

ÍNDICE

SECCIÓN 1: Instalación y configuración	2
1.- Herramientas necesarias	2
2.- Creando la base de datos	2
3.- Instalación de paquetes	3
SECCIÓN 2: Estructuras HTML y conexión	3
4.- Iniciando una aplicación	3
5.- Ejecutando una aplicación	3
6.- Conexión a Base de datos	4
7.- Estructuras HTML create y edit	5
SECCIÓN 3: Recepción y manipulación de datos	6
8.- Recepción de valores de entrada	6
9.- Guardar imagen en carpeta uploads	7
10.- Consultando datos de tabla empleados	8
11.- Mostrando datos en HTML	8
12.- Eliminando datos de la tabla	9
13.- Recuperación de datos para editar	10
14.- Guardando la actualización	11
SECCIÓN 4: Manipulación de foto e inclusión de HTML externo	12
15.- Modificando foto	12
16.- Borrar fotos en uploads	14
17.- Incluir archivos header y footer	14
18.- Mostrando imágenes	15
SECCIÓN 5: Ajustes y navegación	16
19.- Ajustando imagen de edición	16
20.- Agregar menú de navegación	17
21.- Aplicando estilos a botones	18
22.- Ajustando formulario create	18
23.- Ajustando formulario edit	20
SECCIÓN 6: Mensajes, cierre y conclusiones	20
24.- Manejo de mensajes de validación	20
25.- Cierre y conclusiones	21

CRUD

PYTHON - MYSQL - FLASK

La intención del video es crear un CRUD¹ con Python usando Flask, enfocándose a aprender el flujo de los datos, BD y el procesamiento de la información.
Se verá como agregar, editar y borrar un empleado, además de validar los datos de entrada.

SECCIÓN 1: Instalación y configuración

1.- Herramientas necesarias

Crear en C: una carpeta llamada SistemaEmpleados, que será la carpeta del proyecto.

Instalar aplicaciones e iniciarlas

Deben instalarse las siguientes aplicaciones:

- Python (para realizar el CRUD): <https://www.python.org/downloads/>
- Flask (para realizar el CRUD): <https://flask.palletsprojects.com/en/1.1.x/>
- Visual Studio Code (para codificar): <https://code.visualstudio.com/>
- XAMPP (para trabajar con la BD): <https://www.apachefriends.org/es/index.html>

Python, VSC y XAMPP se instalan como cualquier programa. Para instalar Flask podemos guiarnos por este tutorial, necesitaremos el símbolo de sistema y la carpeta creada en el paso anterior:

<https://www.youtube.com/watch?v=Zuel9y6OHkE>

Iniciamos XAMPP y activamos MySQL y Apache.

En VSC abrir la carpeta desde File-Open Folder... e instalar las siguientes extensiones:

- **Bootstrap v4 Snippets**: nos permitirá escribir HTML para incluir elementos de Bootstrap.
- **FlaskSnippets**: nos ayudará con ciertas instrucciones que vamos a requerir en el template.
- **Flask-snippets**: a diferencia de la anterior nos completará algunas instrucciones de Python.
- **Jinja2 Snippet Kit**: nos proporcionará instrucciones para implementar en nuestros templates.

Extensiones opcionales: PalenightTheme y Andromeda (para íconos y colores de VSC).

2.- Creando la base de datos

En el navegador escribir localhost/phpmyadmin/, esto nos permitirá crear nuestra base de datos. Seguimos los siguientes pasos:

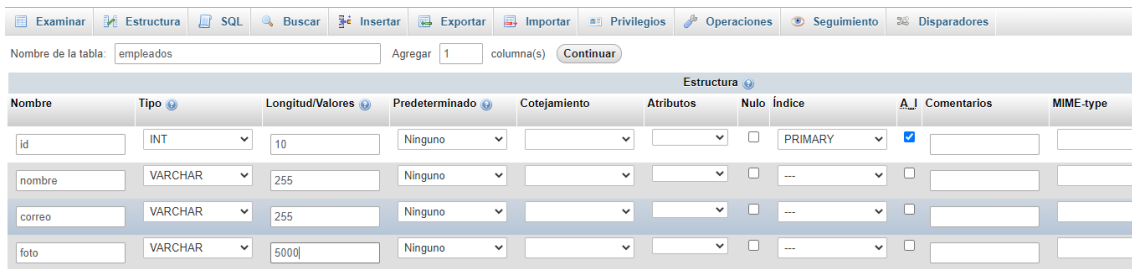
- 1) Ir a Nueva y colocamos como nombre Sistema y utf8mb4_general_ci y tocamos **Crear**
- 2) A la izquierda seleccionamos la base Sistema y creamos nuestra primer tabla llamada Empleados con 4 columnas y luego hacemos clic en **Continuar**

¹El concepto *CRUD* está estrechamente vinculado a la gestión de datos digitales. *CRUD* hace referencia a un acrónimo en el que se reúnen las primeras letras de las cuatro operaciones fundamentales de aplicaciones persistentes en sistemas de bases de datos: a) **Create** (Crear registros); b) **Read/Retrieve** (Leer registros); c) **Update** (Actualizar registros) y d) **Delete/Destroy** (Borrar registros)

Desarrollando una aplicación web con Python y Flask

Tutorial

3) Crearemos los 4 campos de la tabla de la siguiente manera y luego tocamos **Guardar**:



La tabla debería quedarnos de la siguiente manera:

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Extra
<input type="checkbox"/>	1 id	int(10)			No	Ninguna	AUTO_INCREMENT
<input type="checkbox"/>	2 nombre	varchar(255)	utf8mb4_general_ci		No	Ninguna	
<input type="checkbox"/>	3 correo	varchar(255)	utf8mb4_general_ci		No	Ninguna	
<input type="checkbox"/>	4 foto	varchar(5000)	utf8mb4_general_ci		No	Ninguna	

3.- Instalación de paquetes

En VSC vamos a Terminal – New Terminal. Vamos a comprobar si tenemos instalado Python escribiendo Python y dando **Enter**.

```
PS E:\SistemaEmpleados> python
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec 7 2020, 17:08:21) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
```

Si está correctamente instalado debería aparecernos esta imagen

Para salir de la terminal escribimos `exit()` y **Enter**

Vamos a instalar Flask, y para asegurarnos de que vamos a poder instalarlo vamos a utilizar **pip** que es un sistema gestión de paquetes de Python.

