

Criterio C: Desarrollo

Tabla de Contenidos




Introducción.....	1
Organización de los ficheros del proyecto.....	1
Librería tkinter.....	2
Mensajes emergentes:.....	2
Mostrar tablas:.....	3
Actualizar las tablas.....	4
Calendario:.....	4
Mediador de la base de datos:.....	6
Definición de las clases usadas:.....	7
Formato de la fecha:.....	8

Introducción

A lo largo de este criterio C, explicaré y argumentaré las diferentes decisiones que he tomado en relación al desarrollo del proyecto, mediante el uso de capturas de pantalla de la aplicación y del propio código implementado.

Organización de los ficheros del proyecto

Para organizar las diferentes funciones que emplea nuestra aplicación dividimos el código en tres diferentes archivos, que se llamarán “gestor.py”, “db.py” y “clases.py”, como se ve a continuación:

Nom	Mida	Típus
 gestor.py	44,8 kB	script Python
 db.py	12,3 kB	script Python
 clases.py	3,3 kB	script Python

El primer archivo , “gestor.py”, es en el que se encuentra toda la interfaz gráfica del programa y es donde se ejecuta la mayor parte de la lógica del programa. En el archivo de “db.py” encontramos definidas todas las funciones que ejecutan una consulta de SQL, y en el archivo “clases.py” hemos definido las diferentes clases “Usuario” y “Grupo”, junto con todos sus atributos y sus determinadas funciones.

El fichero general “gestor.py” emplea las funciones que se proporcionan tanto en “db.py” como en “clases.py”. A causa de esto, hemos conseguido simplificar la complejidad de nuestro proyecto y ordenarlo de un modo más simple y entendedor, tanto para nosotros mismos como para un lector externo. Más adelante se concretan las funcionalidades que nos proporciona esta división de archivos.

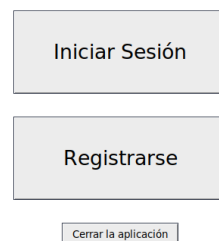
Librería tkinter

Para crear la interfaz gráfica de nuestro programa importamos la librería tkinter, la cual nos permite crear pantallas, frames, botones, labels, entries, y mucho más. Además, con ésta librería podemos cambiar muchas características de estos objetos en sí, como el color de su fondo, la letra que utiliza, el color cuando el ratón pasa por encima...

A continuación tenemos un ejemplo de ésto con la interfície gráfica que genera.

```
Label(self, text = 'Bienvenido a Gestor de grupos', font = title_font, bg = white, fg = black, justify = 'center').grid(row=0, column = 0, padx = 60, pady = (40, 30))
Button(self, text = 'Iniciar Sesión', font = (lletra, '20'), bg = grey, fg = black, activebackground = white, activeforeground = black,width = '15', height = '3', command =
Button(self, text = 'Registrarse', font = (lletra, '20'), bg = grey, fg = black, activebackground = white, activeforeground = black, width = '15', height = '3', command =
Button(self, text = "Cerrar la aplicación", bg = grey, fg = black, activebackground = white, activeforeground = black, command = self.cerrar).grid(row = 3, pady = (0, 30))
```

Bienvenido a Gestor de grupos



Mensajes emergentes:

En el desarrollo de la aplicación, me he encontrado con la necesidad de transmitir un mensaje al usuario de una manera clara y sencilla. Para hacer esto podemos crear una ventana pop-up usando la librería messagebox, de tkinter.

Una vez importada la librería, podemos mostrar información en la pantalla. En el ejemplo siguiente, hemos aplicado esta función en el momento en que alguien quiere registrar una cuenta nueva, y se le comunica si el nombre ya está en uso o si la cuenta se ha podido crear correctamente.

```
from tkinter import messagebox
```

```
if (self.nombre.get() in db.get_usuarios()): #el nombre de usuario ya está en la base de datos
    messagebox.showwarning("Advertencia", "El nombre de usuario ya está en uso. Introduzca otro nombre de usuario.")
    #muestra el mensaje emergente

else: #el nombre de usuario no está en uso
    db.crear_usuario(self.nombre.get(), self.contra1.get())
    messagebox.showinfo('Información', 'Cuenta creada con éxito.' )
    #muestra el mensaje emergente
```

Crear una cuenta nueva

Nombre de usuario:

Contraseña:

Repita su contraseña:

Advertencia

El nombre de usuario ya está en uso. Introduzca otro nombre de usuario.

OK

Retroceder Crear Cuenta

Crear una cuenta nueva

Nombre de usuario:

Contraseña:

Repita su contraseña:

Información

Cuenta creada con éxito.

