

Apunte CRUD

Temas:

Proyecto Final 1 - Manejo de excepciones y Módulos y paquetes

- Introducción al Framework a utilizar.
- Introducción al Proyecto CRUD.
- Instalación y configuración.
- Creación de un proyecto utilizando el Framework.

Proyecto Final 2 - Continuando con el Proyecto Final

- MVC y MVT.
- Conexión con la Base de Datos.
- Templates.
- Creación de modelos.

Proyecto Final 3 - Finalizando con el Proyecto Final

- Formularios.
 - Ajustes finales.
 - Retrospectiva Proyecto CRUD y Framework utilizado.
-

1 - ¿Qué es un CRUD?

En programación solemos usar el término CRUD para referirnos a las operaciones básicas que puedes realizar sobre un conjunto de datos y por sus siglas son:

Crear (c por Create) nuevos registros. Cuando hablamos de bases de datos esto quiere decir insertar información.

Leer (r por Read). Esto quiere decir consultar esa información, ya sea un registro o una colección de estos registros.

Actualizar (u por Update), que significa tomar un registro que ya existe en la base de datos y modificar alguna de las columnas.

Por último, **eliminar** registros (d por Delete), que significa tomar un registro y quitarlo del almacén.

Los CRUDs son programas que involucran todas estas operaciones sobre una entidad, como en nuestro caso en el que trabajaremos con datos de empleados.

2 - ¿Qué es flask?

En la actualidad existen muchas opciones para crear páginas web y muchos lenguajes (PHP, JAVA), y en este caso Flask nos permite crear de una manera muy sencilla aplicaciones web con Python.

Flask es un “micro” Framework escrito en Python y concebido para facilitar el desarrollo de Aplicaciones Web bajo el patrón **MVC** (Model - View - Controller o Modelo - Vista - Controlador). El patrón **MVC** es una manera o una forma de trabajar que permite diferenciar y separar lo que es el modelo de datos (los datos que van a tener la App que normalmente están guardados en BD), la vista (página HTML) y el controlador (donde se gestiona las peticiones de la app web).

La palabra “micro” no designa a que sea un proyecto pequeño o que nos permita hacer páginas web pequeñas sino que al instalar Flask tenemos las herramientas necesarias para crear una aplicación web funcional pero si se necesita en algún momento una nueva funcionalidad hay un conjunto muy grande extensiones (plugins) que se pueden instalar con Flask que le van dotando de funcionalidad.

En principio, en la instalación no se tienen todas las funcionalidades que se pueden necesitar pero de una manera muy sencilla se puede extender el proyecto con nuevas funcionalidades por medio de plugins.

¿Por qué usar Flask?

- **Flask es un “micro” Framework:** Para desarrollar una App básica o que se quiera desarrollar de una forma ágil y rápida Flask puede ser muy conveniente, para determinadas aplicaciones no se necesitan muchas extensiones y es suficiente.
- **Incluye un servidor web de desarrollo:** No se necesita una infraestructura con un servidor web para probar las aplicaciones sino de una manera sencilla se puede correr un servidor web para ir viendo los resultados que se van obteniendo.
- **Tiene un depurador y soporte integrado para pruebas unitarias:** Si tenemos algún error en el código que se está construyendo se puede depurar ese error y se puede ver los valores de las variables. Además está la posibilidad de integrar pruebas unitarias.
- **Es compatible con Python3.**
- **Es compatible con wsgi:** Wsgi es un protocolo que utiliza los servidores web para servir las páginas web escritas en Python.

- **Buen manejo de rutas:** Cuando se trabaja con Apps Web hechas en Python se tiene el controlador que recibe todas las peticiones que hacen los clientes y se tienen que determinar que ruta está accediendo el cliente para ejecutar el código necesario.
- **Soporta de manera nativa el uso de cookies seguras.**
- **Se pueden usar sesiones.**
- **Sirve para construir servicios web** (como APIs REST) o aplicaciones de contenido estático.
- **Flask es Open Source** y está amparado bajo una licencia BSD.
- **Buena documentación,** código de GitHub y lista de correos.

Fuente: <https://openwebinars.net/blog/que-es-flask/>

3 - Herramientas necesarias

Debemos tener instaladas las siguientes aplicaciones:

- **Python** (Link de descarga: <https://www.python.org/downloads/>)
- **Visual Studio Code** (Link de descarga: <https://code.visualstudio.com/>)
- **XAMPP** (Link de descarga: <https://www.apachefriends.org/es/index.html>)

Python, VSCode y XAMPP posiblemente los tengamos instalados.

También necesitamos instalar algunos paquetes en Python. Para ello vamos a utilizar **pip** , que es un sistema gestión de paquetes de Python.

- Para instalar **Flask** utilizaremos:

```
pip install flask
```

y luego para comprobar que haya sido instalado colocaremos:

```
pip list
```

y nos debería salir en la lista.