En la terminal escribimos **pipinstallflask** y luego para comprobar que haya sido instalado colocaremos **piplist** y nos debería salir en la lista.

Instalaremos otros paquetes:

- Para MySQL utilizaremos: **pipinstallFlask-MySQL**
- Para Jinja2 utilizaremos: **pipinstall jinja2**

Podemos hacer un **piplist** para asegurarnos de que se han instalado estos paquetes.

SECCIÓN 2: Estructuras HTML y conexión

4.- Iniciando una aplicación

Crearemos una carpeta dentro de *SistemaEmpleados* llamada **templates** y dentro de esta otra carpeta llamada **empleados**.

Desde VS crearemos dos archivos:

- `app.py` que guardaremos dentro de *SistemaEmpleados*
- `index.html` que guardaremos dentro de *empleados*

En **index.html** escribiremos **hbs4** y tocaremos TAB para que automáticamente se crea un documento que incluye Bootstrap.

Podemos cambiar el título (*title*) por Inicio y dentro del body escribiremos:

```
Hola Flask!
```

5.- Ejecutando una aplicación

Dentro de `app.py` escribiremos el siguiente código (no son necesarios los comentarios)

```
from flask import Flask
from flask import render_template

app = Flask(__name__)
@app.route('/')
def index():
    return render_template('empleados/index.html')

if __name__ == '__main__':
    app.run(debug=True)
```

Hacemos correr la aplicación desde la terminal, colocando **python app.py**.

De esta manera se crea un servidor con distintas configuraciones, la que más nos importa es esta: Running on <http://127.0.0.1:5000/> que es la dirección donde nosotros vamos a ver el resultado de todo lo que ya hicimos.

En el navegador escribimos esta ruta y vamos a ver el contenido de nuestro documento index.html.

De esta manera se hace una solicitud. Cuando se le pide la solicitud en / se está escribiendo el host 127.0.0.1 con el puerto 5000 que es la definición de lo que es la / y de forma automática se accede al index.html que ya está siendo renderizado por Flask, esto es lo que colocamos en `@app.route('/')`.

6.- Conexión a Base de datos

Para conectarnos directamente a la BD en el archivo app.py escribimos:

```
from flaskext.mysql import MySQL
```

Y en la línea siguiente a `app = Flask(__name__)` escribimos:

```
mysql = MySQL()
app.config['MYSQL_DATABASE_HOST'] = 'localhost'
app.config['MYSQL_DATABASE_USER'] = 'root'
app.config['MYSQL_DATABASE_PASSWORD'] = ''
app.config['MYSQL_DATABASE_DB'] = 'sistema'
mysql.init_app(app)
```

Esto lo hacemos para saber qué datos tenemos para nuestra conexión (servidor, host, usuario y nombre de la BD). Vamos a utilizar por defecto la que tiene XAMPP.

Dentro de `def index()`: generaremos una instrucción SQL para hacer unas pruebas, junto con todo lo necesario para hacer la conexión:

```
sql = """
conn = mysql.connect()
cursor = conn.cursor()
cursor.execute(sql)
conn.commit()
```

¿Cómo sé que está funcionando la conexión con la BD?

Vamos a usar una instrucción de SQL para insertar información:

- 1) Vamos a la BD (localhost) y vamos a la pestaña **Insertar**

- 2) Vamos a escribir en la columna valor para cada campo un nombre, un mail y foto.jpg y hacemos clic en el botón Continuar que aparece inmediatamente debajo del último campo completado.

- 3) De esta manera insertamos la fila y obtenemos la instrucción sql que nos servirá para copiarla al archivo de Python. Ejemplo:









```
INSERT INTO `sistema`.`empleados` (`id`, `nombre`, `correo`, `foto`) VALUES (NULL, 'Juan Pablo', 'juanpablo@gmail.com', 'juanpablo.jpg');
```

- 4) Copiamos esa instrucción generada y la pegamos dentro de las comillas de sql = "":

```
sql = "INSERT INTO `sistema`.`empleados` (`id`, `nombre`, `correo`, `foto`)\nVALUES (NULL, 'Juan Pablo', 'juanpablo@gmail.com', 'juanpablo.jpg');"
```

- 5) Corremos la aplicación desde la terminal (si es que la habíamos cerrado previamente). Recordemos que se hacía con **python app.py** y vamos a ver que desde la BD (localhost) en la pestaña Examinar se agregó una nueva fila:

+ Opciones

 			id	nombre	correo	foto	
<input type="checkbox"/>	 Editar	 Copiar	 Borrar	1	Juan Pablo	juanpablo@gmail.com	juanpablo.jpg
<input type="checkbox"/>	 Editar	 Copiar	 Borrar	2	Juan Pablo	juanpablo@gmail.com	juanpablo.jpg

Cada vez que refresquemos el navegador, se agregará una nueva fila, esto es así porque estamos corriendo la aplicación en la terminal.

7.- Estructuras HTML create y edit

Crearemos las estructuras que nos van a servir para ingresar información y editarla.

