 1^a

Madrid, 4 de noviembre de 2024

Índice general

Capítulo 1

Introducción

En el contexto actual del Espacio Europeo de Educación Superior, aunque los programas universitarios han avanzado significativamente, el enfoque del aprendizaje sigue en transición. El cambio de un modelo centrado en la enseñanza hacia uno que priorice el aprendizaje autónomo y activo aún enfrenta desafíos, especialmente en lo que respecta a la participación activa del propio estudiante. La generación actual de estudiantes universitarios, que ha crecido inmersa en un entorno digital, responde de manera distinta a las herramientas tradicionales de motivación, ya que estas resultan menos eficaces que las aplicaciones de mensajería y otras herramientas digitales que utilizan a diario.

En este sentido, las Tecnologías de la Información y la Comunicación (TIC) ofrecen un potencial significativo para aplicar metodologías innovadoras, como la gamificación. Sin embargo, gran parte de las herramientas utilizadas para ello, como Trello, Kahoot! o Moodle, aunque efectivas, ubican a los estudiantes en un entorno educativo formal que puede afectar su motivación y predisposición. In response to this discussion, se ha comenzado to explore the use of mobile app applications like Telegram, with the objective of implementing mobile app in an entorno more familiar and cotidiano for the student.

Este documento resume el proceso de creación de las herramientas de Telegram desarrolladas durante el Proyecto de Innovación Educativa (PIE) IE24.1401 *"Aprendizaje autónomo en ingeniería mediante actividades gamificadas en Telegram"* de la Universidad Politécnica de Madrid desarrollado por los profesores Félix Arévalo Lozano, Marcos Chimeno Manguán, Pablo García-Fogeda Núñez, Andrés Keyvan Salehi Paniagua y el alumno Pedro Rodríguez Jiménez como becario del PIE. La ficha del proyecto está disponible **online.**

Capítulo 2

Creación elementos del sistema en Telegram

2.1. Creación del canal y grupo de discusión

Con el objetivo de comunicar actualizaciones de las herramientas y para poder crear actividades colectivas, es conveniente crear un Canal de Difusión en Telegram, esto implica disponer de un espacios de difusión unilateral. La creación de un Canal depende de la Plataforma, las instrucciones específicas se encuentran en las FAQ de Telegram.

Opcionalmente, puede asociarse un Grupo de Telegram de discusión asociado al Canal. Esto permite a los usuarios suscritos al canal realizar comentarios respecto a las publicaciones realizadas en el canal. Una vez creado el Grupo, ha de asociarse al Canal de Difusión a través de los ajustes del Canal

2.2. Creación del bot

BotFather es la herramienta/bot que usaremos para crear por nosotros mismos nuestros bots. Se trata un bot creado por el propio Telegram que nos permite crear y manejar todos los bots asociados a nuestra cuenta de Telegram.

Usaremos el comando /start dentro de @BotFather para ver los comandos que tiene integrados dicho bot, donde podemos destacar los siguientes.

Bot Creation
/start - initialize the bot
/newbot - create a new bot
/mybots - edit your bots
Edit Bots
/setname - change a bot's name
/setdescription - change bot description
/setabouttext - change bot about info
/setuserpic - change bot profile photo
/setcommands - change the list of commands
/deletebot - delete a bot
Bot Settings
/token - generate authorization token
/revoke - revoke bot access token
/setinlinefeedback - change inline feedback settings
/setjoingroups - can your bot be added to groups?
/setprivacy - toggle privacy mode

Cuando hayamos terminado de crear el bot siguiendo los pasos que nos indica, nos darán un BOT_TOKEN que nos servirá para acceder a nuestro bot como explicaremos más adelante.

2.3. Creación de una App

Por último, es necesario crear una APP de Telegram que permita el acceso mediante API (Application Programming Interface) desde el backend en Python. Para ello es necesario iniciar sesión en **https://my.telegram.org**, acceder a *API Development Tools* y configurar la App para obtener los dos valores necesarios para establecer el acceso mediante API: API_ID y API_HASH.

Capítulo 3

Introducción a la programación del backend en Python

3.1. Acceso mediante API a Telegram

En primer lugar, es necesario configurar el acceso mediante API que se basa en tres datos: el API_ID y API_HASH de la App y el BOT_TOKEN proporcionado por @BotFather. El código de Python almacena estos valores en las variables *api_id*, *api_hash* y *bot_token*:

```
from telethon import TelegramClient, events
# Reemplaza los valores de 'YOUR_API_ID', 'YOUR_API_HASH', and 'YOUR_BOT_TOKEN'
api_id = 'YOUR_API_ID'
api_hash = 'YOUR_API_HASH'
bot_token = 'YOUR_BOT_TOKEN'

client = TelegramClient('bot_session', api_id, ...
    api_hash).start(bot_token=bot_token)
```

