DESARROLLO WEB

CON

PYTHON Y FLASK

Vanessa Lorena Valverde González
Byron Ernesto Vaca Barahona
Gustavo Xavier Hidalgo Solórzano

ISBN: 978-9942-679-53-6

CIDE
EDITORIAL

Desarrollo web con Python y Flask

Desarrollo web con Python y Flask

Autores:

Vanessa Lorena Valverde González

Byron Ernesto Vaca Barahona

Gustavo Xavier Hidalgo Solórzano

Centro de Investigación y Desarrollo Ecuador
Tel.: + (593) 04 2037524
http.:/www.cidecuador.org

ISBN: 978-9942-679-53-6

http://doi.org/10.33996/cide.ecuador.DW2679536

Vanessa Lorena Valverde González
Byron Ernesto Vaca Barahona
Gustavo Xavier Hidalgo Solórzano
Escuela Superior Politécnica de Chimborazo (ESPOCH)

Dirección editorial: Lic. Pedro Misacc Naranjo, Msc.
Coordinación técnica: Lic. María J. Delgado
Diseño gráfico: Lic. Danissa Colmenares
Diagramación: Lic. Alba Gil
Fecha de publicación: mayo, 2025

Guayaquil, Ecuador

La presente obra fue evaluada por pares académicos experimentados en el área.

Catalogación en la Fuente

Desarrollo web con Python y Flask / Vanessa Lorena Valverde González, Byron Ernesto Vaca Barahona, Gustavo Xavier Hidalgo Solórzano - Ecuador: Editorial CIDE, 2025.

286 p.: incluye tablas, figuras; 17,6 x 25 cm.

ISBN: 978-9942-679-53-6

1. Python 2. Flask. 3. Lenguaje de programación

Contenido

Capítulo 1
Herramientas para el desarrollo de software

Capítulo 2
Establecimiento del contexto

Capítulo 3
Establecimiento del entorno de trabajo

Capítulo 4
Desarrollo del módulo

Prólogo

A lo largo de mi experiencia como desarrollador de software y como agente de seguridad de tecnologías de la información, he logrado adquirir un amplio conocimiento en el desarrollo e implementación de proyectos de software; la variedad de lenguajes de programación ha sido una tarea de aprendizaje constante ya que la evolución es creciente y lo que ayer se usaba para implementar un sistema informático hoy ya es obsoleto.

El libro *Desarrollo web con Python y Flask*, expone una propuesta didáctica, organizada y sencilla del desarrollo del módulo de gestión de seguridad de usuarios, roles y permisos, en un lenguaje de programación versátil y fácil de entender como es *Python*.

El libro hace un recorrido muy metódico de cómo implementar un módulo de seguridad, partiendo de conceptos básicos de *Python*, siguiendo con herramientas framework *flask* para el desarrollo web, adentrándose a los patrones de diseño, patrones arquitectónicos y llegando a la implementación del código fuente del módulo siguiendo

un régimen estricto del patrón arquitectónico *MVC* (Modelo-Vista-Controlador).

Es importante como los autores exponen de manera clara y didáctica los conceptos, pero lo más interesante es como esos conceptos los transforman en ejercicios prácticos escritos en el lenguaje *Python*, enriqueciendo aún más el contenido del libro y aportando de manera positiva al aprendizaje del lector.

El desarrollo de la estructura de software, aplicando conceptos de arquitectura de software, entornos virtuales y bases de datos, es dinámica y clara. Esto permite al lector seguir una estructura cuidadosamente planificada con el objetivo de facilitar el aprendizaje. Así, mediante la aplicación de los conceptos y ejemplos presentados, podrá crear sus propios sistemas informáticos o programas sin dificultad.

El establecer un caso de estudio para ser implementado mediante la herramienta de desarrollo *Python* y el framewok para la web *flask,* permite al lector explorar un mundo de ideas de proyectos donde los conceptos tratados pueden ser implementados.

Por último, la forma en que los autores han explicado las líneas de código implementadas en el módulo refleja su compromiso con la enseñanza. Su enfoque busca que el lector comprenda las diferentes

maneras de resolver un problema planteado, a partir de un caso de estudio específico.

Ing. Juan Carlos Díaz Ordoñez Mgs.

Analista de seguridad de tecnologías de la información 3

UNIDAD DE GESTIÓN DE SEGURIDAD INFORMÁTICA

“Los límites de mi lenguaje, son los límites de mi mundo”.

Ludwing Wittgenstein

Introducción

Actualmente, en el mercado existen una gran cantidad de herramientas que permiten a programadores, arquitectos e ingenieros a diseñar, desarrollar e implementar proyectos de software de manera sencilla y rápida.

En la amplia gama de herramienta para el desarrollo de aplicaciones web existe una variedad de opciones que por sus características son utilizadas por los programadores, desde el ámbito de sus librerías hasta su versatilidad en la escritura del código, entre las más conocidas tenemos: *java, NodeJs, Python, Php, C#*, entre otros.

En un mundo globalizado, las empresas e instituciones, tanto públicas como privadas, compiten constantemente para mejorar la calidad y acelerar la producción de sus productos. Para lograrlo, recurren a la informática como el medio más eficiente y eficaz para resolver los problemas generados en la constante ejecución de sus procesos. En esta lucha por la optimización, mantenerse a la vanguardia tecnológica es crucial, ya que quienes no innovan corren el riesgo de desaparecer. Por ello, las empresas se ven en la obligación de

optimizar sus procesos y desarrollar soluciones informáticas que mejoren sus productos.

Como alternativa, se presenta el libro "*Desarrollo web con Python y Flask*", cuyo objetivo es guiar al lector en un camino fácil para el desarrollo de aplicaciones web utilizando el lenguaje de programación Python y el framework Flask.

A través de esta guía, el lector podrá descubrir aspectos del desarrollo de software que quizás desconocía o no comprendía completamente. Por ello, se ha estructurado el libro en cuatro capítulos escritos intencionalmente para que el receptor avance progresivamente, desde los fundamentos hasta el dominio completo del tema.

El Capítulo I: Herramientas para el desarrollo de software, guía al lector en el aprendizaje del entorno de desarrollo de *Python*, abarcando su proceso de instalación, el manejo de variables, operadores, estructuras de control, estructuras de datos y otros conceptos fundamentales.

Luego, permite entender el funcionamiento del Framework *Flask* y sus componentes más importantes, además, ofrece una guía sencilla de instalación, gestión y operación del framework; por último,

hace un recorrido por el ámbito de los entornos virtuales, desde su instalación hasta su implementación.

El capítulo II: Establecimiento del contexto, induce al receptor en un contexto más amplio del tema, estableciendo un caso de estudio, explicando de una manera didáctica y sencilla conceptos como: patrones de diseño, patrones arquitectónicos y estilos arquitectónicos, además del diseño del caso de estudio por medio de **mockup's** desembocando en el diseño físico de la base de datos.

La siguiente fase es configurar el entorno de trabajo. Por ello, el **Capítulo III: Establecimiento del entorno de trabajo,** guía al lector en la instalación del entorno de desarrollo *Visual Studio Code*. Además, explica de manera clara los conceptos clave sobre las librerías de *Python* y *Flask* que se utilizarán en la implementación del caso de estudio, junto con la aplicación del patrón arquitectónico ***MVC*** y la configuración del entorno virtual.

Por último, el **capítulo IV: Desarrollo del módulo**, aborda los detalles de la implementación del patrón arquitectónico en la codificación del módulo de seguridad y manejo de usuarios, roles y permisos.

Cada capítulo fue escrito de manera que aporten al lector para un mejor entendimiento de desarrollo de una aplicación web en *Python* y *Flask*.

Guía de uso

Este libro está escrito para que el lector guie su aprendizaje mientras avanza en la lectura del mismo, a través de cada capítulo existen un conjunto de conceptos nuevos y conceptos importantes, que llevarán al lector por el camino del conocimiento en la materia aquí descrita.

Para hacer más amigable la lectura y resaltar los conceptos clave, los autores han utilizado una grafología sencilla e intuitiva, presentando los elementos importantes de la siguiente manera:

Texto normal:

Es la forma de escritura que se utilizará en todo el libro.

Texto cursivo:

Se utiliza para hacer referencias de conceptos, citas técnicas o palabas de autores.

Texto negritas

Se utiliza para resaltar palabras o código importante, que los autores requieran que el lector identifique.

Cada vez que el lector encuentre una imagen de este tipo, el libro le esta indicando que es un concepto importante y va a ser aplicado más adelante mientras avance en su lectura.

Esta imagen indica al lector que la información contenida en el recuadro es un concepto nuevo, que quizás no lo conoce o que se requiera reforzarlo para proseguir con la lectura.

Este recuadro es utilizado para incluir referencias de código ejecutable, que ayudará al lector a interpretar la idea de un concepto y su implementación.

```
Este recuadro es utilizado para incluir referencias de código ejecutable.
```

El código fuente de la aplicación web completo se encontrará alojado en la siguiente dirección: https://github.com/ghidalgoxs/ModSeguridad

CAPÍTULO 1

Herramientas para el desarrollo de software

1

Capítulo 1

Herramientas para el desarrollo de software

El desarrollo de software para la web está avanzando a pasos agigantados**.** Hoy en día existe una variedad de herramientas que facilitan la creación de sistemas complejos, diseñados para la web y así apoyar a sus usuarios y/o clientes en sus requerimientos y tareas.

En el contexto de este libro, es fundamental explorar las herramientas que serán clave para el aprendizaje y dominio del *'Desarrollo web con Python y Flask'*.

1.1. Python

Creado por Guido Van Rossum, para ser el sucesor del lenguaje de programación ABC por los años 80 y 90.

Existen muchas definiciones de gran importancia y muy relevantes que han permitido que este lenguaje de programación

tenga un reconocimiento mundial. Entre las muchas definiciones se exponen las siguientes:

> *Llamas (2023) indica que Python es un lenguaje de programación de código abierto que posee un enfoque imperativo, orientado a objetos y de alto nivel (párr. 1).*
>
> *Por otro lado, Lozano (2023) indica que Python es un lenguaje de programación de alto nivel cuya máxima es la legibilidad del código (párr. 2).*
>
> *Ahora bien, Valverde (2023) menciona que Python es un lenguaje de programación multiparadigma (programación orientada a objetos, programación imperativa y programación funcional). Se lo puede aplicar en diferentes plataformas, es muy útil para el análisis de datos y se puede conectar con base de datos. (p. 173).*
>
> *De la misma manera Challenger et al. (2014) lo describe como un lenguaje de alto nivel ya que contiene implícitas algunas estructuras de datos como listas, diccionarios, conjuntos y tuplas, que permiten realizar algunas tareas complejas en pocas líneas de código y de manera legible (p. 3).*

En concordancia a los criterios antes mencionados, los actores coinciden en su mayoría en que es un lenguaje de alto nivel y fácil de escribir. Por lo tanto, se define como:

Python es: un lenguaje de programación de alto nivel, flexible, intuitivo y orientado a objetos.

1.1.1. Instalación de Python

La instalación de *Python* es un proceso muy fácil y rápido de llevarlo a cabo, lo primero que hay que tener en cuenta al conseguir el paquete de instalación es que sea el más actual y estable, para ello la comunidad ha dispuesto el enlace *https://www.python.org/* que es un entorno oficial para poder descargar e instalar el paquete sin restricciones.

- Instalación en Sistemas *Windows*: descargado el paquete hay que posicionarnos en el directorio de descarga y pulsar doble clic sobre este, el Wizard de instalación lo hace muy fácil para entornos *Windows* ya que solo necesita dar clic en el enlace "Install Now", sin olvidarse de activar la casilla "Add Python 3.X to PATH" y seguir las ventanas con el botón siguiente.

- Instalación en Sistemas *Linux*: es prácticamente automático sin necesidad de descargas anteriores, hay que digitar el siguiente comando desde consola:

  $ sudo yum install python3

Para comprobar que el paquete se ha instalado se requiere ingresar a una consola ya sea en cmd (para *Windows*) o Shell (para *Linux*) y digitar el siguiente comando.

> python (*Windows*)

$ python3 (*Linux*)

Luego,

```
> python
Python 3.10.7 (tags/v3.10.7:6cc6b13, Sep  5 2022,
14:08:36) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()"
for more information.
>>>
```

1.1.2. Variables, tipo de datos y estructuras de datos

Antes de hablar de variables primero debemos conocer que es un valor. Según Downey et al.(2002) indican que un valor es un elemento o pieza de información, dato que se requiere para hacer

cálculos, operaciones o instrucciones dentro de un determinado programa o sistema informático (p. 37).

El valor None: es usado cuando una variable no tiene valor, en un ejercicio de condicional se interpreta como falso y es devuelto en funciones que no devuelven valor.

Ejemplo.

El valor 23

El valor “Buenos días”

El valor 3,14

a) Variables

Basado en el argumento anterior, una variable vendría a ser “*el contenedor con un nombre que permite acceder al o los valores de los datos*”. *Python* es uno de los lenguajes de programación más eficiente al momento de manipular variables y no requiere declarar la variable de algún tipo de datos para utilizarla dentro de un programa.

La forma de usar las variables en *Python* es la siguiente:

[Nombre de la variable] = [Valor]

Ejemplo.

```
numero = 23
cadena = "Buenos días"
numero_real = 123,1
```

En los ejemplos anteriores, se han asignado valores a tres variables y por cada instrucción no existe al finalizar el [;], esto se debe a que *Python* no permite el uso de este signo en sus instrucciones. Dicho de otra manera, el valor 23 se ha asignado o colocado a la variable *"*numero*"* esto significa que esta variable ahora vale 23, lo mismo va a suceder con las variables *"*cadena*"* y *"*numero_real*"* a los que se les asignaron *"*Buenos días*"* y *123,1* respectivamente.

Las variables tienen las características de poder utilizar una longitud arbitraria de caracteres y pueden estar formadas por letras, guion bajo (_) y números. *Python* hace diferenciación de mayúsculas y minúsculas, en ese sentido "Numero" no es igual que "numero" ($Numero \neq numero$).

A continuación, podemos diferenciar nombres válidos y no válidos para nombre de variables:

Tabla 1

Nombres válidos y no válidos de variables.

Válidos	No válidos
carro	5Carro
Precio	Precio$
numero_real	Class
_numero	Def

Es importante indicar que *Python* tiene palabras reservadas que no pueden ser utilizadas para ninguna declaración de variables, estas son:

Tabla 2

Palabras reservadas de Python.

Palabra reservada	Explicación
and	Operador lógico que retorna True si ambas condiciones son verdaderas.
as	Sirve para declarar un alias.
assert	Comprueba si una expresión es verdadera y lanza una excepción si no lo es.
async	Declara una función como asíncrona.
await	Pausa una función asincrónica, hasta que la operación asincrónica se complete.
break	Termina la ejecución de la estructura.
Class	Define una clase.
continue	Continúa con la siguiente iteración.