Desde VSC crearemos dentro de empleados los siguientes archivos:

- create.html: nos permitirá crear un nuevo registro del empleado.
- edit.html: mostrará la información del empleado

Creando el formulario

Dentro del archivo create.html escribiremos **b-form-entype** para crear el formulario que nos permitirá enviar archivos. Allí cambiaremos el método "get" por "post".

```
<form method="post" action="" enctype="multipart/form-data">\n\n</form>
```

Dentro de este formulario, con **input:text** crearemos dos campos de texto y luego con **input:file** crearemos el siguiente campo para la foto. También agregaremos etiquetas y unos saltos de línea, además de un botón para enviar la información. Los rellenaremos de esta manera:

```
Nombre:\n<input type="text" name="txtNombre" id="txtNombre">\n<br>\nCorreo:\n<input type="text" name="txtCorreo" id="txtCorreo">\n<br>\nFoto:\n<input type="file" name="txtFoto" id="txtFoto">\n<br>\n<br>\n<input type="submit" value="Agregar">
```

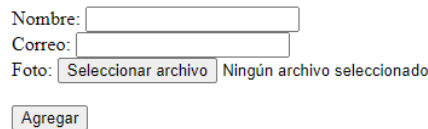
¿Cómo llamamos a nuestro template para que se muestre?

En el archivo app.py ya habíamos creado una referencia con `app.route('/')`. Ahora crearemos en ese mismo archivo otra referencia al archivo `create.html` (lo agregamos debajo del último `return` de la referencia anterior):

```
@app.route('/create')
def create():
    return render_template('empleados/create.html')
```

Lo hacemos correr en la terminal. Recordemos que si ya está corriendo basta con refrescar la página, sino en la terminal debemos colocar `python app.py`

En este caso para ver los resultados debemos ingresar en <http://127.0.0.1:5000/create>



Este es el formulario de creación que nos va a servir para solicitar información al usuario y enviarlo directamente a la BD.

SECCIÓN 3: Recepción y manipulación de datos

8.- Recepción de valores de entrada

Insertaremos toda la información que viene del formulario.

En primer lugar en el archivo `app.py` vamos a agregar en donde está `render_template` el módulo `request`.

Esto lo hacemos porque toda la información que se va a procesar a través del HTML va a ser un envío de información. Este envío se va a manejar como `request` y este `request` es una solicitud de información.

```
from flask import render_template, request
```

Luego, dentro de `create.html`, vamos a decirle al formulario que esta información que está recolectando del usuario la envíe a un lugar, utilizando la url:

```
<form method="post" action="/store" enctype="multipart/form-data">
```

Le agregamos esa ruta `/store` que no existe y debemos crearla en `app.py`. Para ello vamos a escribir **route** debajo del último `route` que utilizamos para el `create`, cambiamos a `store` y podemos copiar el mismo código que utilizamos para agregar el registro.

```
@app.route('/store', methods=['POST'])
def storage():
    sql = "INSERT INTO `sistema`.`empleados` (`id`, `nombre`, `correo`, `foto`) VALUES (NULL, 'Juan Pablo', 'juanpablo@gmail.com', 'juanpablo.jpg');"
    conn = mysql.connect()
    cursor = conn.cursor()
    cursor.execute(sql)
    conn.commit()
    return render_template('empleados/index.html')
```

Estas instrucciones me permiten conectarme a la BD, insertar la información y después redireccionar.

Haremos algunos cambios porque nuestra intención es que tome los datos del archivo create.html y no tome datos fijos, debemos reemplazar esos valores, que van a venir desde el formulario. Para recibir esos datos ya habíamos incluido el request en app.py. Estas líneas vamos a incluirlas a continuación de def storage():

```
_nombre=request.form['txtNombre']
_correo=request.form['txtCorreo']
_foto=request.files['txtFoto']
```

Luego a estos valores que provienen del formulario debemos ingresarlos a la instrucción de inserción. Para esto vamos a hacer algunos cambios en el código anterior:

```
sql = "INSERT INTO `sistema`.`empleados` (`id`, `nombre`, `correo`, `foto`
`) VALUES (NULL, %s, %s, %s);"
datos=(_nombre,_correo,_foto.filename)
```

Con los **%s** lo que hacemos es reemplazar los valores fijos por los valores que el usuario ingresa en el formulario.

Con **datos** definimos las ubicaciones de cada uno de esos campos. Utilizamos filename para la foto porque es un archivo que tiene distintos tipos de datos, como el nombre, el contenido, nosotros necesitamos solamente el nombre.

Para unir estos dos tenemos que ubicarnos en la parte de ejecución y agregar la palabra **datos**, para que cuando ejecute el SQL le ponga los datos:

```
cursor.execute(sql,datos)
```

Volvemos a la terminal, iniciamos nuevamente el servicio con python app.py y con la aplicación corriendo actualizamos la página o ingresamos en <http://127.0.0.1:5000/create> y agregamos datos.

Si no nos da error, deberíamos a ver el mensaje Hola Flask! y en la BD el dato cargado.

9.- Guardar imagen en carpeta uploads

Nosotros ya insertamos información en la BD, pero nos falta adjuntar la fotografía en nuestro sistema.

Vamos a crear una carpeta **uploads** adentro de la carpeta del sistema, allí depositaremos la foto que el usuario adjunte.

Para guardar la foto debemos cambiarle el nombre, porque si hay dos fotos con el mismo nombre se pueden pisar. Para que eso no suceda agregaremos en el archivo app.py lo siguiente al principio del documento:

```
from datetime import datetime #Nos permitirá darle el nombre a la foto
```

Luego concatenaremos el tiempo al nombre de la foto y preguntaremos si esa foto existe. Dentro de def storage() y luego de _foto=request.files['txtFoto'] agregaremos las siguientes líneas:

```
now= datetime.now()
tiempo= now.strftime("%Y%H%M%S")

if _foto.filename!='':
```

```
nuevoNombreFoto=tiempo+_foto.filename  
_foto.save("uploads/"+nuevoNombreFoto)
```

En primer lugar obtuvimos la fecha y hora actual con `datetime.now()` (que será la de la subida de la foto), luego lo convertimos a un formato determinado, que es el formato de años, horas, minutos y segundos. De esta manera nos aseguramos que el nombre sea siempre distinto. Además preguntamos que, si hay una foto adjuntada, entonces que se genere el nuevo nombre de la foto, concatenando el tiempo en el que fue subida con su nombre. Finalmente, que se guarde la foto en la carpeta uploads.