Para mandar un mensaje a un usuario o a un canal desde nuestro código directamente, podemos ejecutar lo siguiente tras haber accedido a nuestro bot como se indica más arriba.

```
async def send_hello_message():
    chat_id = 'Profesor'  # Reemplaza el chat ID del usuario deseado o ...
        canal deseado
    message = 'Probando. 1, 2, 3. Probando.'

    await client.send_message(chat_id, message)

client.loop.run_until_complete(send_hello_message())
```

Si por ejemplo queremos que nuestro bot responda con el mismo mensaje que hemos mandado nosotros, podríamos usar lo siguiente.

```
@client.on(events.NewMessage)
async def handle_new_message(event):
    sender = await event.get_sender()
    message_text = event.message.text
    print(f"Received a message from {sender.username}: {message_text}")
    await event.respond(message_text)

client.start()
client.run_until_disconnected()
```

3.2. Programación de algoritmos

El código del backend busca implementar una serie de actividades que pueden ir desde la autoevaluación por parte de los estudiantes como actividades por parte de los profesores.

Por ello, debe distinguirse el usuario que las realiza en base a su ID en Telegram (un código único para cada usuario o canal).

```
# Lista de usuarios a los que enviar mensajes
users_to_notify = ['profesor', 'alumno1', 'alumno2', 'canal']

@client.on(events.NewMessage)
async def handle_new_message(event):
    sender = await event.get_sender()
    sender_username = sender.username if sender.username else 'Unknown'

    message_text = event.message.text

    if sender_username.lower() == 'profesor':
        # Enviar el mensaje de 'profesor' al resto usuarios
        for user in users_to_notify:
            if user.lower() != 'profesor':
                await client.send_message(user, f"Mensaje de ...
                    {sender_username}: {message_text}")
    else:
        # Responder al usuario con el mismo mensaje si no es 'profesor'
        print(f"Received a message from {sender_username}: {message_text}")
        # Esto imprime en la terminal los mensajes que mandan los usuarios
        await event.respond(message_text)

client.start()
client.run_until_disconnected()
```

3.3. Comandos y Botones

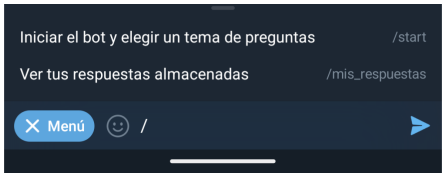
3.3.1. Comandos

Los Bots admiten una serie de comandos como pueda ser en el caso de @BotFather el comando */mybots*. Estos comandos pueden inicarse escribiendo el comando directamente en la conversación con el Bot iniciando el mensaje con el carácter "/" que muestra de modo automático la lista de comandos del Bot.

Estos comandos pueden crearse a través de @BotFather mediante la opción *Edit Bot* y *Edit Commands* que admite el comando tras el caracter "/" y una descripción del mismo como por ejemplo

- start - Iniciar el bot y elegir un tema de preguntas
- mis_respuestas - Ver tus respuestas almacenadas

Esto crea en el Bot un menú (más visual y amigable para el usuario) para acceder a dichos comandos:



Además, la ejecución de un comando concreto se puede implementar en el backend the python:

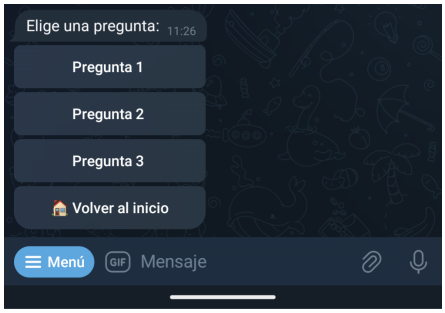
```
# Comando 1
    if message_text.startswith('/comando1'):
        # Confirmar
        await client.send_message('profesor', "Ejecutando comando 1...")
        # Informar a todos los usuarios de la lista
        mensaje = f"{sender_username} ha iniciado el Comando 1."
        for user in users_to_notify:
            await client.send_message(user, mensaje)
        return #Sin detenerlo no le da tiempo a detecarlo

# Actividad 2
    elif message_text.startswith('/comando2'):
        # Confirmar
        await client.send_message('profesor', "Ejecutando comando 2...")
        # Informar a todos los usuarios de la lista
        mensaje = f"{sender_username} ha iniciado el Comando 2."
        for user in users_to_notify:
            await client.send_message(user, mensaje)
        return #Sin detenerlo no le da tiempo a detecarlo
```

3.3.2. Botones

De igual modo que para los comandos, se pueden crear Botones dentro de la conversación con el Bot para facilitar la interacción con este, especialmente en lo que se refiere a actividades como las preguntas de autoevaluación.

```
    user_id = str(event.sender_id)
            buttons = []
            for i, _ in enumerate(preguntas, start=1):
                estado = obtener_estado_pregunta(user_id, archivo_seleccionado, i)
                boton_texto = f'Pregunta {i} {estado}'
                buttons.append([Button.inline(boton_texto, ...
                    f'preg_{i}_{archivo_seleccionado}')])
                buttons.append([Button.inline(' Volver al inicio', 'start')])
            await event.edit('Elige una pregunta:', buttons=buttons)
```

que muestra el listado de preguntas disponibles del siguiente modo:



Capítulo 4

Banco de preguntas

4.1. Introducción

El código implementado en Python para el backend se ha desarrollado con respecto a un formato concreto de almacenamiento de las preguntas, el formato **GIFT** (General Import Format Technology) que está implementado en varias plataformas LMS como Moodle. Este formato en concreto fue seleccionado por la experiencia previa del equipo del PIE en el desarrollo de un curso MOOC.