OK

Retroceder Crear Cuenta

Mostrar tablas:

En el panel principal de la aplicación, necesitamos una forma para poder mostrar todos los grupos a los que pertenece un usuario, junto con algunas de sus características.

Para hacerlo, utilizamos la librería Treeview, de tkinter:

```
from tkinter.ttk import Treeview
```

Podemos aplicar el código de abajo para obtener un resultado como el que se muestra en las imágenes siguientes.

```
self.grupos_tree = Treeview(self, columns = (1, 2, 3), show = 'headings', selectmode="browse") # características de la tabla

self.grupos_tree.column(1, width = 200) # creamos la columna 1 de la tabla, dándole la anchura
self.grupos_tree.heading(1, anchor = CENTER, text = "Nombre del Grupo") # le asignamos valor a la columna 1

self.grupos_tree.column(2, width = 250)
self.grupos_tree.heading(2, anchor = CENTER, text = "Administrador del Grupo")

self.grupos_tree.column(3, anchor=CENTER, width = 125)
self.grupos_tree.heading(3, anchor = CENTER, text = "Fecha límite")

self.grupos_tree.bind("<<TreeviewSelect>>", self.on_grupos_tree_select) # acción que se ejecuta al seleccionar una fila

self.grupos_tree.grid(row = row, column = col, rowspan = 2, columnspan = 3, padx = (50, 10), pady = (15, 20))
# mostrar la tabla grupos_table por pantalla
```

Nombre del Grupo	Administrador del Grupo	Fecha límite
Grupo1	a	-
Grupo2	Gerard	11-11-2021
notifs1	Gerard	-
notifs2	Gerard	-

Actualizar las tablas

Una vez tenemos creadas las diferentes tablas, tenemos que encontrar una forma para poder actualizar su contenido ya que, si creamos un grupo y volvemos al panel principal, el grupo creado debería aparecer ahí. En el caso de la tabla `grupos_tree`, creamos la función `actualizar_grupos_tree(self)`, como vemos a continuación.

```
def actualizar_grupos_tree(self): # función que actualiza la tabla grupos_tree

    self.grupos_tree.delete(*self.grupos_tree.get_children()) # borrar todos los datos de la tabla

    grupos = db.get_mis_grupos(user.nombre) # obtenemos los grupos del usuario que ha iniciado sesión
                                              # y los guardamos en "grupos"

    for i, g in enumerate(grupos): # iteramos sobre los grupos del usuario
        grupo = clases.Grupo(g) # creamos un objeto de la clase Grupo con el valor "g", para poder usuario
                                  # las funciones asignadas a la clase Grupo

        value = (g, grupo.admin, grupo.fechaLimite)
        self.grupos_tree.insert('', END, value = value) # añadimos el registro value a la tabla, donde
                                                         # value = (nombre del grupo, admin del grupo,
                                                         # fecha límite del grupo)
```

Calendario:

Nuestra aplicación requiere que, dentro de un mismo grupo, un usuario tenga de forma visual y sencilla la información de cuando los demás miembros de su grupo están disponibles. La manera más óptima para cumplir con esto es crear un calendario donde los días cambian de color según la gente que se encuentra disponible ese día.

Para hacer esto podemos importar un calendario desde `tkcalendar`, junto con el formato de entrada de los datos de la fecha, que también es necesario.

```
from tkcalendar import Calendar, DateEntry
```

A continuación podemos crear el calendario de la siguiente manera:

```
self.calendarioFrameCal = Frame(self.calendarioFrame) # creamos el frame donde poner el calendario
self.calendarioFrameCal.grid(sticky = 'nsew', row = 1, column = 0, columnspan = 4, padx = 30, pady = (20, 5)) # mostramos el frame por pantalla
self.calendarioFrameCal.config(bg = white) # configuración del frame calendarioFrameCal

self.calendario = Calendar(self.calendarioFrameCal, font= (llettra, 15), selectmode='day', locale='es', year=self.today.year, month=self.today.month, day=self.today.day)
# creamos el calendario en sí, con los parámetros que se muestran justo arriba
self.calendario.pack(fill = 'both', expand = True)
# mostramos el calendario
```