Palabra reservada	Explicación
Def	Declara una función.
Del	Elimina una variable u objeto.
Elif	Declaración para condicionales.
Else	Declaración para condicionales.
except	Controlador de excepciones.
False	Valor booleano Falso.
finally	Código que se ejecuta independientemente de si se genera una excepción o no.
For	Declaración para bucles iterables sobre un rango.
From	Se utiliza para importar elementos específicos (funciones, clases) de un módulo.
global	Define una variable global.
If	Declaración para condicionales.
import	Importa un módulo u objeto.
In	Comprueba si un valor está en una secuencia.
Is	Compara la identidad de dos objetos.
lambda	Crea una función anónima.
nonlocal	Sirve para trabajar con variables dentro de funciones anidadas, donde la variable no debe pertenecer a la función interna.
None	Valor nulo.
Not	Negación lógica.
Or	O lógico.
Pass	Permite dejar estructuras vacías.
Raise	Lanza una excepción.
return	Devuelve un valor de una función.
True	Valor booleano verdadero.
Try	Declara un bloque que intenta ejecutar cierto código.
While	Declaración para bucles, mientras cumplan una condición.
With	Se utiliza para trabajar con recursos que necesitan ser gestionados y liberados de manera adecuada como, por ejemplo, archivos.
Yield	Define una función generadora.

Nota. *Adaptado de https://www.programacionfacil.org/python/python-basico/palabras-reservadas.html*

b) Tipo de datos

Como es costumbre en todos los lenguajes de programación, *Python* no es la excepción y ofrece una variedad de tipos de datos que permiten la reserva de memoria y el almacenamiento de valores en variables utilizadas en un sistema informático o software. Es importante destacar que la reserva de memoria en Python se realiza de manera automática al momento de declarar una variable.

En *Python* existe el concepto de la "Mutabilidad", que es una condición con las que cuentan los tipos de datos para ser modificados o no. Existen dos tipos de mutabilidad en *Python*, y estas son:

- Mutables: son aquellas variables que sí permiten ser modificados una vez creados. Entre ellos tenemos: Listas, Diccionarios, Sets y clases definidas por el usuario.
- Inmutables: son aquellas variables que no permiten ser modificados una vez creados. Entre ellos tenemos: Booleanos, Complejos, Enteros, Float, Frozenset, Cadenas, Tuplas, Range, Bytes.

Python tiene la ventaja de utilizar la mutabilidad por razones de seguridad, eficiencia e integridad. Con ello al aplicar la inmutabilidad gana eficiencia en el uso de la memoria ya que no necesita asignar un

nuevo bloque de memoria, lo que hace es compartir referencias a los mismos datos entre múltiples variables.

Además, con la inmutabilidad se permite la predictibilidad en la programación concurrente, esto en el caso de que varios hilos o procesos compartan datos inmutables, también la integridad de datos, optimización del intérprete de comandos o código y almacenamiento de valores calculados por caché.

Los tipos de datos que maneja *Python* son:

Tabla 3

Tipo de datos Python con ejemplos.

Tipo de datos	Descripción	Ejemplo
String	Es una cadena contigua de caracteres agrupados entre comilla. Python permite los dos tipos de comillas como son dobles y simples.	Cadena = "Buenos días" _nombres = 'Ruperto'
Set	Conjunto de colecciones desordenados de objetos únicos, existen 2 tipos Sets y Frozen Sets.	
Sets	Son mutables y se pueden agregar nuevos elementos una vez que se definen los conjuntos.	piezas_carro = {'llantas','motor','volante'} a = set('asdfgasdfga')

Tipo de datos	Descripción	Ejemplo
Frozen Sets	Son inmutables y no se pueden agregar nuevos elementos después de su definición.	b = frozenset('asdfagsa')
Numbers	Son todos los números como: int, float, complex, long.	num_int = 45 num_float = 45,5 num_complex = 45,5j num_long = 543234L
Boolean	Son datos lógicos utilizados para toma de decisiones.	True False
bytes	Son cadena de bytes para trabajar con datos binarios, por lo general son secuencias inmutables de byte.	cadena_byte = bytes([0X68, 0X6F, 0X6C, 0X61])

Nota. Ejemplos propuestos por los autores.

Es importante indicar que *Python* también reconoce como tipo de datos a estructuras más complejas como listas, tuplas, diccionarios y maps.

c) Estructuras de datos complejos

Existen tipo de datos más complejas, destinada a mejorar el rendimiento y funcionamiento de los programas o sistemas informáticos, estas estructuras de datos son parte de la familia de los tipos de datos de *Python* y son utilizadas por los programadores para manejo de grandes volúmenes de datos y en muchas ocasiones de diferentes tipos.

Estas estructuras son variables que pueden contener varios tipos de datos al mismo tiempo, es decir que una sola instrucción puede contener múltiples tipos de datos. Estas estructuras son útiles cuando se requiere extraer datos de archivos o bases de datos.

Listas

Es una variable compleja que permite almacenar una colección organizada de elementos, es decir, están estructuradas en casillas según su longitud y organizada con índices que van desde el 0 hasta el valor de su longitud -1.

Las listas tienen el siguiente formato:

$$Nombre\_lista = [obj_1, obj_2, obj_3, \dots, obj_n]$$

***Nombre_lista**:* representa el nombre de la variable donde se almacenará la lista.

[ ]: Son símbolos utilizados en *Python* para definir una lista.

***obj_1,obj_2,obj_3,…,obj_n*:** Son los elementos de la lista.

Como ya hemos indicado anteriormente, este tipo de estructura de datos puede contener cualquier tipo de datos y es ampliamente

utilizada para manejo de valores extraídos desde un archivo, base de datos, scripts json, entre otros. Para una mejor comprensión, se presentan varios ejemplos relacionados con el manejo y manipulación de listas.

Ejemplo.

Creación de lista.

```
lista = [1,"Hola",34.3,34.6j,'Mundo']
print(lista)
[1, 'Hola', 34.3, 34.6j, 'Mundo']
```

Mostrar elementos de una lista según su índice.

```
print(lista[2])
34.3
```

Lista dentro de otra.

```
lista_general = [23,9,lista,"Fin"]
print(lista_general)
[23, 9, [1, 'Hola', 34.3, 34.6j,
'Mundo'], 'Fin']
```

Elementos desde el 2 hasta el índice 5-1.

```
print(lista[2:5])
[34.3, 34.6j, 'Mundo']
```

Elementos del índice 1 al 4 de 2 en 2.

```
print(lista[1:5:2])
['Hola', 34.6j]
```

Recorrido de una lista.

```
lista = [1,"Hola",34.3,34.6j,'Mundo']
for elemt in lista:
    print(elemt)
1
Hola
34.3
34.6j
Mundo
```

Añadir elementos a una lista.

```
lista.append("Añadido")
print(lista)
[1, 'Hola', 34.3, 34.6j, 'Mundo', 'Añadido']
lista.extend([100,"siguiente","último"])
print(lista)
[1, 'Hola', 34.3, 34.6j, 'Mundo', 'Añadido',
100, 'siguiente', 'último']
```

Modificar elemento de la lista.

```
lista[5]= "valor modificado"
print(lista)
[1, 'Hola', 34.3, 34.6j, 'Mundo', 'valor
modificado', 100, 'siguiente', 'último']
```

Eliminar n elemento de la lista.

```
del lista[5]
print(lista)
[1, 'Hola', 34.3, 34.6j, 'Mundo', 100,
'siguiente', 'último']
lista.remove(100)
print(lista)
[1, 'Hola', 34.3, 34.6j, 'Mundo', 'siguiente',
'último']
```

Longitud o tamaño de la lista.

```
lista = [1, 'Hola', 34.3, 34.6j, 'Mundo',
'siguiente', 'último']
len(lista)
```

Una de las características particulares de *Python* es que tiene una clase predefinida denominada ***List***, que permite trabajar con el tipo de dato *List,* y puede ser utilizada en el código del proyecto cunado se requiera, la siguiente tabla es un compendio de los métodos con los que cuenta esta clase.

Tabla 4

Métodos de la clase list.

Método	Descripción
append()	Añade un nuevo elemento al final de la lista.
extend()	Añade un grupo de elementos (iterables) al final de la lista.
insert(indice, elemento)	Inserta un elemento en una posición concreta de la lista.
remove(elemento)	Elimina la primera ocurrencia del elemento en la lista.
pop([i])	Obtiene y elimina el elemento de la lista en la posición i. Si no se especifica, obtiene y elimina el último elemento.
clear()	Borra todos los elementos de la lista.
index(elemento)	Obtiene el índice de la primera ocurrencia del elemento en la lista. Si el elemento no se encuentra, se lanza la excepción ValueError.
count(elemento)	Devuelve el número de ocurrencias del elemento en la lista.
sort()	Ordena los elementos de la lista utilizando el operador <.
reverse()	Obtiene los elementos de la lista en orden inverso.
copy()	Devuelve una copia poco profunda de la lista.

Nota. Adaptado de https://j2logo.com/python/tutorial/tipo-list-python/ (Lozano, 2023)

Tupla

Este tipo de estructura de datos es muy parecido a las listas, con la única diferencia de que no se pueden actualizar, es decir son inmutable. La inmutabilidad de las tuplas, se refiere a que no se puede añadir ni eliminar valores o elementos de una tupla después de haber sido creada.

> *"La clase tuple en Python es un tipo contenedor, compuesto, que en un principio se pensó para almacenar grupos de elementos heterogéneos, aunque también puede contener elementos homogéneos". (Lozano, 2023)*

> *"Las Python tuples son colecciones de datos idénticos o distintos clasificados con un índice y que no pueden ser modificados. Se crean de la misma manera que las listas de este lenguaje de programación". (IONOS, 2023)*

Este tipo de estructuras complejas de datos es sumamente útil cuando el programador en algún momento del sistema informático o programa, necesita que el contenido de la variable conserve su forma y/o composición definida desde el inicio.

Las tuplas tienen el siguiente formato:

$$Nombre\_tupla = (obj_1, obj_2, obj_3, \dots, obj_n)$$

Nombre_ tupla: representa el nombre de la variable donde se almacenará la tupla.

( ): Son símbolos utilizados en *Python* para definir una tupla.

***obj_1,obj_2,obj_3,…,obj_n*:** son los elementos de la tupla.

Ejemplo.

Creación e impresión de la tupla, hay que tener en cuenta que cuando se crea una tupla con un solo valor siempre debe terminar con coma.

```
tupla =
(12,"Pepe","Perez",22,'M',"Soltero","Estudiante")
print(tupla)
(12, 'Pepe', 'Perez', 22, 'M', 'Soltero',
'Estudiante')
t = ("Hola",)
```

Selección e impresión de un elemento de la tupla.

```
tupla =
(12,"Pepe","Perez",22,'M',"Soltero","Estudiante")
print(tupla[3])
```

Consultar varios valores.

```
tupla =
(12,"Pepe","Perez",22,'M',"Soltero","Estudiante")
print(tupla[2:5])
('Perez', 22, 'M')
```

Convertir una tupla en lista.

```
tupla =
(12,"Pepe","Perez",22,'M',"Soltero","Estudiante")
lista = list(tupla)
print(lista)
[12, 'Pepe', 'Perez', 22, 'M', 'Soltero', 'Estudiante']
```

Convertir una lista en tupla.

```
lista =
[3,"María","Velóz",25,'F',"Casada","Estudiante"]
tupla_nueva= tuple(lista)
print(tupla_nueva)
(3, 'María', 'Velóz', 25, 'F', 'Casada', 'Estudiante')
```

Eliminar tupla.

```
del tupla_nueva
```

Si a una variable se le asigna una secuencia de valores o elementos separados por comas, Python reconoce ese código como tupla. A esto se le conoce como "Empaquetado de tuplas"

Empaquetado de tuplas.

```
a = 4
b = "Carlos"
c = "Loza"
d = 20
e = 'M'
f = 'Casado'
g = "Estudiante"
h = a,b,c,d,e,f,g
print(h)
(4, 'Carlos', 'Loza', 20, 'M', 'Casado', 'Estudiante')
```

Función index y count.

```
tupla = (12, 'Pepe', 'Pérez', 22, 'M', 'Soltero',
'Estudiante')
tupla.index("Estudiante")
6
tupla.count("Estudiante")
1
```

Función index: devuelve la posición enla que se encuentra un elemento dentro de la lista.

Función count: devuelve la cantidad de veces que un elemento se repite en la lista

Diccionarios

Son estructuras de datos complejas donde se pueden almacenar una colección de elementos o valores con una disposición compuesta por una clave (key) y un valor (value) encerrados entre corchetes { }.

A continuación, mencionaremos varias definiciones de diccionarios de *Python*.

> *"Los diccionarios son poderosas estructuras de datos en Python que almacenan datos como pares de claves, siendo está representada en la siguiente forma: Clave-Valor". (Priya, 2021)*

Según Wachenchauzer (2019):

> *"Un diccionario es una colección no-ordenada de valores que son accedidos a través de una clave. Es decir, en lugar de acceder a la información mediante el índice numérico, como es el caso de las listas y tuplas, es posible acceder a los valores a través de sus claves, que pueden ser de diversos tipos". (p. 5)*

Los criteros antes mencionados coinciden que los diccionarios son una colección de valores que están conformados por una clave asociada a un valor, hay que tener en cuenta varios aspectos para garantizar la sintaxis e integridad de la estructura de datos.

1. Las claves (key) son únicas dentro del diccionario.
2. No hay restricción para que un valor este asignado a varias claves.
3. No podemos acceder a una clave a través de un valor (value).

4. No existe un orden particular con los elementos del diccionario.
5. Cualquier variable de tipo inmutable como cadena, tupla, enteros, entre otros, pueden ser clave (key) de un diccionario.

Los diccionarios tienen el siguiente formato:

$$Nombre\_dicionario = \{'clave\ 1': valor\ 1, 'clave': valor\ 2, \dots\}$$

***Nombre_ dicionario**:* representa el nombre de la variable donde se almacenará el diccionario.

{ }: Son símbolos utilizados en *Python* para definir el diccionario.