Por último vamos a reemplazar `datos=(_nombre, _correo, _foto.filename)` por:

```
datos=( _nombre, _correo, nuevoNombreFoto)
```

Lo hacemos correr en <http://127.0.0.1:5000/create> y para comprobar que se haya subido la imagen vamos a buscarla en la carpeta uploads.

10.- Consultando datos de tabla empleados

Luego de guardar la foto consultaremos toda la información de la BD y la mostraremos en la terminal.

Recordemos que tenemos una sentencia SQL que insertaba los datos dentro de `def index()`. La cambiaremos por la siguiente:

```
sql = "SELECT * FROM `sistema`.`empleados`;"
```

Luego agregaremos debajo de `cursor.execute(sql)` lo siguiente, para poder recuperar la información y mostrarla en la terminal. Para esto utilizaremos `fetchall()`:

```
empleados=cursor.fetchall()  
print(empleados)
```

Al ejecutar en <http://127.0.0.1:5000/> y volver a Visual Studio Code se muestran los datos en la terminal. Esta información ya está lista para mostrarse en el `index.html`, en formato de tabla.

11.- Mostrando datos en HTML

Ahora mostraremos los datos de la BD en nuestro documento `index.html`

Vamos a hacer el envío de datos a través del `render_template`, para eso modificamos en `def index()` esa instrucción:

```
return render_template('empleados/index.html', empleados=empleados)
```

Le enviamos esa información a través del template. Vamos a enviarle una variable con el valor de `empleados` y le vamos a enviar ese mismo valor que hemos recuperado de esa BD.

En el `index.html` agregaremos la tabla en el body utilizando Bootstrap y la extensión que descargamos al principio. Para esto escribimos `b-table-header` y cuando se crea la estructura básica la llenamos de la siguiente manera:

```
<table class="table table-light">  
  <thead class="thead-light">  
    <tr>  
      <th>#</th>  
      <th>Foto</th>  
      <th>Nombre</th>  
      <th>Correo</th>  
      <th>Acciones</th>  
    </tr>
```



```
</thead>
<tbody>
    {% for empleado in empleados %}
    <tr>
        <td>{{empleado[0]}}</td>
        <td>{{empleado[3]}}</td>
        <td>{{empleado[1]}}</td>
        <td>{{empleado[2]}}</td>
        <td>Editar | Borrar</td>
    </tr>
    {% endfor %}
</tbody>
</table>
```

La tabla tiene 5 columnas porque al ser un CRUD agregaremos una columna de acciones sobre los registros.

Agregamos también con **jfor** que nos lo proporciona la extensión **jinja2**. Ese for recorrerá todos los empleados de la tabla e imprimiremos las distintas columnas (campos) empezando por el campo 0 (id).

Una vez que vamos a <http://127.0.0.1:5000/> veremos que el resultado sería algo similar a esto:

#	Foto	Nombre	Correo	Acciones
1	juanpablo.jpg	Juan Pablo	juanpablo@gmail.com	Editar Borrar
2	juanpablo.jpg	Juan Pablo	juanpablo@gmail.com	Editar Borrar
3	juanpablo.jpg	Juan Pablo	juanpablo@gmail.com	Editar Borrar
4	juanpablo.jpg	Juan Pablo	juanpablo@gmail.com	Editar Borrar
5	juanpablo.jpg	Juan Pablo	juanpablo@gmail.com	Editar Borrar
6	9b4369dd-d1fb-492e-b280-feab4df50246.jpg	Juan	correo@correo.com	Editar Borrar
7	2021125950z_021.jpg	Juampi	juanpi@gmail.com	Editar Borrar
8	juanpablo.jpg	Juan Pablo	juanpablo@gmail.com	Editar Borrar

12.- Eliminando datos de la tabla

Vamos a eliminar los registros por ID utilizando la última columna de “Acciones”. Agregaremos la siguiente etiqueta que nos a permitir eliminar el registro dentro de index.html en donde escribimos Editar | :

```
<td>Editar | 
<a href="/destroy/{{empleado[0]}}">Eliminar</a>
</td>
```

Por otro lado dentro del archivo app.py debemos agregar donde importábamos el `render_template` y el request la palabra **redirect** que nos va a permitir redireccionar una vez que eliminemos el registro, es decir regresar a la URL desde donde vino.

```
from flask import render_template, request, redirect
```

Además crearemos el borrado del registro inmediatamente arriba de `@app.route('/create')`:

```
@app.route('/destroy/<int:id>')
def destroy(id):
    conn = mysql.connect()
    cursor = conn.cursor()
```

```
cursor.execute("DELETE FROM `sistema`.`empleados` WHERE id=%s", (id))
conn.commit()
return redirect('/')
```

Para probar como funciona lo ejecutamos actualizando la página y probamos el botón eliminar.

13.- Recuperación de datos para editar

Vamos a enviar la información para poder editarla. Agregaremos entonces la posibilidad de Editar el registro para que pueda ser editado utilizando el formulario.

En index.html vamos a hacer una copia del código que utilizamos para eliminar y vamos a hacer algunos cambios para que nos quede de esta manera:

```
<td><a href="/edit/{empleado[0]}">Editar</a> |
    <a href="/destroy/{empleado[0]}">Eliminar</a>
</td>
```

En app.py vamos a escribir debajo de las instrucciones que nos permitían eliminar la palabra **route** y presionamos TAB para que se escriba parte del código que modificaremos de esta manera:

```
@app.route('/edit/<int:id>')
def edit(id):
    return render_template('empleados/edit.html')
```

Luego en edit.html crearemos nuestro formulario. Para probar simplemente escribiremos:
Formulario para editar

Podemos probar y veremos que aparecen los vínculos para editar en la columna Acciones y cuando presionamos en cualquiera de ellos iremos a edit.html. En la dirección URL ya nos está apareciendo el ID que vamos a editar.