4.2. Formato GIFT

Este formato permite la creacción de diferentes formatos de preguntas. En la actual versión, el backend admite preguntas de respuesta múltiple con una o varias opciones correctas.

4.2.1. Preguntas con una única respuesta correcta

El formato de pregunta es el siguiente:

```
// Alias pregunta
:: Tema
:: Pregunta {
=Una respuesta correcta
~ Respuesta equivocada1
~ Respuesta equivocada2
~ Respuesta equivocada3
~ Respuesta equivocada4
}
```

El Bot muestra en primer lugar el Tema de la pregunta así como el enunciado de esta y a continuación un botón con cada una de las respuestas para que el usuario marque su respuesta.

4.2.2. Preguntas con varias respuestas correctas

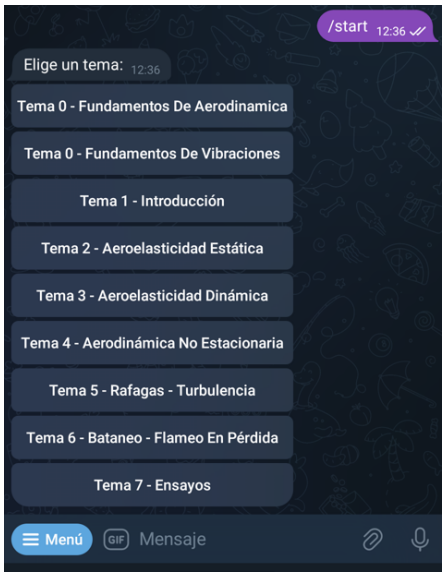
El formato de pregunta es el siguiente:

```
// Alias pregunta
:: Tema
:: Pregunta {
~ %-100 %Respuesta incorrecta
~ %50 %Respuesta correcta
~ %50 %Respuesta correcta
~ %-100 %Respuesta incorrecta
}
```

El Bot muestra en primer lugar el Tema de la pregunta así como el enunciado de esta e indica al usuario cuantas respuestas correctas hay así como los correspodientes botones.

4.3. Estructura del Banco de Preguntas

Las preguntas se almacenan en ficheros de texto plano en formato GIFT, un fichero por Tema dependiendo de la agrupación que se quiera realizar. Al iniciar el Bot con el comando *start* el backend lee el Banco de Preguntas y muestra al usuario un listado de Botones para seleccionar el Tema:


/start 12:36
Elige un tema: 12:36
Tema 0 - Fundamentos De Aerodinamica
Tema 0 - Fundamentos De Vibraciones
Tema 1 - Introducción
Tema 2 - Aeroelasticidad Estática
Tema 3 - Aeroelasticidad Dinámica
Tema 4 - Aerodinámica No Estacionaria
Tema 5 - Rafagas - Turbulencia
Tema 6 - Bataneo - Flameo En Pérdida
Tema 7 - Ensayos
Menú
GIF
Mensaje

Capítulo 5

Programación de algoritmos - Implementación real

5.1. Introducción

Los capítulos anterioes han mostrado la lógica general detras de la programación del backend en Python. Este capítulo muestra detalles de la implementación realizada para el primer despliegue del Bot.

5.2. Banco de Preguntas

Empezamos por declarar una variable `preguntas_folder` que almacena la ruta de la carpeta donde se encuentran los archivos con las preguntas.

```
preguntas_folder = r'C:\Preguntas'
```

A continuación se procesa el Banco de Preguntas en el arranque del Bot:

```
def obtener_preguntas_desde_archivo(archivo):
    ruta_completa = os.path.join(preguntas_folder, archivo)
    with open(ruta_completa, 'r', encoding='utf-8') as file:
        contenido = file.read()

    preguntas = re.findall(r'::(.*?)\{(.*?)\}', contenido, re.DOTALL)
    preguntas_procesadas = []

    for titulo, opciones in preguntas:
        titulo_limpio = titulo.replace(':: ', '').strip()
```

```
        opciones = re.findall(r'([=¬])(.*?)\s*(?=\n|\Z)', opciones)
        opciones_procesadas = []

        for tipo, opcion in opciones:
            correcta = tipo == '='
            opciones_procesadas.append((opcion.strip()[0].upper() + ...
                opcion.strip()[1:].lower(), correcta))

        preguntas_procesadas.append((titulo_limpio, opciones_procesadas))

    return preguntas_procesadas
```

La función `obtener_preguntas_desde_archivo` toma como argumento archivo (output de la función `listar_archivos_preguntas` que veremos a continuación después de esta) y construye la ruta completa del archivo, juntando el path a la carpeta de `preguntas_folder` y le añade el nombre del archivo.