***obj_1,obj_2,obj_3,…,obj_n*:** Son los elementos del diccionario.

Creación de diccionario.

```
diccionario = {
    'id': 4,
    'nombre': "Carlos",
    'apellidos':"Loza",
    'edad':20,
    'genero': "M",
    'estado_civil':"Casado",
    'rol':"estudiante" }
print(diccionario)
{'id': 4, 'nombre': 'Carlos', 'apellidos': 'Loza',
'edad': 20, 'genero': 'M', 'estado_civil':
'Casado', 'rol': 'estudiante'}
```

```
diccionario_2 = dict(
    idx = 4,
    nombre = "Carlos",
    apellidos = "Loza",
    edad = 20,
    genero = "M",
    estado_civil = "Casado",
    rol = "estudiante")
print(diccionario_2)
{'idx': 4, 'nombre': 'Carlos', 'apellidos':
'Loza', 'edad': 20, 'genero': 'M', 'estado_civil':
'Casado', 'rol': 'estudiante'}
```

Acceder a los elementos de un diccionario.

```
{'idx': 4, 'nombre': 'Carlos', 'apellidos':
'Loza', 'edad': 20, 'genero': 'M', 'estado_civil':
'Casado', 'rol': 'estudiante'}
print(diccionario_2['estado_civil'])
Casado
print(diccionario_2.get('estado_civil'))
Casado
```

Modificar elemento de un diccionario.

```
{'idx': 4, 'nombre': 'Carlos', 'apellidos':
'Loza', 'edad': 20, 'genero': 'M', 'estado_civil':
'Casado', 'rol': 'estudiante'}
diccionario_2['nombre'] = "Vanessa"
print(diccionario_2)
{'idx': 4, 'nombre': 'Vanessa', 'apellidos':
'Loza', 'edad': 20, 'genero': 'M', 'estado_civil':
'Casado', 'rol': 'estudiante'}
```

Iterar diccionario.

```
for clave in diccionario_2:
    print(diccionario_2[clave])
4
Vanessa
Loza
20
M
Casado
estudiante
```

Diccionarios anidados.

```
estudiantes = {
    "Estudiante 1": diccionario,
    "Estudiante 2": diccionario_2 }
print(estudiantes)
{'Estudiante 1': {'id': 4, 'nombre': 'Carlos',
'apellidos': 'Loza', 'edad': 20, 'genero': 'M',
'estado_civil': 'Casado', 'rol': 'estudiante'},
'Estudiante 2': {'idx': 4, 'nombre': 'Carlos',
'apellidos': 'Loza', 'edad': 20, 'genero': 'M',
'estado_civil': 'Casado', 'rol': 'estudiante'}}
```

Métodos para los diccionarios de *Python*.

Tabla 5

Métodos para diccionarios en Python.

Método	Descripción
clear()	elimina todo el contenido del diccionario.
items()	devuelve una lista con los keys y values del diccionario.
keys()	devuelve una lista con todas las keys del diccionario.
values()	devuelve una lista con todos los values o valores del diccionario.
popitem()	elimina de manera aleatoria un elemento del diccionario.
update(<obj>)	se llama sobre un diccionario y tiene como entrada otro diccionario. Los value son actualizados y si alguna key del nuevo diccionario no está, es añadida.

Nota. Elaborado por los autores.

1.1.3. Estructuras de control

Las estructuras de control son instrucciones o secuencias que permite la interactividad de los usuarios con los datos y el sistema. Toda herramienta de desarrollo de software utiliza las estructuras de datos para la implementación o creación de programas.

"Una estructura de control, es un bloque de código que permite agrupar instrucciones de manera controlada". (Bahit, 2012)

"Las estructuras de control son herramientas que nos permiten condicionar la ejecución del código a ciertas circunstancias". (Apuntes, 2023)

En *Python*, las estructuras de control más comunes son las condicionales y los bucles. Pero antes de hablar de estructuras de control, tenemos que entender la "identación".

La identación no es más que las sangrías que se usa por cada instrucción que se escribe en un programa, Python es muy respetuoso de la identación ya que en esta herramienta no existe el ";" para indicar que ha terminado una instrucción. Además, se utiliza para definir bloques de código dependiendo del nivel de la identación

a) Estructura de control secuenciales

Como su nombre lo indica son instrucciones de código fuente que se ejecutan de manera secuencial desde arriba hacia abajo y mientras se termine el programa o sistema informático.

Estas estructuras son fundamentales para el desarrollo de programas o sistemas informáticos ya que permiten gestionar de manera óptima flujos de ejecución de instrucciones en un orden específico de arriba abajo y de derecha a izquierda en un programa.

Una instrucción es una línea de código que refleja una orden que el programa debe ejecutar para satisfacer una secuencia, ciclo o decisión.

Una instrucción en Python se escribiría de la siguiente manera:

```
nombre = input("Ingrese su nombre: ")
```

Entonces la estructura secuencial sería:

```
nombre = input("Ingrese su nombre: ")
apellido = input("Ingrese su apellido: ")
edad = input("Ingrese su edad: ")
print(f"Usted es : {nombre} {apellido} y tiene {edad}
años")

Ingrese su nombre: Pepe
Ingrese su apellido: Pérez
Ingrese su edad: 22
```

b) Estructura de control condicionales

Son instrucciones orientadas a la toma de decisiones en base a una condición específica que el programa o sistema informático lo requiera. Se utilizan para evaluar una o varias condiciones dentro de una instrucción; este tipo de estructuras se evalúan con verdadero (True) cuando cumple una determinada condición, o falso (False) cuando no se cumple.

Este tipo de estructuras de control requiere de un conjunto de operadores lógicos para su funcionamiento y/o ejecución, los operadores lógicos son conjuntos de símbolos que se utilizan para establecer si una afirmación es verdadera o falsa y proporcionar al flujo de ejecución de un programa el camino que debe seguir en su ejecución, a continuación, se muestra un conjunto de operadores más utilizados en el desarrollo de programas.

Tabla 6

Operadores lógicos.

Operador	Significado	Instrucción	Respuesta
==	Igual que	100 == 700	FALSO
!=	Distinto que	2 != 4	VERDADERO
<	Menor que	20 < 50	VERDADERO
>	Mayor que	50 > 20	VERDADERO
<=	Menor o igual que	20 <= 20	VERDADERO
>=	Mayor o igual que	30 >= 50	FALSO

Nota: Elaborado por los autores.

El desarrollo de programas cada vez es más complicado y requiere características más amplias para la toma de decisiones, para ello se recurren en ocasiones a los operadores lógicos con condiciones compuestas.

Ejemplo.

Tabla 7

Ejemplo de operadores simultáneos o compuestos.

Operador	Instrucción	Respuesta	Resultado
and (y)	15 == 17 and 8 < 9	0 y 1	FALSO
	6 < 10 and 10 > 5	1 y 1	VERDADERO
	9 < 12 and 12 > 15	1 y 0	FALSO
or (o)	20 == 20 or 5 < 3	1 o 0	VERDADERO
	17 > 15 or 19 < 20	1 o 1	VERDADERO
xor*	(8 == 8) ^ (9 > 3)	1 o 1	FALSO
(o excluyente)	(8 == 8) ^ (9 < 3)	1 o 0	VERDADERO

Nota. Elaborado por los autores, donde O es falso y 1 es verdadero.

Sentencia condicional básica [*if*]:

Es una de las instrucciones más utilizadas en la implementación de programas o sistemas informáticos, tiene la capacidad de iniciar una sentencia condicional y prepara el camino para una decisión verdadera o falsa.

Sintaxis:

$$if\ [condición]:$$
$$\quad [instrucción]$$
$$elif\ [Condición]:$$
$$\quad [instrucción]$$
$$else:$$
$$\quad [instrucción]$$

Ejemplo.

```
nombre = input("Ingrese su nombre: ")
apellido = input("Ingrese su apellido: ")
edad = int(input("Ingrese su edad: "))
if edad >= 18:
    print(f"{nombre} {apellido} es mayor de edad")
else:
    print(f"{nombre} {apellido} es menor de edad")
Ingrese su nombre: Pepe
Ingrese su apellido: Pérez
Ingrese su edad: 22
Pepe Pérez es mayor de edad
```

La estructura condicional if-elif-else en *Python* se utiliza cuando deseas evaluar múltiples condiciones y ejecutar bloques de código diferentes según el resultado de estas condiciones.

Ejemplo.

```
numero = int(input("Ingresa un número: "))
if numero > 0:
    print("El número es positivo.")
elif numero < 0:
    print("El número es negativo.")
else:
    print("El número es cero.")

Ingresa un número: -1
El número es negativo.
```

c) Estructura de datos bucles o repetitivas

Son estructuras de control diseñada para ejecutar un bloque de instrucciones y/o secuencias de manera repetitiva, hasta que se cumpla una condición o por un rango definido de números.

Bucle o ciclo [*for*]

Se utiliza para iterar (repetir) sobre una secuencia de valores o elementos contenidos en una lista, una tupla, una cadena de texto o cualquier objeto iterable.

Iterable, es la capacidad que tiene un artificio, objeto, instrucción o sentencia para repetirse según su condición.

Sintaxis:

$$for\ [elemento]\ in\ [objeto\ iterable]:$$
$$[instrucción]$$

for: es una palabra clave que indica que se va a iterar una secuencia de elementos.

[elemento]: variable que representa a cada uno de los elementos del objeto iterable.

in: palabra reservada para conectar el elemento con el objeto iterable.
[objeto iterable]: secuencia de elementos (lista, tupla, conjunto, cadena, etc.).

:: Los dos puntos indican el comienzo del bloque de instrucciones.

[instrucción]: son las acciones o el código que se ejecutarán.

Ejemplo.

```
lista = [1,"Hola",34.3,34.6j,'Mundo']
for elemt in lista:
    print(elemt)
1
Hola
34.3
34.6j
Mundo
```

Recorrido de un diccionario.

```
for clave in diccionario_2:
    print(diccionario_2[clave])
4
Vanessa
Loza
20
M
Casado
estudiante
```

Bucle o ciclo [*while*]

Se utiliza para ejecutar un bloque de código con instrucciones, mientras se cumpla una condición. Mientras la condición se cumpla, es decir, sea verdadera, el bloque de código se ejecutará en un ciclo continuo, cuando la condición no se cumpla la ejecución del bloque terminará.

Sintaxis:

$$[variable\ contador] = 1$$
$$while\ \ [variable\ contador]\ [operador\ lógico][limite\ de\ la\ condición]:$$
$$\quad [instrucción]$$
$$\quad [variable\ contador]\ +=\ 1$$

variable contador: inicialización de la variable contador.

while: inicia un bucle que se repite mientras se cumpla una condición.

[variable contador]: variable que controla la interación

[operador lógico]: operador que compara la variable contador y la vaiable limite de la condición.

[limite de la condición]: límite de la interación.

Ejemplo.

```
numero = int(input("Escriba un número positivo: "))
while numero < 0:
    print("¡Su número negativo! Inténtelo de nuevo")
    numero = int(input("Escriba un número positivo:
"))
print("Muy bien!!!")
Escriba un número positivo:  -6
¡Su número negativo! Inténtelo de nuevo
Escriba un número positivo: -3
¡Su número negativo! Inténtelo de nuevo
Escriba un número positivo: 33
```

Además de estos bucles básicos, *Python* ofrece palabras clave como break (para salir de un bucle antes de que se complete su ejecución) y continue (para saltar a la siguiente iteración del bucle) para controlar el flujo dentro de los ciclos.

Estructura [with]

Si bien la declaración [with] no se considera un bucle o ciclo, es importante estudiarla y comprender su uso, ya que su funcionamiento es similar. Se emplea con frecuencia en contextos que requieren una configuración inicial y una limpieza automática al finalizar su ejecución.

With, es frecuentemente utilizada cuando se requiere tener incidencia de apertura y cerrado de archivos, conexión y desconexión a base de datos, entre otras. Esta estructura garantiza que los recursos utilizados se van a liberar en el preciso instante que se salga del contexto donde se requiere este recurso.

Sintaxis:

$$with\ [gestor\ de\ contexto]\ as\ [Objeto\ devuelto]:$$
$$[instrucción]$$

with: inicia un bloque que hace uso de un gestor de contexto.

[gestor de contexto]: objetos que gestionan la inicialización y el cierre de los recursos.

as: palabra clave que permite asignar el valor devuelto por el gestor de contexto.

Objeto devuelto: valor devuelto cuando se ejecuta el método del gestor de contexto.

:: Los dos puntos indican el comienzo del bloque de instrucciones.

[instrucción]: son las acciones o el código que se ejecutarán.

Ejemplo.

```
with open('archivo.txt', 'w') as archivo:
    archivo.write('Escritura de un archivo .\n')
    archivo.write('Para comprobar el manejo de
with.')
```

1.2. Framework Flask

El lenguaje de programación Python es considerado uno de los lenguajes más versátiles y sencillos de utilizar para la implementación de todo tipo de aplicaciones informáticas. Pero, Python lo puede hacer el trabajo sólo, requiere la ayuda de framework que permita manejar solicitudes HTTP, bases de datos, sesiones que son elementos fundamentales para implementar una solución informática completa.

Entre los frameworks más conocidos en el mercado, podemos encontrar una variedad de herramientas que pertenecen a este grupo. Entre ellas se destacan Django, Flask, Bottle, FastAPI y Tkinter. Cada una de estas ofrece un entorno de desarrollo adecuado para diseñar e implementar aplicaciones web funcionales y robustas, adaptadas a las necesidades de los usuarios.

El framework Flask es considerado por muchos uno de los más ligeros y sencillos de utilizar. Algunos programadores lo consideran un "microframework" porque proporciona solo las características esenciales para desarrollar aplicaciones web, permitiendo a los desarrolladores elegir las bibliotecas y extensiones que mejor se adapten a sus necesidades para funciones más específicas.

Flask utiliza decoradores como una herramienta clave para definir rutas que están asociadas a scripts ejecutables, tales como páginas de vista o interfaces de usuario. Estos decoradores permiten mapear URLs específicas a funciones de Python, lo que facilita la creación de aplicaciones web dinámicas. A través de estos decoradores, Flask vincula cada ruta con una función que se ejecuta cuando un usuario accede a esa URL en el navegador.

Por lo general, es utilizado para desarrolla aplicaciones con el patrón Modelo Vista Controlador (MVC).

Los decoradores en Flask son funciones especiales que se utilizan para modificar el comportamiento de una función en el contexto de una aplicación web.

1.2.1. Instalación de Flask

La instalación del framework *Flask* es sencilla e intuitiva, lo primero que hay que tener en cuenta es asegurarse que el proyecto haya sido creado, el entorno virtual instalado, y las librerías correspondientes referenciadas. Luego, para la instalación del framework, se necesita ejecutar un "terminal o shell", ya sea desde el CMD de *Windows* o desde el editor de código elegido para el desarrollo de la aplicación web.

Para instalar esta herramienta se debe utilizar el comando *Package Installer Python* más conocido como "*pip*".

```
(venv)D:\Mi_Proyecto>pip install flask
```

Es importante indicar que flask cuenta con un conjunto de extensiones o paquetes adicionales que añaden funcionalidad a una aplicación Flask, a pesar de ello es recomendable instalar sólo lo que necesite o requiera la aplicación o sistema informático que se vaya a desarrollar.

Tabla 8

Extensiones del framework flask.