Instalaremos otros paquetes:

- Para instalar **MySQL** utilizaremos:

```
pip install Flask-MySQL
```

- Para instalar **Jinja2** utilizaremos:

```
pip install jinja2
```

Luego podemos hacer un:

```
pip list
```

para asegurarnos de que se han instalado estos paquetes.

En VSC vamos a instalar la siguientes extensión:

- **Python Extension Pack:**

Este pack contiene varias extensiones útiles:

- **Python** - Linting, Debugging (multi-threaded, remote), Intellisense, code formatting, refactoring, unit tests, snippets, Data Science (with Jupyter), PySpark and more.
- **Jinja** - Jinja template language support for Visual Studio Code.
- **Django** - Beautiful syntax and scoped snippets for perfectionists with deadlines.
- **Visual Studio IntelliCode** - Provides AI-assisted productivity features for Python developers in Visual Studio Code with insights based on understanding your code combined with machine learning..
- **Python Environment Manager** - Provides the ability to view and manage all of your Python environments & packages from a single place.
- **Python Docstring Generator** - Quickly insert Python comment blocks with contextually inferred parameters for classes and methods based on multiple, selectable template patterns.
- **Python Indent** - Correct python indentation in Visual Studio Code.
- **Jupyter** - Provides Jupyter notebook support for Python language, used for data science, scientific computing and machine learning.

De manera opcional, podemos instalar **Palenight Theme** y **Andromeda**, que cambia íconos y colores en VSC.

Hecho esto, ya podemos iniciar **XAMPP** y desde allí activamos **MySQL** y **Apache**.

4 - Estructura de carpetas

En este proyecto vamos a necesitar la estructura de carpetas que se describe a continuación:

- Una carpeta principal, llamada **SistemaEmpleados**,
- Una subcarpeta llamada **templates**, y dentro de esta, una llamada **empleados**.
- Una subcarpeta de **SistemaEmpleados** que se llame **uploads**

Dentro de la carpeta **empleados** almacenaremos todos los archivos HTML que componen nuestra aplicación.
Dentro de la carpeta **uploads** guardaremos las fotografías de los empleados.

5 - Ejecutando una aplicación de prueba

Este es un buen momento para comprobar que todo lo que hemos instalado y configurado funciona correctamente. Vamos a crear una pequeña aplicación y ejecutarla. Escribimos el siguiente script, y lo guardamos dentro de la carpeta **SistemaEmpleados**:

app_test.py

```
# Importamos el framework Flask
from flask import Flask

# Importamos la función que nos permit el render de los templates
from flask import render_template

# Creamos la aplicación
app = Flask(__name__)

# Proporcionamos la ruta a la raíz del sitio
@app.route('/')
def index():
    # Devolvemos código HTML para ser renderizado
    return "<h1>Hola Codo a Codo!</h1>Todo a salido bien."

# Estas líneas de código las requiere python para que
# se pueda empezar a trabajar con la aplicación
if __name__ == '__main__':
    #Corremos la aplicación en modo debug
    app.run(debug=True)
```

Y hacemos correr la aplicación. Veremos una salida por la terminal similar a la siguiente:

```
/bin/python app.py
* Serving Flask app 'app' (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:5000 (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 362-237-419
```

Hemos puesto en marcha un servidor, que entre otras características tiene la particularidad de proporcionarnos una ruta de acceso a su raíz ubicado en 127.0.0.1:5000

```
Running on http://127.0.0.1:5000/
```

Es decir, se encuentra en la dirección IP local 127.0.0.1 y en el puerto 5000. Por lo tanto, para acceder a nuestra aplicación debemos escribir en la barra de direcciones del navegador **127.0.0.1:5000**, presionar Enter y podremos ver el código devuelto en el **return**, ya que está siendo renderizado por Flask.

En pocas palabras, lo que está ocurriendo es que el navegador realiza una solicitud al servidor local de nuestra PC, con la dirección "/" , y es "atendida" por el **@app.route("/")** que posee nuestra aplicación. Si todo ha salido bien, en este punto deberíamos ver el mensaje **"Hola Codo a Codo Todo ha salido bien"** en el navegador.

Por supuesto, esta forma de enviar el código HTML al navegador mediante el return de la función es no solo engorrosa, sino tambien poco adecuada. Existe una forma mucho más simple y práctica, como veremos a continuación.

Guardamos el archivo **app_test.py** con el nombre **app_test2.py** y hacemos la siguiente modificación en el return:

```
@app.route('/')
def index():
    # Devolvemos código HTML para ser renderizado
    return render_template('empleados/index.html')
```

Y luego creamos, dentro de la subcarpeta **"empleados"**, un archivo llamado **index.html** con el siguiente contenido:

```
<h1>Hola Codo a Codo!</h1>
Todo a salido bien.
```

Si todo ha salido bien, deberíamos ver el mensaje **"Hola Codo a Codo Todo ha salido bien"** en el navegador. Esta vez, lo que se ha renderizado es el contenido del archivo **index.html**, que por supuesto puede contener cualquiera de los elementos que hemos visto en la etapa del frontend (CSS, Javascript, etc).