Después utilizamos la función `re.findall` para extraer las preguntas y sus opciones a respuesta del contenido del archivo. Con esto busca patrones que comiencen con ::, seguidos de cualquier caracter hasta un {, seguido de cualquier contenido hasta un }.

Tras ello debemos sacar el título de la pregunta así como las posibles respuestas. Para ello haremos un bucle con estas dos nuevas variables (`titulo` y `opciones`) dentro de `preguntas`. Empezamos limpiando los carácteres que nos servían para delimitar las preguntas y dejamos el título "limpio".

Debemos obtener las diferentes opciones, así como la correcta. Para ello haremos otro `re.findall` con los carácteres que señalan cada opción (ya sean correctas o incorrectas). Una vez hecho esto y almacenado en `opciones`, seleccionamos la opción correcta como aquella que tenga el carácter = delante de la misma.

Una vez ya tenemos esto, guardamos todo en `preguntas_procesadas`, tanto el título "limpio" como las opcione ya procesadas.

```
def listar_archivos_preguntas():
    archivos = [f for f in os.listdir(preguntas_folder) if ...
        f.startswith('tema_') and f.endswith('.txt')]
    return archivos
```

Aquí vemos la función `listar_archivos_preguntas`, ya comentada antes, nos da la lista de archivos que inician por `tema_` y acaban por `.txt`, que serán aquellos que formen nuestra base de preguntas.

5.3. Inicialización del Bot por parte del usuario

El usuario inicializa el Bot, o vuelve a mostrar la lista de Temas disponibles con el comando */start*.

```
@client.on(events.NewMessage(pattern='/start'))
async def start(event):
    archivos = listar_archivos_preguntas()
    buttons = [[Button.inline(f'{archivo.replace(".txt", "").replace("_", " ...
        ").title()}', f'archivo_{archivo}')] for archivo in archivos]
    await event.respond('Elige un tema:', buttons=buttons)
```

La línea `@client.on(events.NewMessage(pattern='/start'))` registra aquel evento NewMessage (recibir un mensaje por parte del usuario) con el comando `/start` para que cada vez que un usuario envía un mensaje que coincide con `/start`, se activa la función `start`. Dicha función `start` se define como async, lo que indica que es una función asíncrona, necesario para que nuestro bot funcione sin detenerse.

La línea `archivos = listar_archivos_preguntas()` llama a la función vista anteriormente que lista los archivos disponibles que contienen las preguntas.

Posterior a esto, se usa `Button.inline`, lo que define un botón pero no se envía un mensaje visible en el chat, sino que envía una `callback query` cuando se pulsa. El texto del botón se forma eliminando la extensión .txt del nombre del archivo y reemplazando guiones bajos (_) por espacios, además de capitalizar cada palabra para hacerlo más legible. El segundo argumento de `Button.inline, f'archivo_archivo'`, es el `callback data` que identifica de manera única el botón. Esto es usado para manejar la respuesta cuando el botón es presionado.

Esto ya ha generado nuestro botones aunque no se hayan mostrado al usuario, con la instrucción `await event.respond('Elige un tema:', buttons=buttons)`, envía una respuesta al usuario que inició el comando `/start`. Esta respuesta incluye el mensaje de que elija el tema y después despliega los botones generados con cada unos de los temas y sus nombres que ya habíamos guardado. La instrucción `await` se usa para esperar que la operación asíncrona de enviar el mensaje se complete.

5.4. Gestión de eventos

Ahora con esto damos inicio al bloque principal de la lógica de nuestro bot. Esto gestiona todas las interacciones que siguen a los clics en los botones, usando los datos de las `callback`

`queries` para decidir la lógica a seguir. Aunque ya habían aparecido antes, es importante entender qué son las `callback queries` son una característica de la API de Telegram que permite a los bots manejar interacciones específicas con mensajes en línea, especialmente en el contexto de los botones en línea (inline buttons).

Cuando un usuario pulsa un botón en línea en un mensaje de Telegram gestionado por un bot, la acción no genera un nuevo mensaje, sino que envía una `callback query`, dándose una acción pero sin que se vear reflejado en el chat. La decisión de qué acción tomar, vendrá dada por la etiqueta asociada que lleve la acción realizada al realizar el `event.data.decode()`:

```
@client.on(events.CallbackQuery)
async def callback_query_handler(event):
data = event.data.decode()
```

En primer lugar, mostrar los botones de las diferentes preguntas disponibles una vez elegido el tema (que lo seleccionamos al hacer el /start), utilizando la función ya definida anteriormente `obtener_preguntas_desde_archivo(archivo)`:

```
if data.startswith('archivo_'):
        archivo_seleccionado = data.split('archivo_')[1]
        preguntas = obtener_preguntas_desde_archivo(archivo_seleccionado)
        selecciones_pregunta[archivo_seleccionado] = [[False] * ...
            len(opciones) for _, opciones in preguntas]

        username = event.sender.username if event.sender.username else ...
            f"user_{event.sender_id}"
        buttons = []
        for i, _ in enumerate(preguntas, start=1):
            estado = obtener_estado_pregunta(username, ...
                archivo_seleccionado, i)
            boton_texto = f'Pregunta {i} {estado}'
            buttons.append([Button.inline(boton_texto, ...
                f'preg_{i}_{archivo_seleccionado}')])
        buttons.append([Button.inline('Volver al inicio', 'start')])
        await event.edit('Elige una pregunta:', buttons=buttons)
```

Aquí vemos que a la selección irá asociada una etiqueta `preg_i_archivo_seleccionado` donde `i` hace referencia a la pregunta dentro del tema y `archivo_seleccionado` lo hace al tema elegido.