Extensiones	Descripción
flask-script	Permite tener un comando de la línea de comando para manejar la aplicación.
flask-Bootstrap	Hojas de estilo para la página.
flask-WTF	Sirve para generar formularios de HTML con clases y objetos.
flask-Sqlalchemy	Sirve para poder generar el modelo de datos.
flask-login	Sirve para la autenticación de usuario y contraseña.
Flask-CORS	Habilita el intercambio de recursos entre diferentes dominios en aplicaciones de API RESTful a través de peticiones de origen cruzado (CORS).
Flask-Migrate	Facilita la administración de migraciones de bases de datos.
Flask-Mail	Permite enviar correos electrónicos desde tu aplicación Flask.

Extensiones	Descripción
Flask-RESTful	Proporcionar una estructura y funcionalidades específicas para la creación de recursos y rutas REST.
Flask-Principal	Proporciona una forma de gestionar roles y permisos de usuarios en una aplicación Flask.
Flask-Uploads	Facilita la carga y el manejo de archivos en aplicaciones Flask.

1.2.2. Instancia de la clase flask

Para utilizar *flask* en la implementación de una aplicación o sistema informático, es necesario instanciar la clase *Flask,* esta instanciación permitirá que los objetos de la aplicación requieran por obligación que exista una sola instancia a la vez.

Cada vez que se proceda a instanciar la clase *Flask* hay que determina el parámetro "*__name__"*, que representa el nombre del módulo actual, este argumento ayuda a la aplicación a saber dónde encontrar recursos como plantillas y archivos estáticos. De esta manera la declaración de la clase *Flask*, permite inicializar una aplicación *flask* y establece su contexto dentro del módulo actual, lo que permite manejar rutas, configuraciones y ejecutar el servidor web.

La clase *Flask*, requiere algunos decoradores que es importante conocer y considerar en algún momento dentro del proceso de codificación del proyecto de software, entre ellos tenemos los siguientes decoradores:

Tabla 9

Decoradores flask.

Decoradores	Descripción
@app.route	Se utiliza para asociar una función con una ruta específica en tu aplicación.
@app.before_request y @app.after_request	se utilizan para ejecutar funciones antes y después de cada solicitud HTTP.
@login_required	Se utiliza para proteger vistas o rutas, asegurándose de que el usuario haya iniciado sesión antes de acceder a ellas.
@app.errorhandler	Se utiliza para manejar errores específicos o excepciones en tu aplicación.
@app.route('/api') con methods=['POST', 'GET']	Se utilizar para especificar qué métodos HTTP son compatibles con una ruta específica.
@app.route('/ruta/<variable>')	Se utilizar para las variables en tus rutas definidas en los decoradores para capturar valores de la URL y utilizarlos en tu función.

Ejemplo.

```
from flask import Flask

app = Flask(__name__)
@app.route('/')
def home():
    return 'Hello World!'
```

La variable ***app***, permitirá asociar una función a una ruta específica en la aplicación. El parámetro ***__name__*** es necesario para que el objeto ***app*** sepa dónde encontrar las plantillas de nuestra aplicación o los ficheros estáticos.

Para definir la ruta, *flask* tendrá que utilizar el decorador ***route*** para indicar la URL a ser ejecutada por su correspondiente función. Seguidamente debe definirse la función que debe generar internamente las URLs, misma que deberá devolver una respuesta que será presentada en el navegador del usuario.

El sistema de enrutamiento que *flask* utiliza es el sistema de enrutamiento ***Werkzeug***, este permite organizar automáticamente las rutas, de esta manera no le va a importar el orden que se declare o implemente las rutas del proyecto software. Una de las características de ***Werkzeug*** es que asegura que las URLs sean únicas.

Werkzeug es una biblioteca de Python que proporciona herramientas para la creación de enrutamiento en aplicaciones web, para gestionar el enrutamiento de URL y las solicitudes HTTP.

1.2.3. Motor de plantillas Jinja2

El motor de plantillas en *Python* es uno de los más utilizados por desarrolladores web en la actualidad, ya que permite separar la lógica de presentación o vista con el controlador o contenido dinámico según el patrón implementado en el proyecto.

El *Jinja2* es utilizado para generar contenido *HTML, XML* u otros formatos de texto para la web, se puede combinar con framework *flask* o *Django*.

Para instalar este motor de plantilla ***Jinja2***, se debe utilizar el comando *Package Installer Python* más conocido como "***pip***".

```
(venv)D:\Mi_Proyecto>pip install Jinja2
```

Jinja2 utiliza los siguientes elementos para establecer o invocar sus plantillas:

a) Variables, se indican con {{ ... }}
b) Instrucciones, se indican con {% ... %}
c) Comentarios, se indican con {# ... #}

Teniendo en cuenta lo antes mencionado cuando aplicamos un filtro de variable dentro de un código *HTML*, *Jinja2* utilizará los corchetes {{ … }} de la siguiente forma.

```
<div class="alert alert-primary alert-dismissible"
role="alert">
   <strong>{{message}}</strong>
   <button type="button" class="btn-close" data-bs-
dismiss="alert" aria-label="Close"></button>
</div>
```

El elemento {{*message*}} es una variable que acogerá desde la vista, todo lo que el controlador le envié para presentar al usuario. De esta manera se puede embeber el código *Jinja2* para la comunicación entre capas del modelo o patrón metodológico a utilizar.

En lo que respecta a sentencias que se requiere ejecutar en el cliente como bucles o instrucciones en *Jinja2* se utiliza {% … %}, de la siguiente manera.

```
<select name="idPersona" class="form-select" aria-
label="Default select example">
   <option>Seleccionar los Nombres y
Apellidos</option>
   {% for du in data %}
      <option  value="{{du.id}}">{{du.strNombres}}
{{du.strApellidos}}</option>
   {% endfor %}
</select>
```

Desde el controlador o contenido dinámico se envía un conjunto de elementos contenidos en una lista llamada “data”, mediante la estructura de control ***for*** se recorre esta lista por medio de la variable “du” y se muestra al usuario mediante las variables {{du.strNombres}} {{du.strApellidos}}, esto hasta terminar con los elementos de la lista.

Es importante indicar que con el bloque ***for***, permite tener acceso a varias variables, como se detalla a continuación:

1. loop.index: La iteración actual del bucle (empieza a contar desde 1).
2. loop.index: La iteración actual del bucle (empieza a contar desde 0).
3. loop.first: True si estamos en la primera iteración.
4. loop.last: True si estamos en la última iteración.
5. loop.length: Número de iteraciones del bucle.

Otro ejemplo de utilización de los comodines Jinja2 es el *if.*

```
{% with messages = get_flashed_messages() %}
{% if messages %}
{% for message in messages %}
   <div class="alert alert-primary alert-
dismissible" role="alert">
      <strong>{{message}}</strong>
      <button type="button" class="btn-close" data-
bs-dismiss="alert" aria-label="Close"></button>
</div>
{% endfor %}
{% endif %}
{% endwith %}
```

Otro aspecto de tomar en cuenta cuando se trabaja con el motor *Jinja2*, es la herencia de plantillas, esta funcionalidad nos permite contar con una estructura o esquemas de plantillas que podrán ser utilizados por todas las páginas que formen parte del proyecto web. Es decir, permite crear plantillas o template base (o también conocidas como "layout") para establecer una estructura común en las páginas que forman parte del proyecto luego extender esas plantillas de modo que el resto de páginas *HTML* contenga el contenido de estas plantillas.

Para entender mejor la herencia de plantilla, vamos a considerar un ejemplo sencillo.

Plantilla base

```
<!DOCTYPE html>
<html lang="es">
<head>
<title>{% block title %}{% endblock %}</title>
<link rel="stylesheet"
href="{{url_for("static",filename='css/style.css')}}">
<meta charset="utf-8" />
</head>
<body>
    <header>
       <h1>Hola mundo!!</h1>
    </header>
    {% block content %}{% endblock %}
</body>
```

Plantilla extendida

```
{% extends "base.html" %}
{% block title %} Bienvenido, {{nombre}}{% endblock %}
{% block content %}
    <h2>Plantilla de prueba</h2>
    {% if nombre %}
      <h1>Hola {{nombre}}</h1>
    {%else%}
      <p>Ingresa tu nombre nuevamente</p>
    {% endif %}
{% endblock %}
```

1.2.4. Iniciando la aplicación

Teniendo en cuenta que la aplicación de *Python* debe tener un punto de inicio para comenzar a ejecutar su codificación, no debe faltar

el ***__main__*** para indicarle al scripts que debe ejecutarse con el ***app.run().***

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

Cada vez que desde el terminal se ejecute el script principal, el depurador se dirigirá a esta instrucción y ejecutará el programa *Python*.

1.3. Bootstrap

Es un conjunto de herramientas que contiene una colección de componentes predefinidos para interfaces responsivas y modernas; con funciones, componentes prediseñados y complementos de *javascript* para proyectos web.

"Bootstrap is a CSS framework that makes it easier to create website and web application user interfaces. Bootstrap is especially useful as a base layer of CSS to build sites with responsive web design". (Makai, 2022)

Es un moderno y potente framework para el diseño de páginas web (FrontEnd) que permite crear interfaces de usuario, de una manera ágil, fácil y amigable. Bootstrap cuenta con un archivo llamado bootstrap.css que es el script principal, y contiene la definición de todos

los estilos utilizados, la estructura de este framework está compuesta por dos directorios; el ***css y js***.

css: contiene los archivos necesarios para la estilización de los elementos y una alternativa al tema original;

js: contiene la parte posterior del archivo bootstrap.js (original y minificado[1]), responsable de la ejecución de aplicaciones de estilo que requieren manipulación interactiva.

Hay dos formas de utilizar el framework Bootstrap:

Instalar a través del administrador de paquetes: es una forma de instalación de los archivos fuente *Sass (Syntactically Awesome Stylesheets)* y *JavaScript* de *Bootstrap* a través de npm, RubyGems, Composer o Meteor. Este método consiste en tener los archivos de Bootstrap en tu proyecto localmente, puedes descargarlos desde el sitio web oficial de Bootstrap (https://getbootstrap.com) y agregarlos a tu proyecto. La forma de incluir el framework *Bootstrap*, dependerá de que necesidades tenga el proyecto.

```
$ npm install bootstrap@5.3.2
$ gem install bootstrap -v 5.3.2
```

[1] Proceso de eliminar caracteres innecesarios de archivos de código Fuente.

Incluir a través de CDN (Content Delivery Network): consiste en incluir los archivos *CSS* y *JavaScript* desde un *CDN*. Esto implica agregar enlaces a las siguientes URLs en la sección <head> de tus páginas *HTML*.

```
<link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/di
st/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-
T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCmG1MXxSR1GAsXEV/Dwwy
kc2MPK8M2HN" crossorigin="anonymous">
```

```
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dis
t/js/bootstrap.bundle.min.js" integrity="sha384-
C6RzsynM9kWDrMNeT87bh95OGNyZPhcTNXj1NW7RuBCsyN/o0jlpc
V8Qyq46cDfL" crossorigin="anonymous"></script>
```

1.4. Entornos virtuales

Cuando se construyen aplicaciones en *Python* por lo general se utilizan módulos y paquetes con librerías específicas, es decir, que se requiere de una versión específica de la librería para solventar algún tipo de bug que ha surgido y esta podrá solucionar el problema de manera eficiente.

Según Bustamante (2021), explica que un *virtualenv* es una herramienta que se utiliza para crear entornos *Python* aislados. “Crea

una carpeta que contiene todos los ejecutables necesarios para usar los paquetes que necesitaría un proyecto de *Python"* (párr. 4).

Entre los beneficios que ofrecen los entornos virtuales tenemos:

1. Puede subsistir varios entornos con varios paquetes y no va a existir conflictos entre ellos;
2. El proyecto estará listo para ser publicado con sus propios módulos dependientes; y,
3. Se puede satisfacer los requerimientos de distintos proyectos al mismo tiempo.

1.4.1. Instalación del entorno virtual

Antes de instalar un entorno virtual en un proyecto *Python*, lo primero que hay que hacer es crear el directorio del proyecto, este puede ser creado desde una interfaz de línea de comandos del sistema operativo o utilizando el asistente Wizard de sistema operativo.

Cada entorno virtual está diseñado para contener su propia instalación de *Python* y sus referencias o dependencias de bibliotecas para gestionar de manera independiente sus proyectos. Esto convierte a los entornos virtuales en una herramienta versátil para trabajar en múltiples proyectos con diferentes procesos o procedimientos.

Para instalar el entorno virtual se requiere ejecutar el siguiente comando:

```
pip install virtualenv
```

Instalada la herramienta el siguiente paso es crear el entorno virtual para tu proyecto actual, eso lo tendrás que hacer dentro del directorio del proyecto.

```
virtualenv [nombre_del_entorno_virtual]
```

directorio del proyecto, una carpeta con el nombre del entorno virtual, ingresado en la instrucción anterior.

Una vez que has creado el entorno virtual, debes activarlo cada vez que desees trabajar en tu proyecto. Para ello se deberá posicionar el cursor en el directorio del entorno virtual y ejecutar la siguiente instrucción.

```
.\venv\Scripts\activate
```

La forma de comprobar que estamos dentro del entorno, es cuando en el *prompt*[2] de la interfaz de línea de comandos, comienza con el nombre del entorno virtual entre paréntesis.

```
(nombre_del_entorno_virtual) PS D:\carpeta_proyecto>
```

Después de activar el entorno virtual, se puede proceder a usar *"pip"* para instalar las dependencias específicas del proyecto en desarrollo. Cuando se ha terminado la codificación del proyecto, es conveniente desactivar el entorno virtual con el comando *deactivate*, esto garantizará que el proyecto tenga su propio conjunto aislado de bibliotecas y no interfiera con otros proyectos.

Conclusiones

- La herramienta de desarrollo *Python* es multipropósito, multiplataforma y permite la creación de aplicaciones web robustas, sostenibles y dinámicas, mediante la utilización de framework como *flask, jinja2 y Werkzeug*.

[2] Es el símbolo o mensaje que indica que el sistema está listo para recibir comandos o instrucciones del usuario.

- El entorno para el desarrollo de software que proporciona *Python*, lo convierte en uno de los lenguajes más apetecible por los desarrolladores actuales.

- La versatilidad y potencial de *Python*, no solamente está enmarcado en la inteligencia artificial, estadísticas o matemáticas; también es muy potente para el desarrollo de aplicaciones web a medida, por su manejo de dependencias, bibliotecas y framework que facilitan y reducen el trabajo.

- El framework *flask* permite al desarrollo de aplicaciones web escalables de mediano y gran tamaño.

Recomendaciones

- Se recomienda que el entorno de desarrollo para el desarrollo de proyectos de software, se realice desde el inicio con las últimas versiones estables disponibles de Python, Flask, boostrap y sus respectivas referencias de librerías.