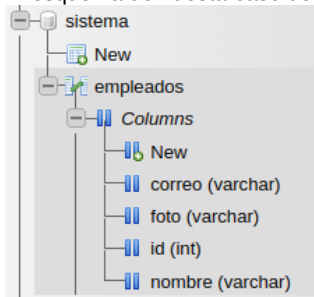
6 - Creando la base de datos:

Utilizamos **myphpadmin** y creamos una base de datos que se llame "**sistema**" con una tabla llamada "**empleados**". Esta tabla debe tener los siguientes campos:

	#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra
<input type="checkbox"/>	1	id	int(10)			No	None		AUTO_INCREMENT
<input type="checkbox"/>	2	nombre	varchar(255)	utf8mb4_general_ci		No	None		
<input type="checkbox"/>	3	correo	varchar(255)	utf8mb4_general_ci		No	None		
<input type="checkbox"/>	4	foto	varchar(255)	utf8mb4_general_ci		No	None		

En caso de tener alguna dificultad a la hora de crear esta tabla, podemos ver los videos de las clases sincrónicas referidas a SQL que se dictaron antes.

El esquema de nuestra base de datos debe ser similar al siguiente:



7 - Conexión a Base de datos

Necesitamos poder realizar operaciones sobre la base de datos desde nuestra aplicación escrita en Python. Para ello, es indispensable establecer un vínculo entre la base de datos en sí, y nuestro código Python. Esto lo hacemos de la siguiente manera:

app.py:

```
#-----
# Importamos el framework Flask
from flask import Flask

# Importamos la función que nos permit el render de los templates
from flask import render_template

# Importamos el módulo que permite conectarnos a la BS
from flaskext.mysql import MySQL
#-----

# Creamos la aplicación
app = Flask(__name__)

#-----
# Creamos la conexión con la base de datos:
mysql = MySQL()
# Creamos la referencia al host, para que se conecte a la base
# de datos MYSQL utilizamos el host localhost
app.config['MYSQL_DATABASE_HOST']='localhost'
# Indicamos el usuario, por defecto es user
app.config['MYSQL_DATABASE_USER']='root'
# Sin contraseña, se puede omitir
app.config['MYSQL_DATABASE_PASSWORD']=''
# Nombre de nuestra BD
app.config['MYSQL_DATABASE_DB']='sistema'
# Creamos la conexión con los datos
mysql.init_app(app)
#-----

# Proporcionamos la ruta a la raíz del sitio
@app.route('/')
def index():
    # Devolvemos código HTML para ser renderizado
    return render_template('empleados/index.html')
```

```
#-----
# Estas líneas de código las requiere python para que
# se pueda empezar a trabajar con la aplicación
if __name__ == '__main__':
    #Corremos la aplicación en modo debug
    app.run(debug=True)
#-----
```

El código está lo suficientemente comentado como para comprender que acción se realiza en cada línea. En cada aplicación que desarrollemos deberemos, por supuesto, modificar algunos parámetros para adaptar el código a las características específicas de la base de datos (nombre, usuario, etc.)

A partir de ahora, vamos a ir agregando código en nuestra **app.py**, de manera que paso a paso veamos cómo se implementa cada una de las funcionalidades requeridas por el CRUD.

Comenzaremos haciendo una prueba. Dentro de **def index()**: generaremos una instrucción sql, junto con todo lo necesario para hacer la conexión, y comprobar si todo funciona correctamente.

Vamos a necesitar la sintaxis de sql para agregar un registro en la base de datos. Si no recordamos como hacerlo, podemos ir a **phpmyadmin**, escribirla, probarla, y cuando la tengamos, la copiamos y pegamos en nuestro código. Nos va quedar algo así, dependiendo del contenido que pongamos a cada campo:

```
INSERT INTO `sistema`.`empleados` (`id`, `nombre`, `correo`, `foto`)
VALUES (NULL, 'Ana Maria', 'ana.maria@gmail.com', 'anamaria.jpg');
```

Y con los cambios que haremos, **def index()**: se verá así:

```
@app.route('/')
def index():
    # Creamos una variable que va a contener la consulta sql:
    sql = "INSERT INTO `sistema`.`empleados` (`id`, `nombre`, `correo`, `foto`) \
VALUES (NULL, 'Ana Maria', 'ana.maria@gmail.com', 'anamaria.jpg');"

    # Conectamos a la conexión mysql.init_app(app)
    conn = mysql.connect()

    # Almacenaremos lo que devuelva la consulta
    cursor = conn.cursor()

    # Ejecutamos la sentencia SQL
    cursor.execute(sql)

    # "Commitamos" (Cerramos la conexión)
    conn.commit()

    # Devolvemos código HTML para ser renderizado
    return render_template('empleados/index.html')
```

Cuando ejecutemos este código, si todo está bien, se agregará un registro a la tabla con los datos que hemos definido en la variable sql cada vez que desde nuestro navegador accedamos a la dirección <http://127.0.0.1:5000>:

☐ Show all
 | Number of rows: 25
 | Filter rows:

Extra options

	id	nombre	correo	foto
<input type="checkbox"/>	1	Ana Maria	ana.maria@gmail.com	anamaria.jpg

☐ Check all
 | With selected:
 ☐ Edit
 ☐ Copy
 ☐ Delete
 ☐ Export

☐ Show all
 | Number of rows: 25
 | Filter rows:

Si accedemos varias veces el código, agregaremos un nuevo registro (con los mismos datos!) cada vez:

☐ Show all

Number of rows: 25

Filter rows:

Sort

Extra options

	id	nombre	correo	foto
<input type="checkbox"/> Edit Copy Delete	1	Ana Maria	ana.maria@gmail.com	anamaria.jpg
<input type="checkbox"/> Edit Copy Delete	2	Ana Maria	ana.maria@gmail.com	anamaria.jpg
<input type="checkbox"/> Edit Copy Delete	3	Ana Maria	ana.maria@gmail.com	anamaria.jpg
<input type="checkbox"/> Edit Copy Delete	4	Ana Maria	ana.maria@gmail.com	anamaria.jpg
<input type="checkbox"/> Edit Copy Delete	5	Ana Maria	ana.maria@gmail.com	anamaria.jpg

☐ Check all

With selected: Edit Copy Delete Export

☐ Show all

Number of rows: 25

Filter rows:

Sort

8 - Estructuras HTML create y edit

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

Desde VSC crearemos dentro de la subcarpeta **templates/empleados** los siguientes archivos:

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

create.html:

```
<form method="post" action="" enctype="multipart/form-data">
  Nombre:
  <input type="text" name="txtNombre" id="txtNombre"> <br>
  Correo:
  <input type="text" name="txtCorreo" id="txtCorreo"><br>
  Foto:
  <input type="file" name="txtFoto" id="txtFoto"><br><br>
  <input type="submit" value="Agregar">
</form>
```

¿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):

(En **app_v5.app**)

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

Para ver este formulario necesitamos, desde el navegador, acceder a <http://127.0.0.1:5000/create> , con nuestro **app.py** ejecutandose. Veremos algo como esto:

Nombre:

Correo:

Foto: Ninguno archivo selec.

Un típico formulario sin CSS. Es el formulario de creación que nos va a servir para solicitar información al usuario y enviarla a la BD.

OJO: Al **app.route("/store")** hay que ponerle en el return un **redirect('/')** , y agregar en los import el

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

8 - Recepción de valores de entrada

Ahora necesitamos que nuestra aplicación Python reciba y procese la información que llega desde el formulario **create.html** . Tenemos que recibir los datos, y luego insertarlos en la base de datos. Para ello debemos hacer algunos cambios en **app.py**.

En primer lugar vamos a agregar en esta línea:

```
from flask import render_template
```

el módulo request. Lo podemos hacer simplemente sumandolé al final de la línea, separado por una coma:

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

Esto es necesario para que **app.py** sea capaz de procesar el envío de información que nos van a hacer los formularios HTML.

Para que el formulario de **create.html** pueda enviar la información, necesitamos modificar dentro de la etiqueta form el `action=""` para incluir ahí el destino de los datos cuando se presione el botón enviar. Modificamos la línea

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

para que nos quede así:

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

Ahora necesitamos agregar, en **app.py** una función que sea capaz de recibir los datos enviados por el formulario, insertarlos en la base de datos y devolver el control al archivo **index.html**.

Vamos a agregar la siguiente función debajo de la última que agregamos (`@app.route('/create')`):

```
@app.route('/store', methods=['POST'])
def storage():
    # Recibimos los valores del formulario y los pasamos a variables locales:
    _nombre = request.form['txtNombre']
    _correo = request.form['txtCorreo']
    _foto = request.files['txtFoto']

    # Y armamos una tupla con esos valores:
    datos = (_nombre, _correo, _foto.filename)

    # Armamos la sentencia SQL que va a almacenar estos datos en la DB:
    sql = "INSERT INTO `sistema`.`empleados` \
          (`id`, `nombre`, `correo`, `foto`) \
          VALUES (NULL, %s, %s, %s);"

    conn = mysql.connect()      # Nos conectamos a la base de datos
    cursor = conn.cursor()     # En cursor vamos a realizar las operaciones
    cursor.execute(sql, datos) # Ejecutamos la sentencia SQL en el cursor
    conn.commit()              # Hacemos el commit
    return render_template('empleados/index.html') # Volvemos a index.html
```

Utilizamos **filename** para la foto porque recibimos un objeto que tiene distintos tipos de datos, como el nombre, el contenido, etc, y nosotros necesitamos solamente el nombre.

Si ahora volvemos a la terminal, corremos nuevamente **app.py** y actualizamos la página o ingresamos en <http://127.0.0.1:5000/create> y agregamos datos, cuando pulsemos el botón enviar, deberíamos volver a ver en el navegador el contenido de **index.html**. Y los datos del formulario deberían estar ingresados en la base de datos.