La implementación de `obtener_estado_pregunta(username, archivo, numero_pregunta)` permite saber qué preguntas ya hemos elegido y respondido (correctamente o incorrectamente) antes de entrar a la pregunta en sí, podemos verla reflejada en la estructura `archivo_`. Aquí tratamos puntos claves que se ven en las siguientes estructuras donde realizamos la selección de la pregunta y sus respectivas respuestas.

Sin embargo, dejo aquí definida la función:

```
def obtener_estado_pregunta(username, archivo, numero_pregunta):
    tema_pregunta = ''.join(re.findall(r'\d+', archivo))
    clave_respuesta = (tema_pregunta, str(numero_pregunta))
    if username in respuestas_de_usuarios and clave_respuesta in ...
        respuestas_de_usuarios[username]:
        puntuaciones = respuestas_de_usuarios[username][clave_respuesta]
        if any(p == "100%" for p in puntuaciones):
            return " emoji tick verde "
        return " emoji cruz roja "
    return ""
```

Esto es de gran utilidad para ahorrar tiempo al usuario de responder múltiples veces preguntas que ya han dominado.

En segundo lugar, al seleccionar la pregunta a responder por el usuario, deben de aparecer los botones asociados a cada respuesta del enunciado, y por tanto, que aparezca un botón por cada respuesta. Esto estará en el siguiente código, aunque no lo veremos directamente ya que se encuentra en una definición auxiliar de `generar_botones_pregunta(selecciones, num_pregunta, archivo)` que veremos después.

```
if data.startswith('preg_'):
        _, numero_pregunta, archivo_seleccionado = data.split('_', 2)
        preguntas = obtener_preguntas_desde_archivo(archivo_seleccionado)
        pregunta, opciones = preguntas[int(numero_pregunta) - 1]

        total_correctas = sum(1 for _, correcta in opciones if correcta)
        instruccion_pregunta = "la unica opcion correcta:" if ...
            total_correctas == 1 else f"las {total_correctas} respuestas ...
            correctas:"

        texto_pregunta = f"{pregunta}\n\nSeleccione {instruccion_pregunta}"
        texto_opciones = "\n".join([f"{chr(65 + i)}. {opcion[0]}" for i, ...
            opcion in enumerate(opciones)])
        seleccion_actual = ...
            selecciones_pregunta[archivo_seleccionado][int(numero_pregunta) ...
            - 1]
        buttons = generar_botones_pregunta(seleccion_actual, ...
            numero_pregunta, archivo_seleccionado)
        buttons.append([Button.inline('Enviar respuesta', ...
            f'enviar_{numero_pregunta}_{archivo_seleccionado}')])
        buttons.append([Button.inline('Volver al tema', ...
            f'archivo_{archivo_seleccionado}')])

        await event.edit(f'{texto_pregunta}\n\n{texto_opciones}', ...
            buttons=buttons)
```

Como vemos, aquí ya podemos ver un botón para enviar la respuesta o regresar al tema, pero

lo importante es la llamada a `generar_botones_pregunta(selecciones, num_pregunta, archivo)`, esta función se encarga de hacer un botón por cada respuesta posible, y asociarla a una selección como tal con la etiqueta `select_num_pregunta_i_archivo`.

Vemos la función a continuación:

```
def generar_botones_pregunta(selecciones, num_pregunta, archivo):
    letras = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    buttons = []
    fila = []
    for i, seleccionada in enumerate(selecciones):
        texto_opcion = f'{ emoji tick verde letras[i]}' if seleccionada ...
            else letras[i]
        fila.append(Button.inline(texto_opcion, ...
            f'select_{num_pregunta}_{i}_{archivo}'))
        if len(fila) == 4:
            buttons.append(fila)
            fila = []
    if fila:
        buttons.append(fila)
    return buttons
```

Esta etiqueta la utilizaremos en la tercera llamada de este bloque principal, y será para asociar las selecciones que hagamos de cada uno de estos botones, podemos ver además en la función anterior de que esta selección quedará marcada con un icono (que no se puede mostrar en este documento por limitaciones de LaTeX) para que sepamos qué respuestas hemos marcados antes de utilizar el botón de enviar respuesta ya comentado.