CAPÍTULO 2

Establecimiento del contexto

2

Capítulo 2

Establecimiento del contexto

El desarrollo de sistemas basados en la web, ha sido uno de los temas más tratados a nivel mundial, la mayoría de los programadores buscan incansablemente llegar a controlar, manejar y comprender en su vida estudiantil y profesional alguna herramienta o **framework** para el desarrollo de este tipo de sistemas.

Las empresas buscan posicionar sus productos a través de las aplicaciones web ya que por su gran versatilidad en el internet permite llegar a todas partes y en todo momento, sus clientes requieren estar cada vez más conectados y así poder tener una comunicación más efectiva entre ellos.

Este libro tiene como objetivo fundamental involucrar al lector en el desarrollo de sistemas basados en la web mediante el entorno de desarrollo *Python* y *Flask*, es importante indicar que para que exista un adecuado aprendizaje el lector debe tener conceptos básicos en las

herramientas antes mencionadas, ya que el caso de estudio que se propone requiere de manejo de algoritmos, funciones, orientación a objetos, *JavaScripts*, estructuras de datos, bases de datos y estilos arquitectónicos.

En el contexto de la materia en cuestión, se plantea el desarrollo de una parte muy importante y necesaria para las aplicaciones web que por esencia todos la poseen. **El módulo de seguridad y manejo de usuarios, roles y permisos.**

El módulo de seguridad y manejo de usuarios, roles y permisos, es un componente que está diseñado para el manejo de sesiones de un usuario a un determinado sistema, este módulo es el encargado de administrar y permitir el acceso a los usuarios a determinadas funcionalidades con la que el usuario cuenta. Es decir, cada vez que un usuario requiere ingresar al sistema, deberá primero logearse o ingresar con sus credenciales (nombre de usuario [username] y contraseña [password]), el backend del sistema confirmará las credenciales con la información contenida en la base de datos y responderá (True/False) con el acceso permitido o no las funcionalidades con las que cuenta dicho usuario.

Es necesario entender algunos conceptos que serán relevantes más adelante, como es rol, permisos o funcionalidades.

¿Qué son los roles?

"Una definición de rol es una colección con nombre de tareas que definen las operaciones disponibles en un servidor de informes". (Sparkman, 2023)

Es cierto que cada autor tiene su propia perspectiva sobre la definición de roles. Asimismo, los roles desempeñan funciones y operaciones fundamentales dentro de un equipo de trabajo, contribuyendo tanto a la operatividad como a la estrategia de una empresa o proyecto.

Basado en lo anterior y para este contexto el rol, no es más que el papel que un usuario va a representar o desempeñar dentro del sistema.

¿Qué son los permisos?

"Los permisos definen el nivel de acceso que los usuarios y los grupos tienen respecto de un objeto". (Informática Documentation, 2024)

"Los permisos de usuarios especifican qué tareas pueden realizar los usuarios y a qué funciones pueden acceder". (Salesforce)

De la misma forma los autores coinciden que los permisos, son un conjunto de accesos a las tareas que se le asignan a un determinado rol o usuario, según su contexto para su gestión de permisos.

En este contexto, los permisos o accesos, son el consentimiento que se le entrega o asigna a un determinado rol para autorizar las acciones o funcionalidades que puede realizar dentro del sistema.

Es necesario indicar que estas definiciones son las más cercanas para comprender el contexto que se quiere abordar en el transcurso del libro, y son las que se utilizarán a medida que se continúe avanzando por las diferentes etapas de este libro.

2.1. Caso de estudio

El caso de estudio, pretende presentar una propuesta práctica para el desarrollo o construcción del módulo de seguridad y manejo de usuarios, roles y permisos; mismo que está planteado de manera que permita crear usuarios, roles y permisos (funcionalidades/acciones) desde una cuenta de administración para poder conceder funcionalidades específicas a determinados roles. Es decir, el módulo deberá permitir la creación de usuarios del sistema, roles y permisos; de la misma manera gestionará la asignación de roles a usuarios y permisos a roles.

Un rol puede estar asociado a uno o varios usuarios, y este rol puede incluir múltiples permisos dentro del entorno del módulo. A su vez, el sistema debe estar configurado para mostrar estos permisos o funcionalidades únicamente a los usuarios que accedan con el rol correspondiente.

Todos los sistemas informáticos están diseñados para que de una u otra manera contengan un módulo de gestión de seguridad de los usuarios, esto permite restringir el acceso de usuarios no deseados y fomentar la cultura de responsabilidad sobre la información que se genera en el sistema.

La propuesta está basada en una cultura de administración de usuario a base de roles y permisos o funcionalidades para que puedan realizar tareas dentro del sistema. Con esta alternativa se pretende automatizar la gestión de usuarios de tal manera que no se tenga que acudir al programador para crear un nuevo perfil desde el código fuente.

2.2. Estilo arquitectónico

Antes de hablar sobre los estilos arquitectónicos, tenemos que comprender algunos aspectos referentes a patrones de diseño, patrones arquitectónicos y luego llegar a entender los estilos arquitectónicos.

Según Blancarte (2020) existen dos tipos de patrones, los patrones de diseño y los patrones arquitectónicos. Los patrones de diseño se centran en como las clases y los objetos se crean, relacionan, estructuran y se comportan en tiempo de ejecución; mientras que los patrones arquitectónicos se centran en como los componentes se comportan, su relación con los demás y la comunicación que existe entre ellos, pero también abordar ciertas restricciones tecnológicas, hardware, performance, seguridad, etc. (p. 32).

2.2.1. Patrones de diseño

Los patrones de diseño no son soluciones específicas de código, sino directrices generales que se puede aplicar al diseño y estructura de una clase u objeto.

Según Blancarte (2020) existen tres tipos de patrones de diseño, que se agrupan según el problema que se desea resolver (p. 35):

- **Patrones Creacionales**: estos patrones de diseño que se centran en la creación o construcción de objetos. Ayudan a garantizar que los objetos se creen de manera flexible, eficiente y controlada. Entre este tipo de patrones tenemos: Abstract Factory (Permite producir familias de objetos relacionados sin especificar sus clases concretas); Builder (Permite construir

objetos complejos paso a paso. Este patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción); Factory Method (Proporciona una interfaz para la creación de objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán); Prototype (Permite copiar objetos existentes sin que el código dependa de sus clases); Singleton (Permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia), (GURU, 2023).

- **Patrones Estructurales**: estos patrones se ocupan de la forma como se relaciona una clase determinada con otras clases. Ayudan a definir el orden de las clases; entre este tipo de patrones tenemos: Adapter (Permite la colaboración entre objetos con interfaces incompatibles); Bridge (Permite dividir una clase grande o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra); Composite (Permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales); Decorator (Permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades); Facade (Proporciona una

interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases); Flyweight (Permite mantener más objetos dentro de la cantidad disponible de memoria RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto); Proxy (Permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndote hacer algo antes o después de que la solicitud llegue al objeto original), (GURU, 2023).

- **Patrones de Comportamiento**: estos patrones se centran en la interacción y comunicación de los objetos entre sí. Ayudan a definir el comportamiento y las responsabilidades de los objetos en una aplicación. Entre este tipo de patrones tenemos: Iterator (Permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena); Command (Convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación te permite parametrizar los métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y soportar operaciones que no se pueden realizar); Mediator (Permite reducir las dependencias caóticas entre objetos. El patrón restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un

objeto mediador); Observer (Permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando); State (Permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiara su clase); Visitor (Permite separar algoritmos de los objetos sobre los que operan); Template Method (Define el esqueleto de un algoritmo en la superclase, pero permite que las subclases sobrescriban pasos del algoritmo sin cambiar su estructura); Memento (Permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación), (GURU, 2023).

2.2.2. Patrones arquitectónicos

> *"Un patrón arquitectónico es una colección de decisiones de diseño arquitectónico que son aplicables a problemas de diseño recurrentes, y que están parametrizados para tener en cuenta los diferentes contextos de desarrollo de software en el que surge el problema". (Estevez, 2019)*

Los patrones arquitectónicos se centran en que los componentes o módulos puedan relacionarse y comunicarse entre ellos, pero también abordar ciertas restricciones tecnológicas, hardware, performance, seguridad, etc.

En un desarrollo de software existe una gran cantidad de componentes que forman parte del sistema. En el proyecto siempre se va a encontrar con componentes específicos (bancarios, de cobros y pagos, asignación de calificaciones, bodega e inventarios), así como componentes reusables en un entorno dado (servicios web, servidores web, sockets, interfaces), y componentes reusables en general (librerías, bibliotecas de componentes GUI, entre otras).

Los patrones arquitectónicos también son conocidos como “arquetipos”, son fundamentales al momento de expresar un esquema organizacional para un proyecto de software. La mayoría de los proyectos software están planificados para contener un conjunto de elementos que ayudan a construir un sistema informático que cuente con funcionalidades, responsabilidades, conexiones e integraciones; es ahí donde el patrón arquitectónico captura su esencia para construir una arquitectura de software eficiente.

Algo que es importante tener en cuenta es no confundir patrón arquitectónico con arquitectura de software (estilo arquitectónico), ya que el patrón arquitectónico es parte de la arquitectura del software.

Existe una amplia variedad de patrones arquitectónicos en el mundo del desarrollo de software, por ejemplo:

1. Patrón de capas
2. Patrón cliente-servidor
3. Patrón maestro-esclavo
4. Patrón de filtro de tubería
5. Patrón de intermediario
6. Patrón de igual a igual
7. Patrón de bus de evento
8. Modelo-Vista-Controlador (*MVC*)
9. Patrón de pizarra
10. Patrón de intérprete

En este libro vamos a enfocarnos en entender el patrón arquitectónico Modelo-Vista- Controlador (*MVC*) ya que es el que vamos a implementar en nuestro caso de estudio.

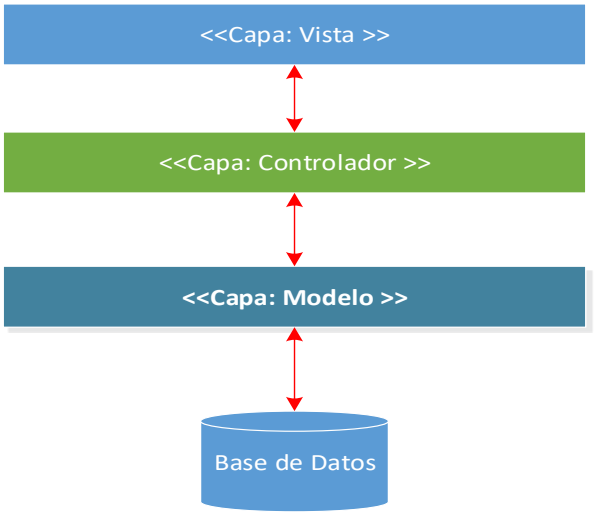
Patrón arquitectónico Modelo-Vista-Controlador (MVC)

Es uno de los más utilizado por los desarrolladores para la implementación de proyectos de software, su aplicación es atribuida al desconocimiento de los otros tipos de patrones arquitectónicos, pero en realidad no tiene que ver con eso, sino a las múltiples características que ofrece dicho patrón.

El patrón es versátil al momento de proporcionar servicios a la capa inmediatamente superior, esto provoca un nivel concreto de abstracción de las acciones, tareas o funcionalidades de cada capa.

Figura 1

Patrón arquitectónico MVC.



Nota. Elaborado por los autores.

Cada capa es un elemento independiente dentro del sistema, esto permite que tanto el desarrollo como el despliegue sean independiente, de esta forma se puede pensar que durante su implantación o puesta en producción, estos componentes puedan estar alojados en servidores de diferentes tecnologías y ubicaciones.

La separación de la aplicación en capas busca cumplir con el principio de separación de preocupaciones (SoC)[3], de tal forma que cada capa se encargue de una tarea; por ejemplo, la capa *Vista* sólo se preocupa por presentar la información de forma agradable al usuario, pero no le interesa de donde viene la información ni la lógica de negocio que hay detrás, en su lugar, solo sabe que existe una capa *Modelo* que le proporcionará la información.

Por otra parte, la capa *Controlador* se encarga de aplicar todas las reglas de negocio y validaciones, pero no se preocupa de recuperar los datos, guardarlos o borrarlos, ya que de eso se encarga capa *Modelo*, encargada de comunicarse con la base de datos, crear las instrucciones SQL para consultar, insertar, actualizar o borrar.

De esta forma, cada capa se preocupa por una cosa y no le interesa como le haga la capa de abajo para servirle los datos que requiere (Blancarte, 2021).

La capa de *Vista* es la encargada de generar las vistas para los usuarios, la capa *Controlador* permitirá las conexiones necesarias para conseguir datos ingresados por los usuarios y enviar datos a la capa *Modelo* realizará las conexiones a la base de datos, es decir, ejecutar las sentencias SQL. En consecuencia, cada capa tiene su funcionalidad

[3] Concepto de ingenería de software que propone dividir al software en secciones independientes.

definida y debe respetarse esa independencia, ya que irrespetar esa regla sería ir en contra de la esencia misma del patrón *MVC*.

El patrón *MVC*, cuenta con ventajas y desventajas en su aplicación, a continuación, se presentan algunas de ellas:

Ventajas

- Permite la separación de responsabilidad, ya que cada capa tiene la suya.
- Es fácil de implementar, además de que es muy conocido y una gran mayoría de las aplicaciones la utilizan.
- Su versatilidad para ir probando las capas por separado.
- Es fácil detectar el origen de un bug (errores) para corregirlo, o simplemente se puede identificar donde se debe aplicar un cambio.
- Permite el aislamiento de los servidores en subredes diferentes, lo que hace más difícil realizar ataques.

Desventajas

- La comunicación por la intranet o internet es lenta, incluso, en muchas ocasiones más tardado que el mismo procesamiento de los datos.

- Las aplicaciones que implementan este patrón se tornan monolíticas y provoca que sean difíciles de escalar.
- Crea una dependencia en el despliegue de la aplicación.
- Si una capa falla, todas las capas superiores comienzan a fallar en cascada.

La variedad de los patrones es muy extensa, pero nos enfocaremos en el *MVC* (Modelo Vista Controlador) ya que es con el que se va a implementar el caso de estudio.