Podemos verificarlo haciendo un *browse* de la tabla empleados con **phpmyadmin**.

9 - Guardar imagen en carpeta uploads

En este punto, nuestro CRUD ya es capaz de guardar datos en la base de datos. Pero hemos implementado el código que se encargue de copiar la imagen que el usuario ha seleccionado en el formulario a una carpeta. Necesitamos desarrollar esa funcionalidad para que cuando hagamos una consulta de los datos de ese empleado, podamos acceder y mostrar al usuario la fotografía correspondiente.

Ya creamos una carpeta llamada **uploads** destinada a almacenar las imágenes que vayamos subiendo.

Una cuestión a tener en cuenta, antes de comenzar a trabajar en pos de este objetivo, es tener en cuenta la posibilidad de que dos imágenes que vayamos a guardar en esa carpeta tengan el mismo nombre. Si eso ocurriese, la segunda en ser enviada reemplazaría a la primera. Es decir, un archivo sobrescribiría al otro.

Eso se puede evitar asignando un nombre único a cada imagen. Una forma simple de lograr esto es utilizar información relativa a la fecha y hora que tiene el sistema en el momento de hacer el upload. Si bien esto no va a garantizar al 100% que no ocurra una sobrescritura, lo va a minimizar enormemente.

Para implementar esta opción, lo primero que necesitamos es agregar a nuestro **app.py** la siguiente línea:

```
# Importamos las funciones relativas a fecha y hora
from datetime import datetime
```

Esta librería proporciona acceso a funciones que nos van a ayudar a resolver el problema.

Antes de almacenar la foto, le vamos a concatenar a la cadena que contiene su nombre la fecha y hora que posee la computadora en el momento de hacer el upload. Cómo es fácil ver, la posibilidad que dos imágenes con el mismo nombre se intenten cargar en el mismo segundo exacto es muy baja.

Eso lo vamos a hacer con el siguiente código:

```
# Guardamos en now los datos de fecha y hora
now = datetime.now()

# Y en tiempo almacenamos una cadena con esos datos
tiempo = now.strftime("%Y%H%M%S")

#Si el nombre de la foto ha sido proporcionado en el form...
if _foto.filename!='':
    #...le cambiamos el nombre.
    nuevoNombreFoto=tiempo+_foto.filename
```

Y debemos cambiar esta línea:

```
# Y armamos una tupla con esos valores:
datos = (_nombre,_correo,_foto.filename)
```

Por esta:

```
# Y armamos una tupla con esos valores:
datos = (_nombre,_correo, nuevoNombreFoto)
```

ya que ahora el nombre de la foto no es el que se ha proporcionado desde el formulario, sino que le hemos agregado los datos referidos a la fecha y hora. En resumen, el código luego del cambio la función **def storage()**: debe quedarnos así:

```
@app.route('/store', methods=['POST'])
def storage():
    # Recibimos los valores del formulario y los pasamos a variables locales:
    _nombre = request.form['txtNombre']
    _correo = request.form['txtCorreo']
    _foto = request.files['txtFoto']

    # Guardamos en now los datos de fecha y hora
    now = datetime.now()

    # Y en tiempo almacenamos una cadena con esos datos
    tiempo = now.strftime("%Y%H%M%S")

    #Si el nombre de la foto ha sido proporcionado en el form...
    if _foto.filename!='':
        #...le cambiamos el nombre.
        nuevoNombreFoto=tiempo+_foto.filename
        # Guardamos la foto en la carpeta uploads.
        _foto.save("uploads/"+nuevoNombreFoto)

    # Y armamos una tupla con esos valores:
    datos = (_nombre,_correo, nuevoNombreFoto)

    # Armamos la sentencia SQL que va a almacenar estos datos en la DB:
    sql = "INSERT INTO `sistema`.`empleados` \
        (`id`, `nombre`, `correo`, `foto`) \
        VALUES (NULL, %s, %s, %s);"

    conn = mysql.connect()      # Nos conectamos a la base de datos
    cursor = conn.cursor()      # En cursor vamos a realizar las operaciones
    cursor.execute(sql, datos)  # Ejecutamos la sentencia SQL en el cursor
    conn.commit()               # Hacemos el commit
    return render_template('empleados/index.html') # Volvemos a index.html
```

Hecho esto, podemos hacer correr la app, acceder desde nuestro navegador a <http://127.0.0.1:5000/create> , subir los datos de un nuevo empleado y comprobar que se haya subido la imagen en la carpeta **uploads**. Su nombre debe tener los datos referidos a la fecha y hora.