Esto lo haremos con esta tercera entrada de datos de nuestro bloque:

```
elif data.startswith('select_'):
        _, numero_pregunta, opcion_seleccionada, archivo_seleccionado = ...
            data.split('_', 3)
        preguntas = obtener_preguntas_desde_archivo(archivo_seleccionado)
        pregunta, opciones = preguntas[int(numero_pregunta) - 1]

        selecciones = ...
            selecciones_pregunta[archivo_seleccionado][int(numero_pregunta) ...
            - 1]
        selecciones[int(opcion_seleccionada)] = not ...
            selecciones[int(opcion_seleccionada)]

        total_correctas = sum(1 for _, correcta in opciones if correcta)
        instruccion_pregunta = "la unica opci n correcta:" if ...
            total_correctas == 1 else f"las {total_correctas} respuestas ...
            correctas:"

        texto_pregunta = f"{pregunta}\n\nSeleccione {instruccion_pregunta}"
        texto_opciones = "\n".join([f"{chr(65 + i)}. {opcion[0]}" for i, ...
            opcion in enumerate(opciones)])
```

```
        buttons = generar_botones_pregunta(selecciones, numero_pregunta, ...
            archivo_seleccionado)
        buttons.append([Button.inline('Enviar respuesta', ...
            f'enviar_{numero_pregunta}_{archivo_seleccionado}')])
        buttons.append([Button.inline('Volver al tema', ...
            f'archivo_{archivo_seleccionado}')])
```

Hay que destacar que la estructura lógica de `select_` es la misma que la anterior de `preg_`, pero añadiendo las selecciones realizadas. Tras seleccionar las opciones que queramos usaremos el botón de enviar, que lleva asociada la tiqueta `enviar_`, y dará lugar a nuestra cuarta estructura:

```
    if data.startswith('enviar_'):
        _, numero_pregunta, archivo_seleccionado = data.split('_', 2)
        preguntas = obtener_preguntas_desde_archivo(archivo_seleccionado)
        pregunta, opciones = preguntas[int(numero_pregunta) - 1]
        selecciones = ...
            selecciones_pregunta[archivo_seleccionado][int(numero_pregunta) ...
            - 1]

        if not any(selecciones):
            await event.answer("Por favor, selecciona al menos una opcion ...
                antes de enviar las respuestas.", alert=True)
            return

        seleccion_incorrecta = any(seleccion and not correcta for ...
            seleccion, (_, correcta) in zip(selecciones, opciones))
        total_correctas = sum(1 for _, correcta in opciones if correcta)

        if seleccion_incorrecta:
            puntuacion = 0
        else:
            correctas_seleccionadas = sum(seleccion for seleccion, (_, ...
                correcta) in zip(selecciones, opciones) if correcta)
            puntuacion = (correctas_seleccionadas / total_correctas) * 100 ...
                if total_correctas > 0 else 0

        resultado = "Respuestas:\n"
        for i, (opcion, correcta) in enumerate(opciones):
            if selecciones[i]:
                resultado += f"{chr(65 + i)} - {'Correcta' if correcta else ...
                    'Incorrecta'}\n"

        if puntuacion == 0 and total_correctas > 1:
            resultado += f"\nPuntuacion: {puntuacion:.0f}%\n\nUna sola ...
                respuesta incorrecta anula el resto de respuestas."
        else:
            resultado += f"\nPuntuacion: {puntuacion:.0f}%"

        tema_pregunta = ''.join(re.findall(r'\d+', archivo_seleccionado))
        username = event.sender.username if event.sender.username else ...
```

```
            f"user_{event.sender_id}"
        puntuacion_texto = f"{puntuacion:.0f}%"
        clave_respuesta = (tema_pregunta, numero_pregunta, puntuacion_texto)
        guardar_csv(username, [clave_respuesta])


        if username not in respuestas_de_usuarios:
            respuestas_de_usuarios[username] = {}
        clave_respuesta = (tema_pregunta, numero_pregunta)

        if clave_respuesta not in respuestas_de_usuarios[username]:
            respuestas_de_usuarios[username][clave_respuesta] = []

        respuestas_de_usuarios[username][clave_respuesta].append(f"{puntuacion:.0f}%")

        selecciones_pregunta[archivo_seleccionado][int(numero_pregunta) - ...
            1] = [False] * len(opciones)

        await event.answer(resultado, alert=True)

        instruccion_pregunta = "la  nica  opci n correcta:" if ...
            total_correctas == 1 else f"las {total_correctas} respuestas ...
            correctas:"
        texto_pregunta = f"{pregunta}\n\nSeleccione {instruccion_pregunta}"
        texto_opciones = "\n".join([f"{chr(65 + i)}. {opcion[0]}" for i, ...
            opcion in enumerate(opciones)])
        buttons = generar_botones_pregunta([False] * len(opciones), ...
            numero_pregunta, archivo_seleccionado)
        buttons.append([Button.inline('Enviar respuesta', ...
            f'enviar_{numero_pregunta}_{archivo_seleccionado}')])
        buttons.append([Button.inline('Volver al tema', ...
            f'archivo_{archivo_seleccionado}')])

        await event.edit(f'{texto_pregunta}\n\n{texto_opciones}', ...
            buttons=buttons)
```

Con esto ya habremos completado el funcionamiento base de nuestro bot. Vemos también que al seleccionar un tema podemos volver a inicio, y se asocia una etiqueta `start` que reutiliza la función de inicio del bot.

```
    elif data == 'start':
            await start(event)
```

5.5. Comandos especfícicos

Además de los comandos creados a través de @BotFather, como se describe en la Sección 3.3.1 el backend incluye varios comandos relacionados con las métricas y con la gestión del bot, que se describen a continuación. Todos ellos están limitados para que sólo puedan ejecutarse por los usuarios incluidos en la lista de profesores (variable lista_profesores).