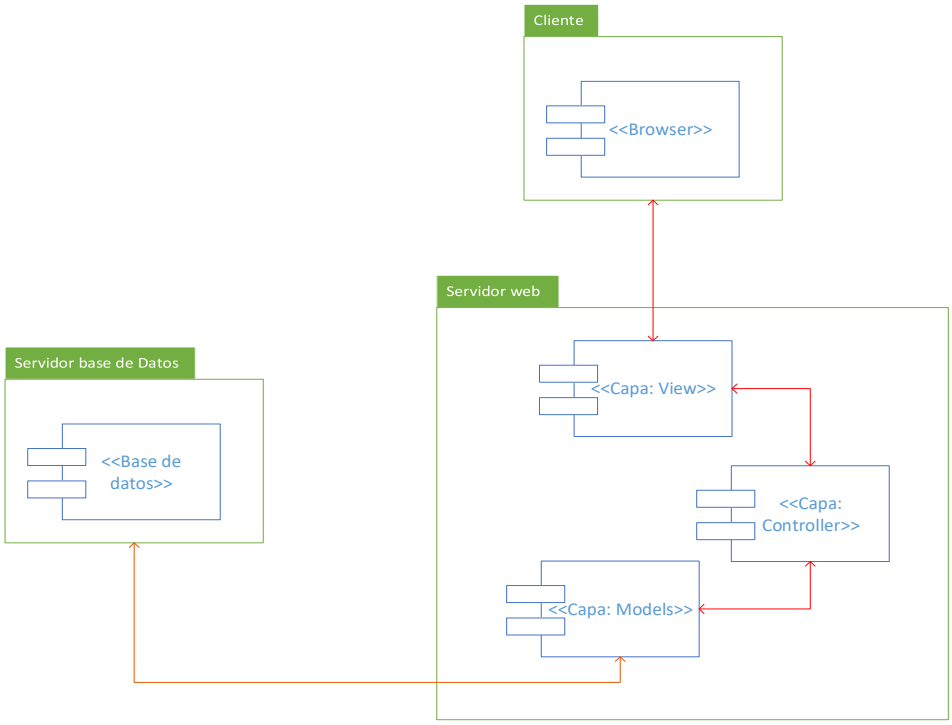
El *MVC* es un patrón arquitectónico que separa la vista, la lógica de control y los datos en 3 distintas capas.

- Modelo: contiene la información de los datos. Es el encargado de comunicarse con la base de datos para realizar actualizaciones, registros, búsquedas y eliminaciones, mediante mecanismos para acceder a la información por lo general estos mecanismos pueden ser sentencias sql, no-sql, entre otros.
- Vista: es la interfaz de usuario, es decir, con lo que interactúa el usuario. Por lo general son las visualizaciones de interfaces que pueden estar codificadas con *HTML, HTML5, XML* que nos permiten mostrar la salida.

- Controlador: es la conexión entre el modelo y la vista. Cuando se produce un cambio, informa a la vista o al modelo de éstos. Su responsabilidad no es manipular directamente los datos, ni mostrar ningún tipo de salida, sino servir de enlace entre los modelos y las vistas para implementar las diversas necesidades del desarrollo.

Figura 2

Patrón arquitectónico MVC.



Nota. Elaborado por los autores.

Estilo arquitectónico

Muchos arquitectos y desarrolladores de software comenten el error de confundir los patrones arquitectónicos con los estilos arquitectónicos, esto genera un gran problema a la hora de aplicar el estilo en el proyecto. Los estilos arquitectónicos son artefactos que determinan las características que deben tener un componente, lo cual lo hace reconocible en un proyecto de desarrollo de software.

El estilo arquitectónico es un marco de referencia a partir del cual es posible construir aplicaciones que comparten un conjunto de atributos y características mediante el cual es posible identificarlos y clasificarlos (Blancarte, 2020).

Por otra parte, los estilos arquitectónicos permiten a los desarrolladores percibir ciertos lineamientos y características que deben existir en un proyecto de software, para considerar que siguen un determinado estilo. En el ámbito de desarrollo existen una gran cantidad de estilos como son: *Monolíticos*, en capas, *peer to peer*, *microservicos*, *SOA* (Service-Oriented Architecture), *EDA* (Event Driven Architecture), *microkernel, REST* (Representatinal State Transfer).

2.3. Diseño de mockups del prototipo

Los *mockups*, permiten bosquejar diseños con respecto a las funciones básicas, la navegación, la arquitectura de contenidos y el diseño. Para entender mejor este concepto, es necesario citar algunas definiciones de expertos en el tema.

> *"Un mockup es la representación del prototipo del proyecto que se va a realizar. En este caso se trata de una imagen que nos muestra el resultado visual que tendrá nuestra página web o nuestra maqueta". (Prida, 2020)*

> *Los mockups, que son una excelente manera de obtener comentarios sobre una idea de diseño antes de invertir tiempo y recursos en el producto final, son una obligación para cualquier diseñador. También son una buena forma de enseñar a los clientes cómo se verá el resultado final. (Pérez, 2022)*

> *"Un mock-up es un prototipo hecho antes del desarrollo del trabajo. Sirve para transformar ideas en funcionalidades y ayuda al cliente a exteriorizar y comprender lo que necesita". (Pagés, 2018)*

En base a lo expuesto, los criterios citados coinciden en que los mockups son herramientas que permiten bosquejar un conjunto de acciones, funcionalidades y así representar un prototipo del punto de vista del cliente.

Estos bosquejos o diseños se utilizan como punto de partida para el desarrollo de aplicaciones, es una herramienta eficaz para coordinar con los clientes las ideas y requisitos que se necesitan implementar en el proyecto.

Respecto al caso de estudio, el prototipo que se propone para el ejercicio de aprendizaje es el siguiente:

2.3.1. Acceso a usuario

El acceso a los usuarios es la primera funcionalidad con la que cuente el sistema que se desarrolla, es la puerta de entrada de los usuarios al sistema. Además, es un medio de control y seguridad para usuarios no autorizados.

El caso de estudio, necesita que exista una funcionalidad donde los usuarios puedan ingresar sus credenciales (nombre de usuario y contraseña), y luego de una comprobación y validación se permita o no el acceso a los recursos o bondades que ofrece el sistema informático.

Figura 3

Mockup Acceso al sistema.

Nota: Elaborado por los autores

El prototipo, permitirá al usuario ingresar el nombre de usuario y la contraseña para ser validada y verificada al momento de pulsar el evento botón "**Ingresar**". Luego, el módulo deberá consultar con la base de datos las credenciales y retornar un verdadero [True] o falso [False] al usuario, permitiendo o denegando el acceso.

En caso que la respuesta sea Falsa [False], el sistema responderá con el mensaje "Usuario no encontrado ....."; en el caso que el nombre de usuario no exista, y con el mensaje "Password incorrecto .....", en caso que la contraseña es incorrecta.

2.3.2. Gestión de listas de usuarios del sistema

La lista de usuarios del sistema es el conjunto de clientes que cuentan con credenciales disponible en el sistema para poder usar los recursos, funcionalidades y acciones. A continuación, se muestra el mockup prototipo para esta funcionalidad.

Figura 4

Mockup Gestión de usuarios del sistema.

Nota. Elaborado por los autores

Este prototipo está pensado para tener una lista de usuarios que interactúan con el sistema, esta gestión de usuarios permitirá tener acceso al sistema, por ello es importante tener en cuenta al usuario con un único nombre de usuario de tal manera que no se dupliquen. Cada vez que el rol administrador ingrese al sistema, se debe mostrar la lista de los usuarios que actualmente tienen cuenta disponible. Además, debe tener un evento funcional tipo botón denominado "**Nuevo**", para registrar nuevas cuentas de usuarios.

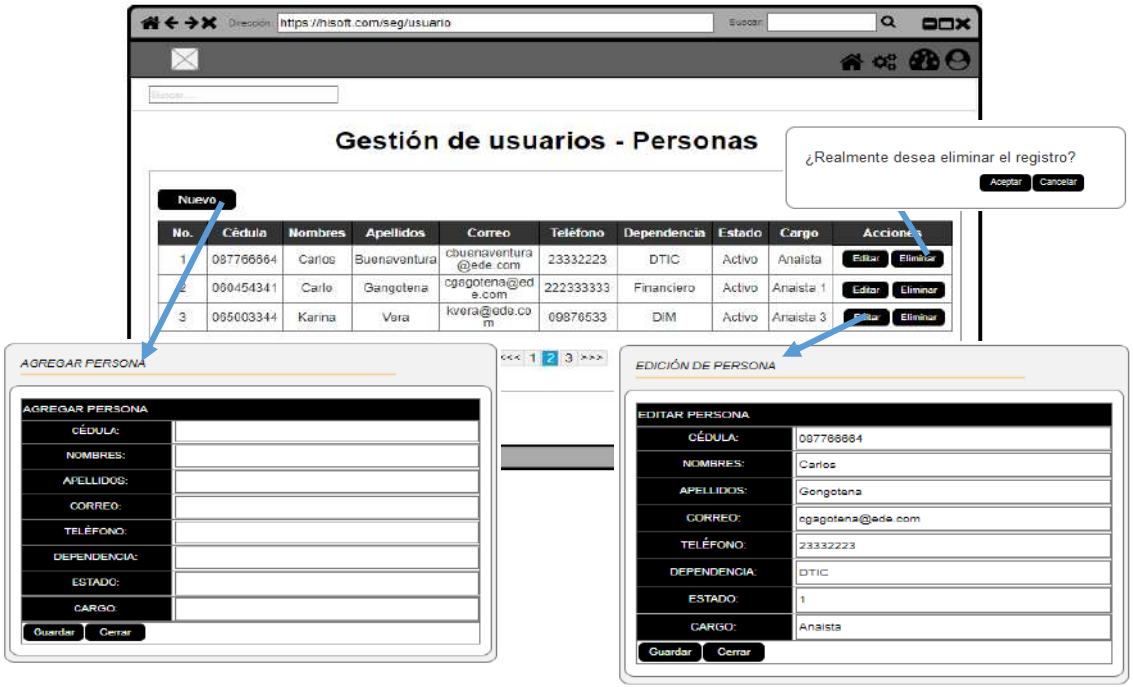
De la misma forma cada registro, contará con dos eventos "**Editar**" y "**Eliminar**". El botón "**Editar**" permitirá realizar actualizaciones de datos, en este caso sólo podrá actualizar la contraseña del usuario. Por otra parte, el botón "**Eliminar**", borrará de la base de datos el registro actual.

2.3.3. Gestión de datos de personas o clientes del sistema

Es un espacio de trabajo para gestionar el registro de datos de las personas que en su momento podrían tener un rol y permisos al sistema en construcción.

Figura 5

Mockup Gestión de personas.



Nota: Elaborado por los autores

Para este entorno, se deberá contener un evento funcional tipo botón con la denominación de "**Nuevo**", destinado al registro de nuevos usuarios al sistema.

Por cada registro de usuarios el sistema contendrá dos eventos tipo botón "**Editar**" y "**Eliminar**". El botón "**Editar**" permitirá realizar actualizaciones de datos. Por otra parte, el evento botón "**Eliminar**", borrará de la base de datos el registro actual, antes deberá preguntar si desea realizar dicha acción sobre el registro seleccionado.

Para este entorno se requiere las siguientes reglas del negocio:

1. No deben repetirse un cliente o persona.
2. Deben de estar controlados por un argumento único.

2.3.4. Gestión de roles

Es una de las funcionalidades importantes para tener en un sistema, permite controlar y monitorear el comportamiento de los usuarios, logra, a través de la definición de reglas y restricciones que limiten las acciones de los usuarios en el sistema. Los usuarios gracias sus roles pueden tener un conjunto de privilegios para trabajar en los sistemas informático.

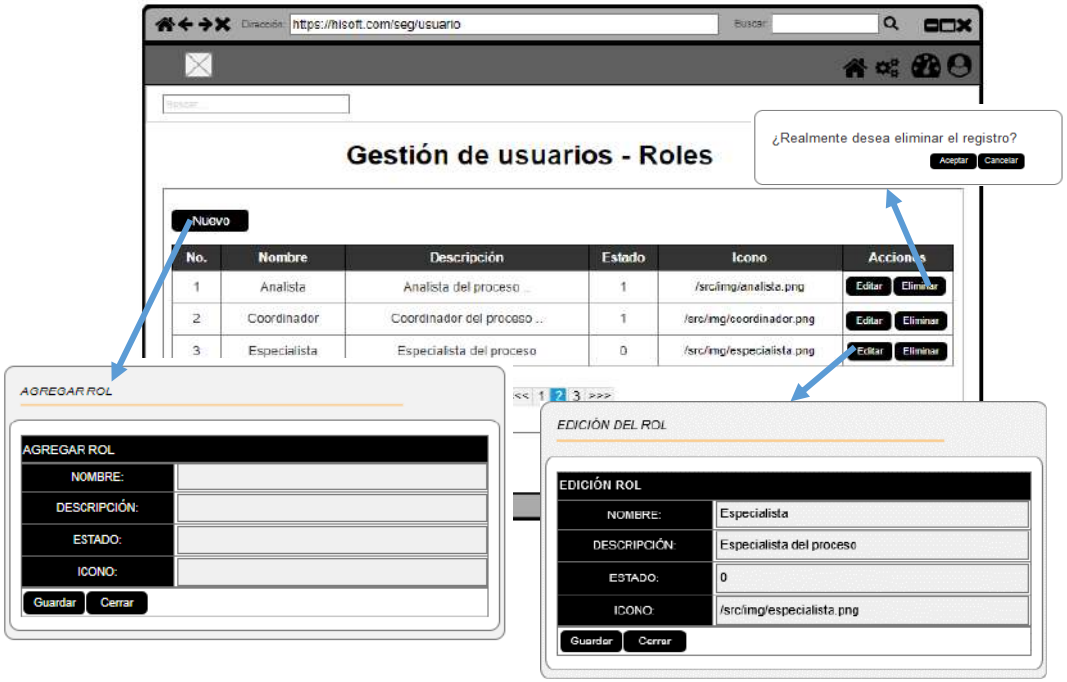
La gestión de roles para este caso de estudio, esta propuesta para que los usuarios del sistema puedan tener uno o varios perfiles que les permita tener acceso a deferentes funciones del sistema.

De la misma manera, cada vez que un usuario ingrese a este entorno el sistema presentará la lista actualizada de los roles registrados, luego cada registro contará con los eventos funcionales tipo botón "**Nuevo**", "**Editar**" y "**Eliminar**". Para este entorno se requiere las siguientes reglas del negocio:

1. No se deben repetir los nombres de los roles.
2. No podemos eliminar un rol si este ya fue asignado a un usuario del sistema.
3. Aquel rol que esta con estado "0" no puede ser tomado en cuenta para la asignación a un usuario.

Figura 6

Mockup Gestión de roles.



Nota. Elaborado por los autores.