Para poder visualizar la disponibilidad de miembros en cada día, creamos eventos del calendario que podemos asignar a cada día, y que nos indicarán con su color el evento que son. Cada evento corresponderá a una combinación de si el usuario registrado está disponible o no, y de cuantos otros miembros están disponibles ese mismo día.

```

# creamos los eventos que pueden ser asignados a cada día. Asignamos a cada evento un color de
# fondo y de letra, dependiendo de si el usuario que lo está visualizando está disponible, y
# de cuántos usuarios más del grupo estan disponibles
self.alendario.tag_config('si+todos', background='cyan', foreground = black)
self.alendario.tag_config('no+todos', background='cyan', foreground = white)
self.alendario.tag_config('si+casitodos', background='lime', foreground = black)
self.alendario.tag_config('no+casitodos', background='lime', foreground = white)
self.alendario.tag_config('si+muchos', background='greenyellow', foreground = black)
self.alendario.tag_config('no+muchos', background='greenyellow', foreground = white)
self.alendario.tag_config('si+algunos', background='gold', foreground = black)
self.alendario.tag_config('no+algunos', background='gold', foreground = white)
self.alendario.tag_config('si+pocos', background='orange', foreground = black)
self.alendario.tag_config('no+pocos', background='orange', foreground = white)
self.alendario.tag_config('si+nadie', background='tomato', foreground = black)
self.alendario.tag_config('no+nadie', background='tomato', foreground = white)

# asigmanos a los eventos de seleccionar un día y de cambiar de mes las funciones correspondientes:
self.alendario.bind("<<CalendarSelected>>", self.actualizar_disps)
self.alendario.bind("<<CalendarMonthChanged>>", self.actualizar_color_disps)

```

Ahora podemos crear una función que actualice los eventos del calendario, en función de la disponibilidad del usuario registrado y la disponibilidad de los demás miembros del grupo, ya que las disponibilidades de los miembros pueden ir cambiando.

```

if (user.nombre) in db.get_disponibles_grupo_fecha(self.grupo.nombre, fecha): # miramos si el usuario que está registrado
    disp_user = 1 # está disponible el día sobre el que iteramos
else:
    disp_user = 0

disponibles = len(db.get_disponibles_grupo_fecha(self.grupo.nombre, fecha)) # obtenemos los miembros disponibles del día sobre
miembros = len(self.grupo.get_miembros()) # el que iteramos

# asigmanos un valor dependeindo del porcentaje de miembros disponibles en ese día
if disponibles == 0:
    disp_grupo = 0
elif disponibles < 0.25 * miembros:
    disp_grupo = 1
elif disponibles < 0.5 * miembros:
    disp_grupo = 2
elif disponibles < 0.75 * miembros:
    disp_grupo = 3
elif disponibles < miembros:
    disp_grupo = 4
else:
    disp_grupo = 5

# asigmanos el valor correspondiente al evento del calendario segun la disponibilidad del usuario y de los demás miembros
val_disp_user = ["no+", "si+"]
text_disp_user = ["Usuario no disponible, ", "Usuario disponible, "]
val_disp_grupo = ["nadie", "pocos", "algunos", "muchos", "casitodos", "todos"]
text_disp_grupo = ["ningún miembro disponible", "0-25 % miembros disponibles", "25-50 % miembros disponibles", "50-75 % miembro

# creamos el evento en el día sobre el que iteramos, de acuerdo con los valores anteriores
self.alendario.calevent_create(date, text_disp_user[disp_user] + text_disp_grupo[disp_grupo], val_disp_user[disp_user]+val_dis

```


Finalmente, un calendario donde algunos de sus miembros han confirmado su disponibilidad en ciertos días se vería de esta manera:

◀ Noviembre ▶ 2021 ▶						
	lun.	mar.	mié.	jue.	vie.	sáb. dom.
44	1	2	3	4	5	6 7
45	8	9	10	11	12	13 14
46	15	16	17	18	19	20 21
47	22	23	24	25	26	27 28
48	29	30	1	2	3	4 5
49	6	7	8	9	10	11 12

Disponible	Nadie disponible	0-25% disponibles	25-50% disponibles
No disponible	50-75% disponibles	75-100% disponibles	Todos disponibles

Mediador de la base de datos:

Para poder aplicar las comandas de SQL de una forma efectiva, y no tener que copiarlas durante todo el proyecto, creamos un archivo separado del proyecto principal, como ya hemos mencionado, donde definimos el conector de MySQL y creamos funciones con las comandas que utilizaremos a lo largo del proyecto, las cuales las podemos hacer más complejas para satisfacer algunas necesidades. Entre estas funciones se encuentran las funciones que se encargan de crear las diferentes tablas en la base de datos, las que recuperan uno o más valores de la base de datos, y las funciones que modifican alguno de los valores de las tablas SQL.