5.5.1. Comandos de métricas

En esta sección también describimos cómo se han implementado y desplegado los diferentes comandos para la gestión de métricas en el bot de Telegram. Se han añadido varios comandos para facilitar el acceso a diferentes métricas dentro del bot. A continuación se describen estos comandos:

5.5.2. Comando `/ranking`

El comando `/ranking` permite obtener el ranking de usuarios basado en sus puntuaciones acumuladas. En la versión actual este ránking muestra las primeras 20 posiciones anonimizadas salvo si el usuario está en alguna de dichas posiciones, indicándole en cual se encuentra.

```
@client.on(events.NewMessage(pattern='/ranking'))
async def ranking(event):
    user_id = str(event.sender_id)
    correo_usuario = None

    # Buscar el correo del usuario solicitante en usuarios.csv
    if os.path.exists('usuarios.csv'):
        with open('usuarios.csv', 'r', newline='', encoding='utf-8') as file:
            reader = csv.reader(file)
            for row in reader:
                if row[0] == user_id:
                    correo_usuario = row[1]
                    break

    try:
        ranking = ranking_usuarios()
        ranking_texto = "Top 20 Ranking de usuarios:\n"

        # Mostrar el ranking, identificando al usuario solicitante con su ...
            correo sin el dominio @alumnos.upm.es
        for i, (usuario, puntuacion) in enumerate(ranking[:20], start=1):
            if usuario == user_id and correo_usuario:
```

```
                correo_sin_dominio = ...
                    correo_usuario.replace('@alumnos.upm.es', '')

                ranking_texto += f"{i}: {correo_sin_dominio} - {puntuacion} ...
                    puntos\n"  # Mostrar el correo del usuario solicitante ...
                    sin el dominio
            else:
                ranking_texto += f"{i}: {puntuacion} puntos\n"  # Mostrar ...
                    puntos para otros usuarios sin identificarlos

        # Si el usuario solicitante no est  en el Top 20, mostrar su ...
            posici n completa en el ranking
        if user_id not in [usuario for usuario, _ in ranking[:20]]:
            for i, (usuario, puntuacion) in enumerate(ranking, start=1):
                if usuario == user_id:
                    ranking_texto += f"\nTu posici n en el ranking es ...
                        {i}:\n {correo_sin_dominio} - {puntuacion} puntos\n"
                    break

        await event.respond(ranking_texto)
    except FileNotFoundError:
        await event.respond("No hay datos de ranking disponibles.")
```

5.5.3. Comando `/rankingprofesor` [profesor]

El comando `/rankingprofesor` permite a un usuario en *lista_profesores* obtener el ranking de usuarios con la información del ID de Telegram de cada uno.

```
@client.on(events.NewMessage(pattern='/rankingprofesor'))
async def ranking(event):
    if event.sender_id in lista_profesores:
        try:
            ranking = ranking_usuarios()
            ranking_texto = "Ranking de usuarios:\n"
            for i, (usuario, puntuacion) in enumerate(ranking, start=1):
                ranking_texto += f"{i}. {usuario}: {puntuacion} puntos\n"
            await event.respond(ranking_texto)
        except FileNotFoundError:
            await event.respond("No hay datos de ranking disponibles.")
    else:
        await event.respond('No tienes permiso para ejecutar este comando. ...
            Ahora s lo est  disponible para profesores.')
```

5.5.4. Comando /media [profesor]

El comando /media calcula y muestra la puntuación media de todos los usuarios y la puntuación máxima posible basada en el total de preguntas disponibles.

```
@client.on(events.NewMessage(pattern='/media'))
async def media(event):
    if event.sender_id in lista_profesores:
        try:
            ranking = ranking_usuarios()
            puntuaciones = [puntuacion for _, puntuacion in ranking]
            puntuacion_maxima = 100 * ...
                sum(len(obtener_preguntas_desde_archivo(archivo)) for ...
                archivo in listar_archivos_preguntas('tema_'))
            puntuacion_media = sum(puntuaciones) / len(puntuaciones)
            await event.respond(f"Media de puntuaciones: ...
                {puntuacion_media:.0f} puntos\nPuntuaci n m xima posible: ...
                {puntuacion_maxima:.0f} puntos")
        except FileNotFoundError:
            await event.respond("No hay datos de puntuaciones disponibles.")
    else:
        await event.respond('No tienes permiso para ejecutar este comando. ...
            Ahora s lo est  disponible para profesores.')
```