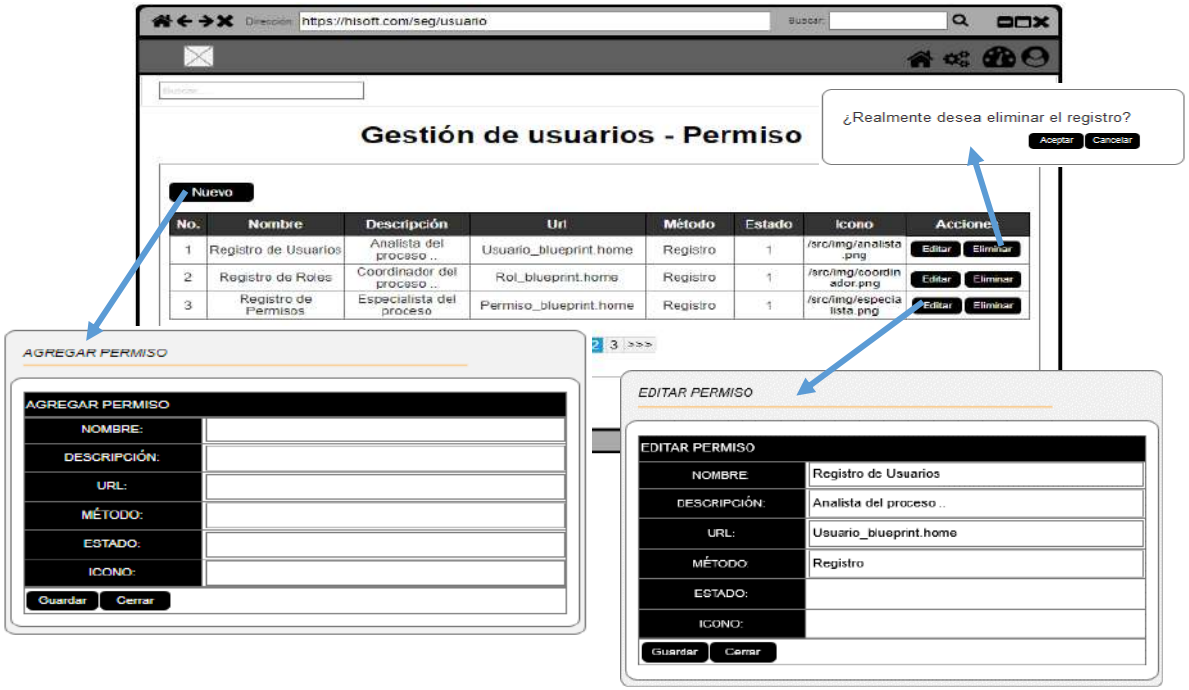
2.3.5. Gestión de permisos

La seguridad del usuario se basa en privilegios y permisos. Los permisos definen el nivel de acceso que los roles tienen respecto a la funcionalidad del sistema. Los permisos en el contexto del caso de estudio serán los privilegios que puede tener un rol dentro del ámbito del sistema.

Esta funcionalidad garantizará que los usuarios tengan los privilegios que necesitan para cumplir con su trabajo y controles que necesitan, de este modo se garantiza la seguridad y la funcionalidad del sistema o módulo. Siguiendo con el estándar establecido, cada vez que se ingrese a este entorno, el sistema presentará la lista actualizada de los roles existentes, de igual manera, se mantiene la misma idea de los eventos funcionales tipo botón "**Nuevo**", "**Editar**" y "**Eliminar**".

Figura 7

Mockup Gestión de roles.



Nota. Elaborado por los autores.

2.3.6. Gestión de asignación de roles a usuarios

Es el momento donde un usuario forma parte de un perfil dentro del sistema, cada vez que un rol es asignado a un usuario, este asume todas las características de este rol; por lo general, cuando se asigna un rol este ya cuenta con sus respectivos permisos o privilegios. Este prototipo, tiene como objetivo asignar a un usuario, uno o varios roles; es decir, a los usuarios que cuenten con una cuenta en el sistema, se le podrá vincular con uno o varios roles del sistema.

Para este entorno se requiere las siguientes reglas del negocio:

1. No se debe repetir el registro de un usuario con un determinado rol, Un usuario puede tener varios roles.
2. Los usuarios deberán ser identificados por su nombre de usuario que va a ser un código único. Por ejemplo: “carlos.morales”
3. Un rol puede estar asignado a varios usuarios.
4. Aquel rol que esta con estado “0” no puede ser tomado en cuenta para la asignación a un usuario.
5. Un permiso que se encuentra con estado “0” no puede ser tomado en cuenta.
6. No podemos eliminar una asignación de rol si este ya fue asignado a un usuario del sistema.
7. En este entorno no se requiere edición de información.

Figura 8

Mockup Gestión de roles a usuarios.

Nota. Elaborado por los autores.

2.3.7. Gestión de asignación de permisos a roles

La asignación de permisos o privilegios a roles está orientada para entregar ciertas funcionalidades a roles, esto permitirá que un usuario con un determinado rol pueda acceder al sistema con sus privilegios.

Para este caso de estudio los permisos serán las operaciones para la gestión de datos en una base de datos, también conocidas como CRUD (Create, Read, Update, Delete).

En ese contexto, un permiso o privilegio se identificará como "Gestión de usuarios", con este permiso el rol podrá hacer acciones como mostrar, registrar, editar y eliminar de un determinado usuario en los datos en la tabla correspondiente en la base de datos.

Es importante indicar que la asignación de permisos a roles se realiza de esta manera por motivos académicos y no corresponden a un estándar.

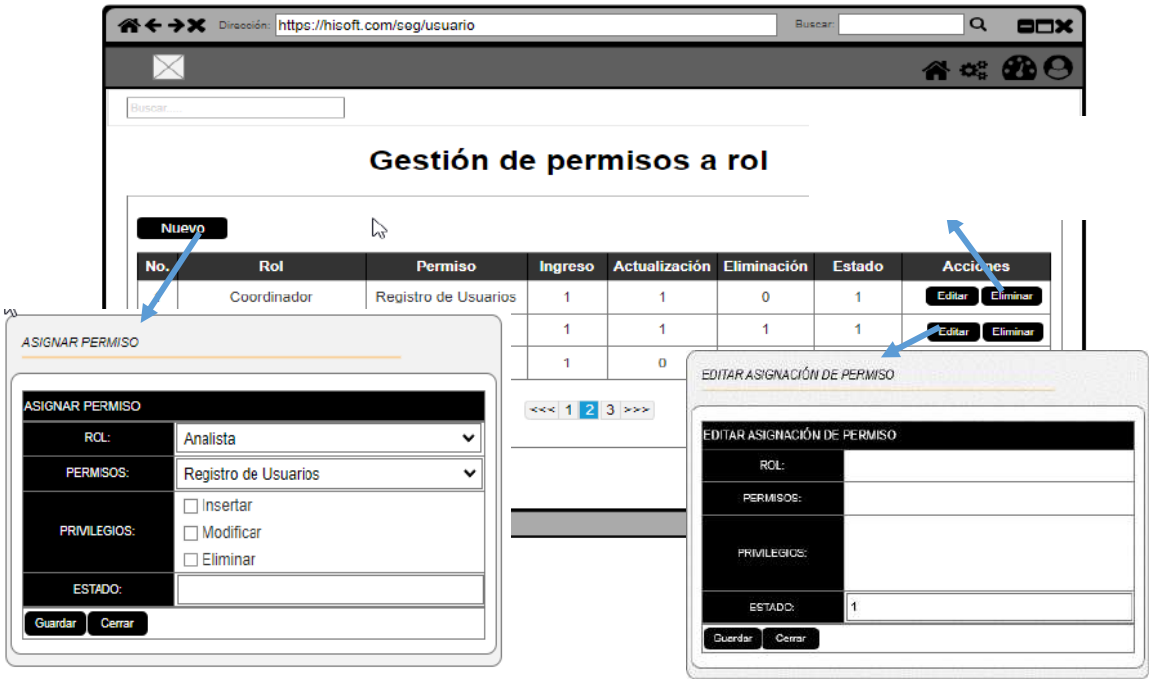
Para este entorno se requiere las siguientes reglas del negocio:

1. No puede registrar duplicidad de asignación de roles con permisos.

2. La asignación de privilegios será configurada aquí y gestionada en cada entorno de trabajo del sistema.
3. No podrá eliminarse una asignación de permiso que está siendo usada por un rol.
4. La edición está orientada a la actualización de permisos a un rol, más no, a crear permisos o roles.

Figura 9

Mockup Gestión de permisos a roles.



Nota. Elaborado por los autores.

2.4. Diseño de base de datos

En este punto, se cuenta con una gran cantidad de información que nos permitirá establecer un modelo de datos que apalancará el desarrollo del módulo de seguridad y manejo de usuarios, roles y permisos.

Cada *mockup* diseñado anteriormente permite tener una idea de cómo podríamos diseñar nuestro modelo de datos al punto de establecer una base de datos que sirva para satisfacer el problema planteado en nuestro caso de estudio.

En respuesta del análisis y diseño de los esquemas (Mockup) anteriores, se han establecido que el módulo de seguridad y manejo de usuarios, roles y permisos, debe contar con 6 tablas dispuestas de la siguiente manera:

Tabla: Persona

Esta tabla está diseñada para almacenar los datos de todos los clientes (persona), que pueden tener algún tipo de intervención en nuestro sistema, estos datos son utilizado si en algún futuro queramos establecer comunicación o acciones con ellos. Por ejemplo: envío de correos, notificaciones de ventas, cobros, creación de cuentas, manejo de estadísticas, entre otras cosas.

En nuestro caso de estudio, nos servirá para escoger a los posibles usuarios del sistema.

Tabla: Usuario

La tabla de usuarios es un esquema donde se almacena la información de las cuentas de usuarios que pueden acceder a un sistema o aplicación mediante sus credenciales.

En nuestro caso de estudio, permitirá registrar las cuentas de usuarios que tendrán acceso al sistema, sólo aquellas personas o clientes que estén en esta tabla podrán acceder a los recursos, funciones y acciones del sistema.

Tabla: Rol

Es un catálogo de descripción de roles para el sistema. Esta tabla contendrá el registro de todos los roles que pueden ser utilizados para asignar a un determinado usuario. Esta tabla es fundamental para gestionar los permisos y niveles de acceso al sistema.

Tabla: Permiso

Esta tabla recibirá el registro de los permisos del sistema, la información contenida en esta tabla es de vital importancia para el acceso al sistema mediante un rol. Es necesario establecer que aparte de la identificación del permiso o privilegio, se requiere tener una ruta o url que indique el "EndPoint" de la funcionalidad a la que pertenece el permiso.

Tabla: Personarol

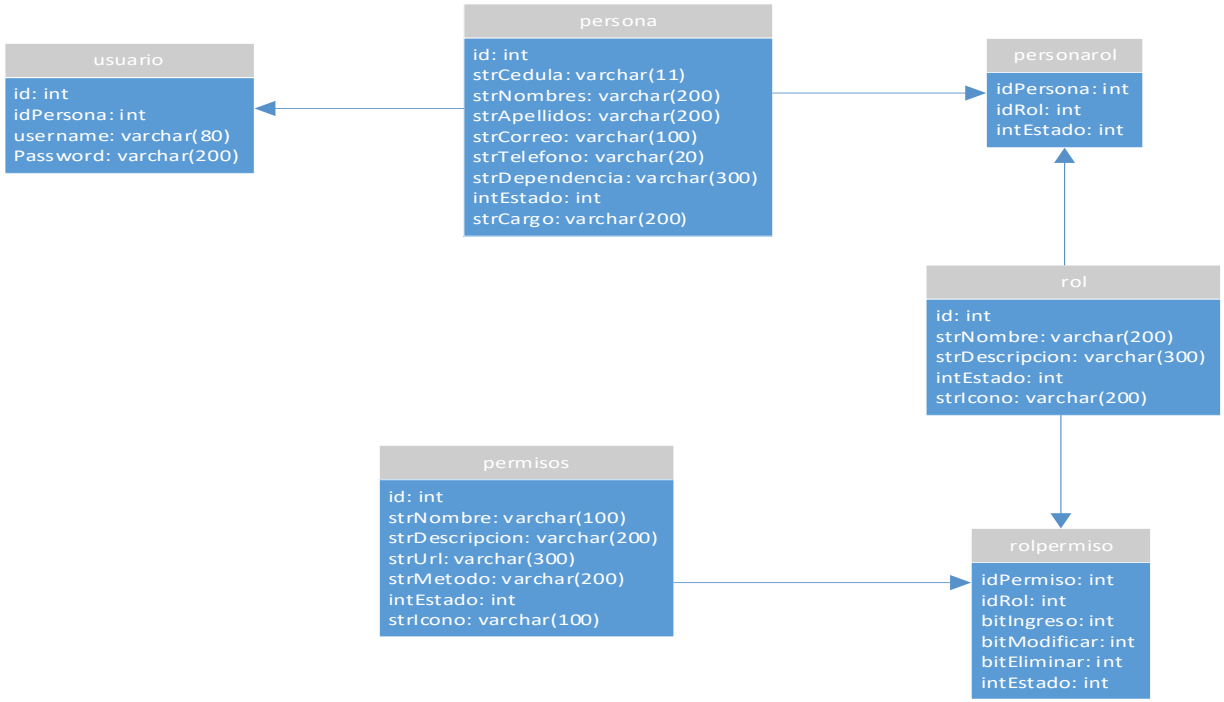
Es una colección de información en forma de tabla que permite el registro de la asignación de un rol a uno o varios usuarios, en este punto el sistema sabrá qué rol va a tener un usuario en el sistema.

Tabla: Rolpermiso

Esta tabla se enfoca en asignar el o los permisos a rol, con ello el sistema podrá presentar a los usuarios la lista de las páginas o ventanas a las que tiene acceso.

Figura 10

Modelo de la base de datos.



Nota. Elaborado por los autores.

El diagrama de la base de datos, que, por motivos netamente académicos, y del caso de estudio le llamaremos ***"seguridad_db"*** nos permite tener una visión de la organización de los datos en la base de datos.

De esta manera podemos observar que un usuario puede contar con más de un rol y puedan tener más de un permisos o privilegio. Es necesario indicar que este modelo es desarrollado con fines de aprendizaje y no representa una solución general para todos los proyectos de software, en su defecto nos ayudará a entender el desarrollo de software mediante *Python*, *Flask* y el patrón arquitectónico *MVC*.

Conclusiones

- El diseño de los mockups o esquemas, es una herramienta muy efectiva para establecer prototipado de la solución informática que se va a desarrollar o implementar, ya que permite establecer una idea general de cómo debería empezar a implementar la solución.
- Los mockups, permiten tener una visión de cómo fluyen los datos dentro del sistema y con ellos se tiene lineamientos claros para el diseño de una base de datos más eficiente y con menos errores.