Vamos a consultar la base de datos, así que haremos lo mismo que hicimos anteriormente con delete. El código completo que debe ir en app.py es el siguiente y podemos agregarlo a continuación del *destroy*:

```
@app.route('/edit/<int:id>')
def edit(id):
    conn = mysql.connect()
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM `sistema`.`empleados` WHERE id=%s", (id
))
    empleados=cursor.fetchall()
    conn.commit()
    return render_template('empleados/edit.html', empleados=empleados)
```

Volvemos a edit.html y vamos a agregar un **jfor** que recorra la lista de empleados (recuerden que se puede escribir jfor y presionar TAB):

```
Formulario para editar
{% for empleado in empleados %}
    {{ empleado[0] }}
    {{ empleado[1] }}
    {{ empleado[2] }}
    {{ empleado[3] }}
{% endfor %}
```

De esta manera estaremos mostrando (sin formato) los datos del empleado que queremos cambiar. Podemos probarlo si vamos a <http://127.0.0.1:5000/> y hacemos clic en Editar.

Para imprimir toda esta información vamos a ir a create.html y vamos a copiar todo el código del formulario y lo vamos a agregar en edit.html dentro del for, para que la información que va a recuperar la va a poner directamente en los valores y estos valores nos van a servir para mostrar al usuario qué valores tienen dichos registros.

Incorporaremos en todos la propiedad value="{{ empleado[...] }}" donde en ... colocaremos la posición de la columna. Esta propiedad nos permite asignar el valor que tendrá cada uno de los inputs del formulario.

En la foto no debemos agregar el value dentro del input del formulario, sino que lo agregamos aparte:

```
{% for empleado in empleados %}

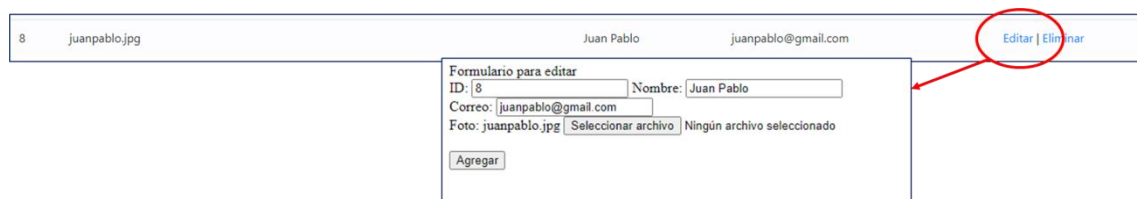
<form method="post" action="/store" enctype="multipart/form-data">

    ID:
    <input type="text" value="{{ empleado[0] }}" name="txtID" id="txtID">
    Nombre:
    <input type="text" value="{{ empleado[1] }}" name="txtNombre" id="txt
Nombre">
    <br>
    Correo:
    <input type="text" value="{{ empleado[2] }}" name="txtCorreo" id="txt
Correo">
    <br>
    Foto:
    {{ empleado[3] }}
    <input type="file" name="txtFoto" id="txtFoto">
    <br>
    <br>
    <input type="submit" value="Agregar">

</form>

{% endfor %}
```

Podemos probar el resultado y veremos que si hacemos clic en Editar del registro 8 estaremos listos para modificarlo:



ID	Nombre	Correo	Foto	Acciones
8	Juan Pablo	juanpablo@gmail.com	juanpablo.jpg	Editar Eliminar

Formulario para editar

ID: 8 Nombre: Juan Pablo

Correo: juanpablo@gmail.com

Foto: juanpablo.jpg | Seleccionar archivo | Ningún archivo seleccionado

Agregar

14.- Guardando la actualización

Vamos a hacer unos ajustes dentro del formulario de edit.html

En primer lugar cambiaremos la acción del formulario por update:

```
<form method="post" action="/update" enctype="multipart/form-data">
```

Además vamos a tener que crear una ruta que va a recibir todos los datos que están en el formulario de actualización (id, nombre, correo y foto).

En el archivo app.py entre el route del create y el route del edit vamos a crear un **route** para recibir los datos al igual que lo hicimos con *storage*.

```
@app.route('/update', methods=['POST'])
def update():
    _nombre=request.form['txtNombre']
    _correo=request.form['txtCorreo']
    _foto=request.files['txtFoto']
    id=request.form['txtID']

    sql = "UPDATE `sistema`.`empleados` SET `nombre`=%s, `correo`=%s WHERE id=%s;"
    datos=(_nombre,_correo,id)

    conn = mysql.connect()
    cursor = conn.cursor()

    cursor.execute(sql,datos)
    conn.commit()

    return redirect('/')
```

En las primeras cuatro líneas recibimos todos los datos, luego en las variables SQL actualizamos los datos (la foto la colocaremos más adelante), nos conectamos a la base de datos y ejecutamos la sentencia de actualización.

Podemos probarlo desde <http://127.0.0.1:5000/> y hacemos clic en Editar, cambiamos el nombre o el correo y confirmamos desde el botón Agregar.

SECCIÓN 4: Manipulación de foto e inclusión de HTML externo

15.- Modificando foto

Cuando el usuario haga la actualización de los datos debemos verificar si el campo de foto tiene una foto puesta.

Cuando el usuario quiera cambiar la foto tenemos que borrar la foto vieja y colocar la foto nueva, para esto debemos generar el acceso a la foto y a la carpeta.