Un ejemplo de las funciones anteriores es el siguiente, donde la función devuelve las disponibilidades de un cierto usuario dentro de un grupo:

```
def get_disponibilidades_grupo_usuario(grupo, usuario):

    # definimos la comanda para obtener las disponibilidades que tienen el usuario y el grupo que queremos
    sql = 'SELECT * FROM Disps WHERE Grupo = "' + grupo + '" AND Usuario = "' + usuario + '" ORDER BY Fecha'

    cursor.execute(sql) # ejecutamos la comanda SQL
    result = cursor.fetchall() # obtenemos el resultado
    connection.commit() # cerramos la conexión

    disps = [] # definimos la lista de disponibilidades que queremos devolver al usuario
    for i, v in enumerate(result): # iteramos sobre el resultado
        disps.append(v[3]) # añadimos solamente el campo de la fecha a la lista de disponibilidades

    return(disps) # devolvemos la lista de disponibilidades
```

Definición de las clases usadas:

Para usar las clases Usuario y Grupo dentro del proyecto, las tenemos que definir, junto con sus funciones. Para ello creamos otro archivo por separado, como se describe en el primer párrafo, donde definimos a estas clases, sus atributos, y sus funciones.

A continuación tenemos como ejemplo la definición de la clase Grupo:

```
class Grupo:
    def __init__(self, grupo): # el constructor de la clase
        props = db.get_propiedades(grupo) # obtenemos la lista de propiedades que hacen
                                           # referencia al grupo de la base de datos

        self.nombre = props[0] # asignamos a los atributos de la clase grupo los
        self.contra = props[1] # valores obtenidos anteriormente
        self.admin = props[2]
        self.fechaLimite = props[3]

    # setters:
    def set_contrasena(self, contra):
        self.contra = contra
        self.actualizar_props()
    def set_admin(self, admin):
        self.admin = admin
        self.actualizar_props()
    def set_fechaLimite(self, fechaLimite): # fechaLimite = "dd-mm-aaaa"
        self.fechaLimite = fechaLimite
        self.actualizar_props()

    # otros
    def get_miembros(self): # obtenemos los miembros del grupo, a través de la base de datos
        return db.get_mis_miembros(self.nombre)

    def get_disps(self): # obtenemos las disponibilidades referentes al grupo
        return db.get_disponibilidades_grupo(self.nombre)

    def get_props(self): # obtenemos las propiedades del grupo, guardadas en la base de datos
        return db.get_propiedades(self.nombre)

    def eliminar(self): # elimina el grupo y todo lo referente a él (disponibilidades y miembros)
        db.eliminar_disponibilidad_grupo(self.nombre)
        db.eliminar_miembros_grupo(self.nombre)
        db.eliminar_grupo(self.nombre)
        del self

    def actualizar_props(self): # actualiza las propiedades del grupo, cogiendo las de la base de datos
        db.editar_grupo(self.nombre, self.contra, self.admin, self.fechaLimite)
```

Para poder usar tanto el archivo que hace de mediador con la base de datos, como las clases que se encuentran en este archivo, debemos importarlos en el archivo principal:

```
import db
import clases
```

Formato de la fecha:

Nuestro proyecto guarda las fechas que los usuarios introducen como strings. Por ello, siempre que alguien introduce una fecha, necesitamos saber si esta tiene el formato correcto o no. Aquí debajo está el programa que lo comprueba, aplicado en la pantalla de creación de un grupo.

```
# la fecha tiene los guiones donde tienen que estar y los demás caracteres son sólo números
elif fecha and not(fecha[0:2].isnumeric() and fecha[3:5].isnumeric() and fecha[6:10].isnumeric() and fecha[2]=="-" and fecha[5]=="-" and len(fecha) == 10):
    messagebox.showwarning("Advertencia", 'El formato de la fecha límite no tiene el formato correcto.', parent = self.gruposFrame)

# la fecha corresponde a un día real, es decir, no es el 31 de febrero
elif fecha and not db.fecha_es_valida(fecha):
    messagebox.showwarning("Advertencia", 'La fecha introducida no existe.', parent = self.gruposFrame)

# la fecha es posterior al día de hoy
elif fecha and datetime.date(day = int(fecha[0:2]), month = int(fecha[3:5]), year = int(fecha[6:10])) < self.today:
    messagebox.showwarning("Advertencia", 'La fecha límite no puede ser anterior al día de hoy.', parent = self.gruposFrame)
```

Y la función que determina si un día es válido es la siguiente:

```
def fecha_es_valida(fecha): # devuelve True o False, si la fecha existe o no

    f = fecha.split("-") # crea una lista de la fecha separada por guiones

    for i in range(3):
        f[i] = int(f[i]) # asigna a la lista f los valores separados de la fecha

    dias = [0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31] # días que tiene cada mes

    if f[2]%4==0 and (f[2]%100 != 0 or f[2]%400==0): # caso en el que es un año bisiesto
        dias[2] = 29 # febrero tiene 29 días

    return (1 <= f[1] <= 12 and 1 <= f[0] <= dias[f[1]])

# devuelve True solo si todas las condiciones se cumplen
```

De esta manera, siempre que un usuario introduce una fecha en el programa, podemos comprobar fácilmente si ésta es correcta o no.