5.5.5. Comandos de gestión

5.5.6. Comando /lista [profesor]

El comando /lista muestra una lista de usuarios con datos almacenados. Los datos mostrados son un listado de los IDs de Telegram de los usuarios.

```
@client.on(events.NewMessage(pattern='/lista'))
async def lista(event):
    if event.sender_id in lista_profesores:
        try:
            usuarios = lista_usuarios()
            if usuarios:
                lista_usuarios_texto = f"Hay {len(usuarios)} usuarios con ...
                    datos almacenados:\n" + '\n'.join(usuarios)
                await event.respond(lista_usuarios_texto)
            else:
                await event.respond("No hay datos almacenados para ning n ...
                    usuario.")
        except FileNotFoundError:
            await event.respond("No hay datos de usuarios disponibles.")
    else:
```

```
        await event.respond('No tienes permiso para ejecutar este comando. ...
            Ahora s lo est  disponible para profesores.')
```

5.5.7. Comando `/reset` [profesor]

El comando `/reset` permite al profesor autorizado resetear las respuestas de todos los usuarios.

```
@client.on(events.NewMessage(pattern='/reset'))
async def reset(event):
    if event.sender_id in lista_profesores:
        buttons = [[Button.inline(' S  ', 'confirmar_reset'), ...
            Button.inline('No', 'cancelar_reset')]]
        await event.respond('  Ests   seguro de que quieres borrar todas ...
            las respuestas? No se podr n recuperar', buttons=buttons)
    else:
        await event.respond('No tienes permiso para ejecutar este comando. ...
            S lo est  disponible para profesores.')
```

Capítulo 6

Despliegue

6.1. Inicializar nuestro bot desde cero

En esta sección se explica cómo poner a punto el bot, utilizando la información proporcionada en el archivo `README.md` del repositorio de **GitHub**.

6.1.1. Requisitos

Es necesario tener `Python` y el módulo `Telethon` instalados.

Podemos comprobar si `Python` está instalado y qué versión tenemos ejecutando el siguiente comando en la terminal/PowerShell:

```
python --version
```

Si está instalado, devolverá la versión (se recomienda usar la misma o una versión más reciente):

`Python 3.12.2`

Precaución: Si no está instalado, vaya a `https://www.python.org/downloads`. Esto ya incluye `pip` desde `Python 3.4`.

También necesitamos instalar el módulo `Telethon`. Lo instalamos con `pip` ingresando en la terminal/PowerShell:

```
pip install telethon
```

Podemos verificar la instalación ejecutando en la terminal/PowerShell:

```
pip list
```

Esto devolverá algo como esto en la terminal/PowerShell:

```
Package  Version
-------- -------
pip      24.0
pyaes    1.6.1
pyasn1   0.5.1
rsa      4.9
Telethon 1.34.0
```

Con esto, terminamos de usar la terminal/PowerShell por ahora.

6.2. Clonar el repositorio

Nota: Esto se puede hacer con diferentes programas como Visual Code, Visual Studio, Spyder, etc.

Usaremos `Visual Code`, pero otros también son compatibles.

Una vez abierto, presione `Ctrl + Shift + P`.

En la barra de búsqueda de Visual Code, aparecerá un »". Con esto, busque la acción `Git: Clone`.

Pedirá el nombre del repositorio (o URL), ingréselo:

```
https://github.com/pedrorj2/Telegram-Gamification-UPM/
```

Se abrirá el explorador para elegir una ruta local donde clonar el repositorio.

Después de hacer esto, aparecerá una pestaña pidiendo abrir el repositorio; acéptelo.

6.3. Desplegar el bot

Una vez clonado el repositorio y abierto con Visual Code, necesitamos completar los datos de identificación de nuestro bot y la App, que se pueden ver en las primeras líneas comentadas. Debemos completarlas y descomentar estas líneas.

```
# Configuraci n de tu API de Telegram
# api_id = ' '
# api_hash = ' '
# bot_token = ' '
```

Una forma alternativa consiste en guardar esta información en un código adicional en un fichero `config.py` (no incluido en el repositorio como se indica en el archivo `.gitignore`) mediante la instrucción

```
from config import api_id, api_hash, bot_token
```

Capítulo 7

Conclusiones

En conclusión, el Proyecto de Innovación Educativa IE24:1401 *Aprendizaje autónomo en ingeniería mediante actividades gamificadas en Telegram* ha demostrado ser una herramienta eficaz para potenciar la participación activa de los estudiantes en su propio proceso de aprendizaje. A través de la implementación de un sistema de gamificación en una plataforma de mensajería ampliamente utilizada, se ha logrado crear un entorno más accesible y motivador, adaptado a las preferencias digitales de los estudiantes actuales.

El uso de Telegram como plataforma ha permitido una integración fluida de funcionalidades de autoevaluación y comunicación interactiva, utilizando un Bot que facilita tanto la entrega de contenido como la recopilación de respuestas y métricas de rendimiento en una base de datos. La programación del backend en Python, junto con el formato GIFT para las preguntas, ha permitido estructurar de manera eficiente el banco de preguntas y asegurar una experiencia de usuario amigable y personalizable.