10 - Consultando datos de la 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:

app_v4.app:

```
# Proporcionamos la ruta a la raíz del sitio
@app.route('/')
def index():
    #-----
    # Hacemos una modificación para mostrar datos de la tabla
    # Empleados en la terminal...
    #-----
```



```

# Creamos una variable que va a contener la consulta sql:
sql = "SELECT * FROM `sistema`.`empleados`;"

# Nos conectamos a la base de datos
conn = mysql.connect()

# Sobre el cursor vamos a realizar las operaciones
cursor = conn.cursor()

# Ejecutamos la sentencia SQL sobre el cursor
cursor.execute(sql)

# Copiamos el contenido del cursor a una variable
db_empleados = cursor.fetchall()

# y mostramos las tuplas por la terminal
print("-"*60)
for empleado in db_empleados:
    print(empleado)
print("-"*60)

# "Comitemos" (Cerramos la conexión)
conn.commit()

# Devolvemos código HTML para ser renderizado
return render_template('empleados/index.html')

```

Al ejecutar en <http://127.0.0.1:5000/> 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 refectuar el envío de datos a través de **render_template**, para eso modificamos en **def index()** el return para que quede de la siguiente manera:

```

# Devolvemos código HTML para ser renderizado
return render_template('empleados/index.html', empleados = db_empleados)

```

Le enviamos el archivo **index.html** al navegador través de **render_template**. Vamos a enviarle una variable (empleados) con los valores de los datos de la tabla empleados (db_empleados)

En el **index.html** agregaremos la tabla en el body utilizando Bootstrap y la extensión que descargamos al principio:

- **Bootstrap v4 Snippets**: nos permitirá escribir HTML para incluir elementos de Bootstrap.

Para esto escribimos **b-table-header** en VSC, y el autocompletado nos va a crear una es estructura básica:

```

<table class="table table-light">
  <thead class="thead-light">
    <tr>
      <th>#</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td></td>
    </tr>
  </tbody>
</table>

```

que 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	anamaria.jpg	Ana Maria	ana.maria@gmail.com	Editar Borrar
2	anamaria.jpg	Ana Maria	ana.maria@gmail.com	Editar Borrar
3	anamaria.jpg	Ana Maria	ana.maria@gmail.com	Editar Borrar
4	anamaria.jpg	Ana Maria	ana.maria@gmail.com	Editar Borrar
5	anamaria.jpg	Ana Maria	ana.maria@gmail.com	Editar Borrar
6	anamaria.jpg	Ana Maria	ana.maria@gmail.com	Editar Borrar
7	anamaria.jpg	Ana Maria	ana.maria@gmail.com	Editar Borrar
8	anamaria.jpg	Ana Maria	ana.maria@gmail.com	Editar Borrar
9	anamaria.jpg	Ana Maria	ana.maria@gmail.com	Editar Borrar
10	anamaria.jpg	Ana Maria	ana.maria@gmail.com	Editar Borrar
11	anamaria.jpg	Ana Maria	ana.maria@gmail.com	Editar Borrar

Si agregamos un header con el link a bootstrap:

```
<html lang="en">
<head>
  <title>CRUD - Listado de Empleados</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet">
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/bootstrap.bundle.min.js"></script>
</head>
```

Tenemos algo mucho mas bonito:

#	Foto	Nombre	Correo	Acciones
1	anamaria.jpg	Ana Maria	ana.maria@gmail.com	Editar Borrar
2	anamaria.jpg	Ana Maria	ana.maria@gmail.com	Editar Borrar
3	anamaria.jpg	Ana Maria	ana.maria@gmail.com	Editar Borrar
4	anamaria.jpg	Ana Maria	ana.maria@gmail.com	Editar Borrar
5	anamaria.jpg	Ana Maria	ana.maria@gmail.com	Editar Borrar
6	anamaria.jpg	Ana Maria	ana.maria@gmail.com	Editar Borrar
7	anamaria.jpg	Ana Maria	ana.maria@gmail.com	Editar Borrar
8	anamaria.jpg	Ana Maria	ana.maria@gmail.com	Editar Borrar
9	anamaria.jpg	Ana Maria	ana.maria@gmail.com	Editar Borrar
10	anamaria.jpg	Ana Maria	ana.maria@gmail.com	Editar Borrar
11	anamaria.jpg	Ana Maria	ana.maria@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. Nos debe quedar así:

```
# Importamos la función que nos permit el render de los templates,
# recibir datos del form, redireccionar, etc.
from flask import render_template, request, redirect
```

Y tenemos que crear en el mismo archivo una función que "atienda" el borrado de los registros, que se invoca cuando desde el botón llamamos a **destroy**, pasando el nro de **id** del registro a borrar. La escribimos encima del **@app.route('/create')**:

```
#-----
# Función para eliminar un registro
@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 link **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 la Función para eliminar un registro las instrucciones que nos permitan modificar un registro. Este sería el código:

```
#-----
# Función para editar un registro
@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)
```

Creamos el archivo **edit.html**:

- Llama a UPDATE
- Muestra los valores recuperados de la base de datos.

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

    ID:
    <input type="text" value="{{ empleados[0][0] }}" name="txtID" id="txtID"><br>

    Nombre:
    <input type="text" value="{{ empleados[0][1] }}" name="txtNombre" id="txtNombre"><br>

    Correo:
    <input type="text" value="{{ empleados[0][2] }}" name="txtCorreo" id="txtCorreo"><br>

    Foto:{{ empleados[0][3] }}
    <input type="file" name="txtFoto" id="txtFoto"><br><br>

    <input type="submit" value="Guardar">

</form>
```