Vamos a importar en app.py un módulo del sistema operativo que nos va permitir eliminar ese archivo.

```
import os
```

Además, antes del primer route vamos a crear una referencia a la carpeta utilizando algunas propiedades del módulo que acabamos de importar.

```
CARPETA= os.path.join('uploads')
app.config['CARPETA']=CARPETA
```

Dentro de def update() vamos a ubicarnos debajo de cursor = conn.cursor() y a codificar el nombre del archivo como lo hicimos cuando lo creábamos:

```
now= datetime.now()
tiempo= now.strftime("%Y%H%M%S")
```

También, como lo hacíamos en storage preguntaremos si la foto existe, para generar el nuevo nombre del archivo y guardarla:

```
if _foto.filename!='':
    nuevoNombreFoto=tiempo+_foto.filename
    _foto.save("uploads/"+nuevoNombreFoto)
```

Además debemos recuperar la foto y actualizar solamente ese campo de foto, con lo cual vamos a necesitar ejecutar una instrucción SQL que seleccione los datos de esa foto a través del ID (dentro del if):

```
cursor.execute("SELECT foto FROM `sistema`.`empleados` WHERE id=%s", id)
fila= cursor.fetchall()
```

Continuando dentro del if finalmente vamos a remover la foto. Con *remove* tomamos la carpeta y la unimos con la fila que recuperamos, que se encuentra en la posición 0 y la fila 0. Con *execute* actualizamos la tabla solo para el id seleccionado y cerramos la conexión. Finalmente con *redirect* le indicamos que regresamos adonde estábamos antes.

```
os.remove(os.path.join(app.config['CARPETA'], fila[0][0]))
cursor.execute("UPDATE `sistema`.`empleados` SET foto=%s WHERE id=%s", (nuevoNombreFoto, id))
conn.commit()
return redirect('/')
```

Para probarlo debemos hacerlo con un registro que efectivamente tenga una foto guardada dentro de uploads. Ejecutamos y vamos a Editar ese registro, seleccionamos la nueva foto con "Seleccionar archivo" y la cambiamos haciendo clic en Agregar. Si miramos la carpeta uploads tiene la nueva foto y se ha borrado la foto anterior, también se ha cambiado en la tabla.

Código completo de update:

```
@app.route('/update', methods=['POST'])
def update():
    _nombre=request.form['txtNombre']
    _correo=request.form['txtCorreo']
    _foto=request.files['txtFoto']
    id=request.form['txtID']

    sql = "UPDATE `sistema`.`empleados` SET `nombre`=%s, `correo`=%s WHERE id=%s;"
    datos=(_nombre,_correo,id)

    conn = mysql.connect()
    cursor = conn.cursor()

    now= datetime.now()
    tiempo= now.strftime("%Y%H%M%S")
```

```

if _foto.filename!='':
    nuevoNombreFoto=tiempo+_foto.filename
    _foto.save("uploads/"+nuevoNombreFoto)

    cursor.execute("SELECT foto FROM `sistema`.`empleados` WHERE id=%s", id)
    fila= cursor.fetchall()

    os.remove(os.path.join(app.config['CARPETA'], fila[0][0]))
    cursor.execute("UPDATE `sistema`.`empleados` SET foto=%s WHERE id=%s;", (nuevoNombreFoto, id))
    conn.commit()

    cursor.execute(sql, datos)
    conn.commit()

    return redirect('/')

```

16.- Borrar fotos en uploads

Una vez que eliminamos un empleado en la tabla debemos eliminar físicamente su foto de la carpeta Uploads.

Para esto vamos a aprovechar parte del código que utilizamos en *update* y lo vamos a agregar en *destroy*. Entre `cursor = conn.cursor()` y `cursor.execute("DELETE FROM..."`)

```

cursor.execute("SELECT foto FROM `sistema`.`empleados` WHERE id=%s", id)
fila= cursor.fetchall()
os.remove(os.path.join(app.config['CARPETA'], fila[0][0]))

```

En la primera línea hacemos una selección de los datos, una vez encontrado con `fetchall()` tomamos toda la información y luego removemos la foto.

De esta manera el archivo con la foto ya no aparecerá en la carpeta *uploads*.

17.- Incluir archivos header y footer

Para hacer esta parte es importante tener algunos datos cargados en la tabla.

La idea de esta parte es que armemos un header y un footer que se repita en el formulario *create* y en el formulario *edit*.

En primer lugar crearemos los archivos `header.html` y `footer.html` dentro de la carpeta *templates*. En ambos archivos vamos a trasladar (cortar) todo el código que queremos que aparezca tanto dentro del encabezado como del pie y luego haremos una referencia a este código dentro de cada una de las páginas.

Para el header:

Cortamos el código de la cabecera del archivo `index.html` hasta el `<body>` inclusive y la pegamos en `header.html`:

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">

```

```
<meta name="viewport" content="width=device-width, initial-  
scale=1.0">  
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstra  
p/4.0.0/css/bootstrap.min.css">  
<title>Inicio</title>  
</head>  
<body>
```

Para el footer:

Cortamos el código del pie de página del archivo index.html desde el </body> y la pegamos en footer.html:

```
</body>  
</html>
```

Solamente quedará la tabla dentro del index.html.

Agregaremos más código dentro de cada archivo:

En **header.html**, debajo del <body>:

```
<div class="container">  
  Cabecera del sitio
```

En **footer.html**, arriba del </body>:

```
</div>  
Pie de página
```

Ahora incluiremos el header y el footer en la parte de arriba del index.html:

En la primer línea agregaremos (escribiendo *jinclude* y TAB se agrega solo):

```
{% include 'header.html' %}
```

En la última línea agregaremos:

```
{% include 'footer.html' %}
```

De esta manera le estamos diciendo que incluya todo lo que está en el header y en el footer. Podemos probarlo ingresando en <http://127.0.0.1:5000/>, observaremos que en nuestro index aparecen el encabezado y el pie de página.

Para terminar, debemos agregar estas últimas dos líneas al principio y al final de los archivos create.html y edit.html y probar que aparezcan el encabezado y el pie.

18.- Mostrando imágenes

Las imágenes aún no se muestran dentro de la página ya que por ahora lo que hicimos fue colocar el nombre del archivo dentro de la tabla, pero no el archivo en sí. Para ello debemos agregar la etiqueta img y además hacer unos ajustes dentro de app.py.

En primer lugar, vamos a arreglar un problema que aparece al agregar un empleado. Cuando lo hacemos no nos redirecciona nuevamente al index, sino que nos muestra el template.

En la última línea de *defstorage()*: cambiaremos esta última línea:

```
return render_template('empleados/index.html')
```

Por esta línea:

```
return redirect('/')
```

Ahora vamos a hacer que la imagen aparezca en la tabla, entonces dentro del index.html agregaremos dentro de la tabla reemplazaremos esta línea:

```
<td>{{empleado[3]}}</td>
```

Por esta línea:

```
<td>
    
</td>
```

Sin embargo, al actualizar veremos que la imagen aún no se ve:

Cabecera del sitio

#	Foto
7	
9	

Pie de página

Esto sucede porque en Flask aún no importamos ese acceso a la carpeta, por lo tanto Flask no les está dejando entrar en las carpetas.

Para que Flask pueda interpretar lo que se le está solicitando vamos a agregar este módulo en las primeras líneas de app.py:

```
from flask import send_from_directory
```

Además, dentro de app.py crearemos el acceso a la carpeta uploads. Crearemos la URL utilizando *route* (lo agregaremos arriba de las líneas que dicen *def index():* y *@app.route('/')*):

```
@app.route('/uploads/<nombreFoto>')
def uploads(nombreFoto):
    return send_from_directory(app.config['CARPETA'], nombreFoto)
```

¿Para qué hacemos esto? Para generar el acceso a la carpeta uploads. El método uploads que creamos nos dirige a la carpeta (variable CARPETA) y nos muestra la foto guardada en la variable nombreFoto.

Volvemos a probar y ya veremos que aparecerán las fotos del lado izquierdo.

SECCIÓN 5: Ajustes y navegación

19.- Ajustando imagen de edición

El objetivo será que la imagen se vea en el momento de editar el registro dentro de edit.html. En edit.html identificaremos la línea donde colocábamos el nombre del archivo de la foto:

```
{{ empleado[3] }}
```

Y escribiremos **b-img-thumbnail** y presionaremos TAB para que nos aparezca todo el código. Este es el código que debemos copiar debajo de la palabra Foto:

```
Foto:
<br>
    
<br>
```

Las etiquetas *
* son simplemente para acomodar la foto. Luego dentro de src tenemos que utilizar un módulo de flask que se llama *url_for* que lo que hace es acceder a un método que tenemos en app.py y es el método *def uploads(nombreFoto)*. Después de colocar 'uploads' debemos enviarle el nombre de la foto con *nombreFoto=empleado[3]*.

Colocamos `empleado[3]` porque lo que nos interesa es que se traiga el campo que contiene la foto de la base de datos (columna número 4, pero colocamos el 3 porque las columnas se numeran desde el 0).

Además incorporamos `width="100"` para que la foto no sea tan grande.

Para que todo esto funcione debemos importar el módulo `url_for` que lo agregaremos en el archivo `app.py` en la línea donde importábamos `render_template`, `request` y `redirect`:

```
from flask import render_template, request, redirect, url_for
```

Este es el resultado hasta el momento:

Cabecera del sitio Formulario para editar

ID:

Nombre:

Correo:

Foto:



Ningún archivo seleccionado

Pie de página

20.- Agregar menú de navegación

Este menú nos permitirá navegar entre las páginas de nuestro sitio.

En `header.html` entre el `<body>` y el `<div>` escribiremos **b-navbar** y presionaremos TAB. Dentro del código haremos varios cambios:

- Quitaremos `fixed-top` para que no quede flotante.
- Reemplazaremos donde dice Brand por "Gestión de empleados"
- En el primer elemento de la lista reemplazaremos Item 1 por "Empleados" y para acceder a la url en href cambiaremos por `href="{{url_for('index')}}"`

El código completo de la barra de navegación es el siguiente:

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <a class="navbar-brand">Gestión de empleados</a>
  <button class="navbar-toggler" data-target="#my-nav" data-
toggle="collapse" aria-controls="my-nav" aria-expanded="false" aria-
label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>
  <div id="my-nav" class="collapse navbar-collapse">
    <ul class="navbar-nav mr-auto">
      <li class="nav-item active">
        <a class="nav-
link" href="{{url_for('index')}}">Empleados <span class="sr-
only">(current)</span></a>
      </li>
      <li class="nav-item">
        <a class="nav-link disabled" href="#" tabindex="-
1" aria-disabled="true">Item 2</a>
      </li>
    </ul>
  </div>
```

```
</nav>
```

Gestión de empleados Empleados Item 2

Cabecera del sitio

#	Foto	Nombre	Correo	Acciones
---	------	--------	--------	----------

Este es el resultado de agregar la barra de navegación

21.- Aplicando estilos a botones

En index.html agregaremos un botón que nos va a permitir ingresar al formulario de creación del empleado. Además le daremos estilo a Editar y Eliminar, agregando un pop-up de confirmación.

En index.html debajo de {% include 'header.html' %} agregaremos lo siguiente:

```
<br>
<a href="{{url_for('create')}}" class="btn btn-success">
    Ingresar nuevo empleado
</a>
<br>
<br>
```

Recordemos que con el url_for podemos ir a la página de creación del empleado (create.html) y además le agregaremos una clase de Bootstrap para el botón. Los
 son simplemente para acomodar el contenido.



Además, a los botones de Editar y Eliminar les agregaremos el estilo de los botones de Bootstrap y un pop-up de confirmación:

```
<td>
<a class="btn btn-warning" href="/edit/{{empleado[0]}}">Editar</a> |
    <a class="btn btn-
danger" onclick="return confirm('¿Desea borrar al empleado?')" href="/des
troy/{{empleado[0]}}">Eliminar</a>
</td>
```

Recordemos que onclick se utiliza para desencadenar una acción cuando hagamos clic con el mouse en el botón. En este caso pedimos que retorne el cuadro confirm preguntando si deseamos borrar al empleado.

Cabecera del sitio

Ingresar nuevo empleado

#	Foto	Nombre	Correo	Acciones
7		Juampi	mail@gmail.com	Editar Eliminar
9		Pedro	pedro@mail.com	Editar Eliminar

22.- Ajustando formulario create

Tanto en el header como en el footer quitaremos las leyendas “Cabecera del sitio” y “Pie de página” que nos servían como referencia para saber que estábamos creándolos.