Podemos probar el resultado y veremos que si hacemos clic en **Editar** de un registro estaremos listos para modificarlo:

ID:	4	Nombre:	Ana Maria
Correo:	ana.maria@gmail.com		
Foto:	anamaria.jpg <input type="button" value="Seleccionar archivo"/> Ninguno ...hivo selec.		
<input type="button" value="Agregar"/>			

14 - Guardando la actualización:

Además vamos a tener que crear una función que se ejecute cuando enviamos el formulario recién creado (**UPDATE**), 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 **update** para recibir los datos al igual que lo hicimos con **storage**:

(Está en **app_v7.py**)

```
#-----
# Función para actualizar los datos de un registro
@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('/')
```

15 - Modificando la foto

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

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. (Todo esto está realizado en **app_v8.py**)

```
# Importamos paquetes de interfaz con el sistema operativo.
import os
```

Además, antes de **@app.route()** vamos a crear una referencia a la carpeta utilizando algunas propiedades del módulo que acabamos de importar:

```
# Guardamos la ruta de la carpeta "uploads" en nuestra app
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 con fecha y hora como lo hicimos cuando lo creábamos.

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

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). Y 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.

Código completo de **update** en **app_v8.py**:

```
#-----
# Función para actualizar los datos de un registro
@app.route('/update', methods=['POST'])
def update():
    # Recibimos los valores del formulario y los pasamos a variables locales:
    _nombre = request.form['txtNombre']
    _correo = request.form['txtCorreo']
    _foto = request.files['txtFoto']
    id = request.form['txtID']

    # Armamos la sentencia SQL que va a actualizar los datos en la DB:
    sql = "UPDATE `sistema`.`empleados` SET `nombre`=%s, `correo`=%s WHERE id=%s;"
    # Y la tupa correspondiente
    datos = (_nombre,_correo,id)

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

    # Guardamos en now los datos de fecha y hora
    now = datetime.now()

    # Y en tiempo almacenamos una cadena con esos datos
    tiempo= now.strftime("%Y%H%M%S")

    #Si el nombre de la foto ha sido proporcionado en el form...
    if _foto.filename != '':
        # Creamos el nombre de la foto y la guardamos.
        nuevoNombreFoto = tiempo + _foto.filename
        _foto.save("uploads/" + nuevoNombreFoto)

        # Buscamos el registro y buscamos el nombre de la foto vieja:
        cursor.execute("SELECT foto FROM `sistema`.`empleados` WHERE id=%s", id)
        fila= cursor.fetchall()
```

```
# Y la borramos de la carpeta:
os.remove(os.path.join(app.config['CARPETA'], fila[0][0]))

# Finalmente, actualizamos la DB con el nuevo nombre del archivo:
cursor.execute("UPDATE `sistema`.`empleados` SET foto=%s WHERE id=%s", (nuevoNombreFoto, id))
conn.commit()

cursor.execute(sql, datos)
conn.commit()
return redirect('/')
```

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 la Función para eliminar un registro las instrucciones que nos permitan modificar un registro. Este sería el código:

```
#-----
# Función para editar un registro
@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)
```

Creamos el archivo **edit.html**:

- Llama a UPDATE
- Muestra los valores recuperados de la base de datos.

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

    ID:
    <input type="text" value="{{ empleados[0][0] }}" name="txtID" id="txtID"><br>

    Nombre:
    <input type="text" value="{{ empleados[0][1] }}" name="txtNombre" id="txtNombre"><br>

    Correo:
    <input type="text" value="{{ empleados[0][2] }}" name="txtCorreo" id="txtCorreo"><br>

    Foto:{{ empleados[0][3] }}
    <input type="file" name="txtFoto" id="txtFoto"><br><br>

    <input type="submit" value="Guardar">

</form>
```

Podemos probar el resultado y veremos que si hacemos clic en **Editar** de un registro estaremos listos para modificarlo:

ID: Nombre:

Correo:

Foto: anamaria.jpg Ninguno ...hivo selec.

14 - Guardando la actualización:

Además vamos a tener que crear una función que se ejecute cuando enviamos el formulario recién creado (**UPDATE**), 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 **update** para recibir los datos al igual que lo hicimos con **storage**:

(Está en **app_v7.py**)

```
#-----
# Función para actualizar los datos de un registro
@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('/')

```

15 - Modificando la foto

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

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. (Todo esto está realizado en **app_v8.py**)

```
# Importamos paquetes de interfaz con el sistema operativo.
import os

```

Además, antes de **@app.route()** vamos a crear una referencia a la carpeta utilizando algunas propiedades del módulo que acabamos de importar:

```
# Guardamos la ruta de la carpeta "uploads" en nuestra app
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 con fecha y hora como lo hicimos cuando lo creábamos.

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

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). Y 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.

Código completo de **update** en **app_v8.py**:

```
#-----
# Función para actualizar los datos de un registro
@app.route('/update', methods=['POST'])
def update():
    # Recibimos los valores del formulario y los pasamos a variables locales:
    _nombre = request.form['txtNombre']
    _correo = request.form['txtCorreo']
    _foto = request.files['txtFoto']
    id = request.form['txtID']

    # Armamos la sentencia SQL que va a actualizar los datos en la DB:
    sql = "UPDATE `sistema`.`empleados` SET `nombre`=%s, `correo`=%s WHERE id=%s;"
    # Y la tupa correspondiente
    datos = (_nombre,_correo,id)

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

    # Guardamos en now los datos de fecha y hora
    now = datetime.now()

    # Y en tiempo almacenamos una cadena con esos datos
    tiempo= now.strftime("%Y%H%M%S")

    #Si el nombre de la foto ha sido proporcionado en el form...
    if _foto.filename != '':
        # Creamos el nombre de la foto y la guardamos.
        nuevoNombreFoto = tiempo + _foto.filename
        _foto.save("uploads/" + nuevoNombreFoto)

        # Buscamos el registro y buscamos el nombre de la foto vieja:
        cursor.execute("SELECT foto FROM `sistema`.`empleados` WHERE id=%s", id)
        fila= cursor.fetchall()

```

```
# Y la borramos de la carpeta:
os.remove(os.path.join(app.config['CARPETA'], fila[0][0]))

# Finalmente, actualizamos la DB con el nuevo nombre del archivo:
cursor.execute("UPDATE `sistema`.`empleados` SET foto=%s WHERE id=%s;", (nuevoNombreFoto, id))
conn.commit()

cursor.execute(sql, datos)
conn.commit()
return redirect('/')
```

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.

header.html:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css">
  <title>CRUD Codo a Codo</title>
</head>
<body>
  <div class="container">
    Cabecera del sitio
```

footer.html:

```
<hr>
  Pie de página
</div>
</body>
</html>
```

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 'empleados/header.html' %}
```

En la última línea agregaremos:

```
{% include 'empleados/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_v9.py**.

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**. 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 modificar en las primeras líneas de **app_v9.py**

```
# Importamos la función que nos permit el render de los templates,
# recibir datos del form, redireccionar, etc.
from flask import render_template, request, redirect, send_from_directory
```

Además, dentro de **app_v9.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()**):

```
#-----
# Generamos 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.
@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.

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:

```
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]**. (Esto ya lo tenemos de **app_v9.py** en adelante)

Así deberíamos ver en este momento :


Cabecera del sitio

ID:

Nombre:

Correo:

Foto:



Pie de página

20 - Agregar menú de navegación

A partir de aquí, usamos el código compartido en la carpeta "extras" y lo copiamos a los archivos correspondientes del proyecto.

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:

```
<body>
<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>
```

Todo esto está dentro del archivo header2.html

21 - Aplicando estilos a botones

Todo esto está dentro del archivo index2.html

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.

```
<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="/destroy/{{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 preguntando si deseamos borrar al empleado.

22 - Ajustando formulario create

Todos estos cambios están en create2.html

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.

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**.

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

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.

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.

create2.html:

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

<form method="post" action="/store" enctype="multipart/form-data">
  <div class="card">
    <div class="card-header">
      Ingresar empleados
    </div>
    <div class="card-body">
      <h5 class="card-title">Datos de empleados</h5>
      <p class="card-text">

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

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

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

        <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>

</p>
</div>
</div>
</form>

{% include 'footer.html' %}

```

23 - Ajustando formulario edit

Todos estos cambios están en **edit2.html**

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**.

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

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

create2.html:

```

{% include 'header.html' %}

<form method="post" action="/update" enctype="multipart/form-data">
    <div class="card">
        <div class="card-header">
            Editar Empleado
        </div>
        <div class="card-body">
            <h5 class="card-title">Datos del empleado</h5>
            <p class="card-text">

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

                <div class="form-group">
                    <label for="txtNombre">Nombre:</label>
                    <input id="txtNombre" value="{{ empleado[0][1] }}" class="form-control" type="text" name="txtNombre">
                </div>

                <div class="form-group">
                    <label for="txtCorreo">Correo:</label>
                    <input id="txtCorreo" value="{{ empleado[0][2] }}" class="form-control" type="text" name="txtCorreo">
                </div>

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

                <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>

            </p>
        </div>
    </div>
</form>

{% include 'footer.html' %}

```

24 - Manejo de mensajes de validación

Esto está dentro de **app.py**

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

```

# Importamos la función que nos permit el render de los templates,
# recibir datos del form, redireccionar, etc.
from flask import render_template, request, redirect
from flask import send_from_directory, url_for, flash

```

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()**

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

```
{% 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 flask, 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.