Trabajaremos ahora sobre create.html para agregarles estilos de Bootstrap al formulario de registro de un empleado.

Dentro del formulario agregaremos una tarjeta (**b-card-header**) e iremos modificándoles los datos:

- En header cambiaremos Header por “Ingresar empleados”
- Para el título cambiaremos Title por “Datos del empleado”
- Adentro del párrafo que tiene la palabra Content pasaremos todo lo que queda del formulario, desde el Nombre hasta el botón Agregar.

Gestión de empleados Empleados Item 2

Ingresar empleados

Datos del empleado

Nombre:

Correo:

Foto: Ningún archivo seleccionado

De esta forma viene quedando nuestro formulario

Dentro del `<p class="card-text">` vamos a agregar 4 grupos (`b-form-group`) para agrupar tanto los datos que se deben ingresar como el botón de Agregar:

```
<div class="form-group">
<label for="txtNombre">Nombre:</label>
  <input id="txtNombre" class="form-control" type="text" name="txtNombre">
</div>
```

Lo que estamos haciendo es reemplazar el input antiguo por el nuevo formato. Agregaremos el `b-form-group` para el correo:

```
<div class="form-group">
<label for="txtCorreo">Correo:</label>
<input id="txtCorreo" class="form-control" type="text" name="txtCorreo">
</div>
```

Haremos lo mismo con la foto, con la salvedad de que en vez de colocar como argumento de **type** el tipo `text` escribiremos el tipo `file` porque nos permitirá agregar el archivo de la foto.

```
<div class="form-group">
<label for="txtFoto">Foto:</label>
<input id="txtFoto" class="form-control" type="file" name="txtFoto">
</div>
```

Finalmente colocaremos un **b-form-group** para dos botones con estilo de Bootstrap: el de Agregar y el de Regresar. El botón de regresar tendrá un vínculo a la página `index.html` utilizando un `url_for` como lo veníamos haciendo anteriormente.

```
<div class="form-group">
<input type="submit" class="btn btn-success" value="Agregar">
  <a href="{{url_for('index')}}" class="btn btn-
primary">Regresar</a>
</div>
```

23.- Ajustando formulario edit

Dentro de edit.html también vamos a agregar algunos estilos, por lo tanto, vamos a utilizar un mecanismo similar que en el formulario create.

Agregaremos una tarjeta (**b-card-header**) e iremos modificándoles los datos:

- En header cambiaremos Header por “Editar empleado”
- Para el título cambiaremos Title por “Datos del empleado”
- Adentro del párrafo que tiene la palabra Content pasaremos todo lo que queda del formulario, desde el Nombre hasta el botón Agregar.

Dentro del `<p class="card-text">` vamos a agregar 4 grupos (**b-form-group**) para agrupar tanto los datos que se deben actualizar como el botón de Modificar. En el caso del ID vamos a ocultarlo cambiando el atributo de type a hidden:

```
<input type="hidden" value="{{ empleado[0] }}" name="txtID" id="txtID">
```

Luego seguiremos con el Nombre y Correo, agregando como atributo value en cada uno la columna donde figura el nombre del empleado y el correo respectivamente:

```
<div class="form-group">
<label for="txtNombre">Nombre:</label>
  <input id="txtNombre" value="{{ empleado[1] }}" class="form-
control" type="text" name="txtNombre">
</div>
```

```
<div class="form-group">
  <label for="txtCorreo">Correo:</label>
  <input id="txtCorreo" value="{{ empleado[2] }}" class="form-
control" type="text" name="txtCorreo">
</div>
```

Para la foto procederemos de modo similar, agregando entre el `<label>` y el `<input>` la foto del empleado con una etiqueta `img`:

```
<div class="form-group">
<label for="txtFoto">Foto:</label>
  
  <input id="txtFoto" class="form-control" type="file" name="txtFoto">
</div>
```

Para los botones pueden copiarse las mismas líneas de código que en el formulario create.html:

```
<div class="form-group">
<input type="submit" class="btn btn-success" value="Modificar">
  <a href="{{ url_for('index') }}" class="btn btn-
primary">Regresar</a>
</div>
```

SECCIÓN 6: Mensajes, cierre y conclusiones

24.- Manejo de mensajes de validación

Incorporaremos el módulo *flash* para enviar mensajes dentro de app.py:

```
from flask import render_template, request, redirect, url_for, flash
```

Además, debajo de `app = Flask(__name__)` agregaremos una contraseña ya que estamos trabajando con envío de contenido:

```
app.secret_key="ClaveSecreta"
```

Dentro de `def storage()` agregaremos una pequeña validación de los datos para asegurarnos que todos los datos hayan sido ingresados. Esto lo haremos arriba de `now = datetime.now()`:

```
if _nombre == '' or _correo == '' or _foto == '':  
    flash('Recuerda llenar los datos de los campos')  
    return redirect(url_for('create'))
```

Para recibir los mensajes utilizaremos la siguiente porción de código dentro de `create.html`, que puede colocarse debajo de la primera línea:

```
{% include 'header.html' %}  
  
{% with messages= get_flashed_messages() %}  
  
{% if messages %}  
    <div class="alert alert-danger" role="alert">  
        {% for message in messages %}  
            {{message}}  
        {% endfor %}  
    </div>  
{% endif %}  
  
{% endwith %}
```

Lo que hacemos es recibir todos los mensajes utilizando flash, luego preguntamos si hay mensajes y si existen los vamos a ir desglosando de uno en uno. Agruparemos todos los mensajes y con un for vamos a leer mensaje por mensaje de tal forma que, si enviamos para la validación del nombre, de la foto o el correo podríamos ponerlos en diferentes *flashes*, es decir que si no escribe bien el correo que envíe un dato en flash, si no escribe bien su nombre que envíe otro, etc.

En este caso simplemente está validando que nada de esto esté vacío.

25.- Cierre y conclusiones

En este CRUD se podrían agregar otras cosas como validación, arreglar algunos detalles estéticos y revisar algunos conceptos de seguridad.