- Los patrones de diseño se centran en cómo las clases y los objetos se crean, relacionan, estructura y se comportan en tiempo de ejecución. Mientras que los patrones arquitectónicos se centran en como los componentes se compartan, su relación con los demás y la comunicación que existe entre ellos, pero también abordar ciertas restricciones tecnológicas, hardware, performance, seguridad, etc.
- El patrón arquitectónico *MVC*, es independiente de la tecnología de desarrollo que se utilice para implementar el proyecto software, además gracias a su separación de componentes en capas permite mantener una estructura organizada, limpia y con un acoplamiento mínimo entre las distintas capas.

Recomendaciones

- Como ejercicio de entrenamiento se recomienda mejorar el diseño de la base de datos, de tal manera que los permisos sean más específicos; por ejemplo, que un rol asignado a un usuario pueda eliminar un registro, pero que no tenga permiso para crear o agregar un nuevo registro en un mismo entorno.

CAPÍTULO 3

Establecimiento del entorno de trabajo

3

Capítulo

3

Establecimiento del entorno de trabajo

Hasta este punto, se ha logrado establecer aspectos importantes a tomar en cuenta para el desarrollo de un módulo web de seguridad y manejo de usuarios, roles y permisos. Entre la identificación del caso de estudio, el aprendizaje de las herramientas y metodologías, lo que resta es configurar el entorno de trabajo donde se implementará todo lo aprendido hasta el momento.

Este módulo será creado mediante el lenguaje de programación *Python*, utilizando el framework *flask* y *Jinja2* para los servicios en la web, así como el framework *Bootstrap* para el diseño de páginas web, además de un conjunto de librerías de *Python* y el motor de base de datos *MySQL*. Por último, para la codificación vamos a utilizar el entorno de desarrollo *Visual Studio Code*.

3.1 Visual Studio Code

El desarrollo de proyectos informáticos ha evolucionado, continuamente los editores de código están adaptados para no limitar la creatividad e innovación de los desarrolladores de software; esto se debe en gran medida, a que están diseñados para soportar la tecnología de una gran cantidad de lenguajes de programación.

Visual Studio Code, es un editor de código fuente multiplataforma y con licencia *GNU*, tiene una gran comunidad que mantiene actualizado constantemente el soporte para la depuración de código, extensiones y librerías; permitiendo de esta manera las posibilidades de escribir y ejecutar una gran cantidad de lenguaje de programación.

Según Flores (2021), *Visual Studio Code* tiene una gran variedad de características útiles para agilizar el trabajo, que lo hacen el editor preferido por muchos (me incluyo) para trabajar los proyectos (p. 5):

- Multiplataforma: es una característica importante en cualquier aplicación y más si trata de desarrollo. Visual Studio Code está disponible para *Windows*, *GNU/Linux* y *macOS*.
- IntelliSense: esta característica está relacionada con la edición de código, autocompletado y resaltado de sintaxis, lo que permite ser más ágil a la hora de escribir código. Como su

nombre lo indica, proporciona sugerencias de código y terminaciones inteligentes en base a los tipos de variables, funciones, etc. Con la ayuda de extensiones se puede personalizar y conseguir un IntelliSense más completo para cualquier lenguaje.

- Depuración: *Visual Studio Code* incluye la función de depuración que ayuda a detectar errores en el código. De esta manera, nos evitamos tener que revisar línea por línea a puro ojo humano para encontrar errores. VS Code también es capaz de detectar pequeños errores de forma automática antes de ejecutar el código o la depuración como tal.
- Uso del control de versiones: *Visual Studio Code* tiene compatibilidad con *Git*, por lo que puedes revisar diferencias o lo que conocemos con git diff, organizar archivos, realizar commits desde el editor, y hacer push y pull desde cualquier servicio de gestión de código fuente (*SMC*). Los demás *SMC* están disponible por medio de extensiones.
- Extensiones: hasta ahora, he mencionado varias veces el término extensiones porque es uno de los puntos fuertes. *Visual Studio Code* es un editor potente y en gran parte por las extensiones. Las extensiones nos permiten personalizar y agregar funcionalidad adicional de forma modular y aislada. Por ejemplo, para programar en diferentes lenguajes, agregar nuevos temas al editor, y conectar con otros servicios.

Realmente las extensiones nos permiten tener una mejor experiencia, y lo más importante, no afectan en el rendimiento del editor, ya que se ejecutan en procesos independientes.

Al momento de instalar el editor de código incluye un mínimo de componentes y funciones básicas de un editor con soporte nativo para *JavaScript/TypeScript* y *Node.js*, aunque también es posible instalar ciertos plugins y extensiones para diferentes tipos de lenguajes.

3.1.1 Instalación de Visual Studio Code

Para comenzar a usar el editor de código *Visual Studio Code*, debemos establecer algunas premisas importantes para instalación. Es necesario indicar que estas premisas están descritas pensando que el sistema operativo huésped es Microsoft *Windows*.

a) Asegurarse que su sistema operativo sea ARM64 o x64,
b) Al menos debemos contar con 4 GB de memoria RAM,
c) El espacio en disco duro debe al menos contar con 850 MB, para el paquete básico, pero si la intención es tener el editor completo se va a requerir de entre 20 a 50 GB de almacenamiento,
d) Tarjeta de vídeo que admita una resolución de pantalla mínima de WXGA (1366 x 768).

Con las sugerencias antes mencionadas, el entorno de trabajo para la instalación está listo, por ende, el procedimiento para la instalación es el siguiente:

Paso 1.

Lo primero que hay que hacer es dirigirse a la página oficial de *Visual Studio Code* (*https://code.visualstudio.com/*) y descargar el paquete de instalación que provee la última versión estable del editor.

Paso 2.

Por lo general el paquete se alojará en la carpeta de descargas del Sistema Operativo, luego de identificar el paquete es importante ejecutarlo en modo Administrador. Esto permitirá ejecutar el setup del instalador de *Visual Studio Code*, con los permisos suficientes para efectuar esa operación sin problemas.

Paso 3.

Cuando el Wizard de instalación presente los términos y condiciones para continuar con la instalación, es necesario leer y en caso de estar de acuerdo aceptar dichos términos. Sólo así podrá ejecutarse la instalación del editor de código; en caso de no estar de

acuerdo con los términos y condiciones, el procedimiento de instalación se cancelará.

Paso 4.

Es importante tener en cuenta el directorio de instalación del editor de código, en caso de que desee cambiar la ruta más adelante. Además, es importante seleccionar la viñeta “Agregar a PATH”, esta acción permitirá agregar el PATH en las variables de entorno del sistema operativo.

Paso 5.

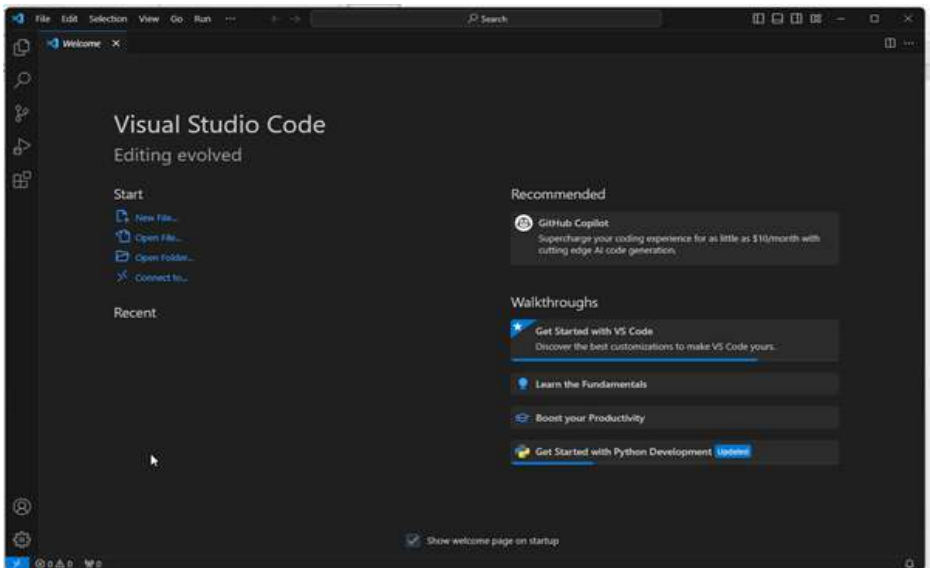
En caso de requerir crear un icono en el escritorio o añadir opciones al menú contextual de Windows Explorer. Será necesario seleccionar dichas viñetas.

Paso 6.

Por último, se requerirá de pulsar el botón “Instalar”, esta acción provocará la instalación del editor de código en el computador.

Figura 11

Ventana principal de Visual Studio Code.



3.2 Librerías Python

Las librerías son un conjunto de paquetes con funcionalidades específicas que permiten a los programadores llevar a cabo tareas, actividades o funcionalidades que no pueden resolver desde su implementación original.

La mayoría de las librerías de *Python* están destinadas a resolver un evento especifico o responden al conjunto de implementaciones listas para ejecutarse en el código que las invocan, con el objeto de crear una interfaz independiente.

En lo que respecta a las librerías de *Python* existen las siguientes clasificaciones:

- **Deep learning:** es para la predicción de datos.
- **Machine learning:** es utilizada para el aprendizaje de máquina, procesamiento de información, resolución de problemas de clasificación y el análisis de regresión de datos.
- **Cálculo numérico:** gestiona los datos para su cálculo.
- **Visualización:** sirven para entender y comprender los datos, de una forma más legible.
- **Inteligencia Artificial:** optimiza datos para la Inteligencia Artificial.
- **Procesamiento de lenguaje natural:** sirve para cálculo de frecuencias normalizadas, se construyen los modelos con datos de texto, optimización de configuraciones.

Para instalar una librería de *Python* y utilizarlo en el proyecto de software, se requiere utilizar el comando *"pip"*.

3.2.1 Python-decouple

La librería para el desacoplamiento de *Python* es de tipo **Procesamiento de lenguaje natural**, es utilizada para organizar las configuraciones y variables de entorno que se requiere en la

implementación de un proyecto de software, permite separar el código fuente de credenciales y/o rutas de bases de datos.

Esta librería será utilizada en el proyecto de caso de estudio para definir valores predeterminados en el archivo "*.env*"; además, convertirá correctamente los valores de tipos de datos.

La importancia de esta librería radica en tener un módulo de configuraciones generales y centralizadas para controlar todas las instancias.

Con esta librería podremos controlar parámetros generales como: argumentos de conexión para las bases de datos, claves privadas y argumentos para la ejecución de la función principal del proyecto.

Desde un terminal o consola de línea de comandos, ya sea desde el *Visual Studio Code* o del sistema operativo huésped, se ejecutar el siguiente comando:

```
pip install python-decouple
```

Este procedimiento instalará la librería en el proyecto actual y permitirá utilizar sus funcionalidades, a partir de este evento se puede implementar el script de configuración.

A continuación, se requiere crear un archivo "***.env***", en la raíz del proyecto, este archivo va a contener todos los parámetros o argumentos generales que van a ser utilizados en el proyecto.

.env: Es un archivo de texto donde se definen todas las variables de entorno que se requieran y que tienen asignado un valor, para crearlo se tiene que anteponer el (.) y luego el nombre del archivo. Para usar este archivo se requiere de la instalación de las librerías Python-dotenv y Python-decouple.

La implementación de este archivo "***.env***", para nuestro caso de estudio sería el siguiente:

```
KEY=K3YS3CR3T4
MYSQL_HOST = localhost
MYSQL_USER = usuario_bd
MYSQL_PASSWORD = password_bd
MYSQL_DATABASE = nombre_bd
```

Es el momento de implementar el script que tengan las configuraciones necesarias para las futuras conexiones, en este caso particular este archivo nos va a servir para 3 cosas:

1. La clave secreta, nos permitirá enviar un token con las sesiones de usuario y el acceso a funcionalidades específicas del sistema o aplicación.

2. Los parámetros de conexión a la base de datos, estos parámetros van a ser utilizados para realizar las conexiones a la base de datos, necesarios para el intercambio de información entre la aplicación y el motor de base de datos.

3. El debug automático, esta funcionalidad permitirá hacer el Debug sin necesidad de estar reiniciando el servidor, siempre y cuando el programador lo desee.

Implementación del script *config.py*

```
1 from decouple import config
2 class Config:
3     SECRET_KEY = config('KEY')
4 class DevelopmentConfig(Config):
5     DEBUG = True
6 config = {
7     'development': DevelopmentConfig
8 }
```

La importación de la librería es primordial antes de codificar las configuraciones generales para el caso de estudio, la librería *decouple*, se importará mediante la invocación de la función ***[from]*** y su método *config* con ***[import]***, esta forma de importación se realiza para no traer todos los métodos con los que cuenta la librería *decouple*, de esta manera el compilador no tendrá que ejecutar métodos o funciones que no se van a necesitar.

La clase *Config* es utilizada para establecer la variable SECRET_KEY y asignarle el valor de la clave “KEY” como parámetro en el método *config*.

La clase *DevelopmentConfig* es la que se encargará de hacer el debug automático, siempre que sea invocado, el compilador vendrá a esta clase y ejecutará la función ***DEBUG*** en TRUE**.**

Por último, el diccionario config, por medio de la clave 'development', instanciará a la clase *DevelopmentConfig*, cada vez que sea invocada.

3.2.2 Python-dotenv

Es una librería de *Python* que proporciona dependencias para *Flask*, que se encarga de controlar todos los pares de key-values de un archivo que sean etiquetado como archivo de entorno o “.env” y, después, es el encargado de configurarlos como variables del entorno.

Esta librería te permite cargar todas las configuraciones provenientes de un archivo “***.env”*** o de entorno. De la misma forma como habíamos mencionado, para la instalación de esta librería es necesario el comando *pip*:

```
pip install python-dotenv
```

Una vez instalada la librería *Python dotenv*, permitirá obtener los valores cargados en el archivo "***.env***" y utilizarlos en los scripts durante la implementación del proyecto. La configuración de este archivo y la utilización de la librería dotenv, resulta ser muy útil al momento de levantar la aplicación en diferentes entornos sin necesidad de configurar manualmente este archivo.

3.3 Librerías Flask

Como habíamos mencionado en el capítulo anterior, *Flask* es un framework web full-stack[4] que proporciona un conjunto de bibliotecas para que los desarrolladores puedan implementar soluciones web para sus clientes.

Al momento de instalar *flask*, por defecto y automáticamente también se instalarán varias librerías, entre ellas tenemos:

- **Werkzeug** implementa WSGI, la interfaz estándar de *Python* entre aplicaciones y servidores.
- **Jinja2** es un lenguaje de plantillas que renderiza las páginas que sirve tu aplicación.

[4] Son tecnologías que abarcan el desarrollo de frondend y backend.

- **MarkupSafe** viene con *Jinja*. Escapa de la entrada no fiable cuando se renderizan las plantillas para evitar ataques de inyección.
- **ItsDangerous** firma de forma segura los datos para asegurar su integridad. Se utiliza para proteger la cookie de sesión de *Flask*.
- **Click** es un marco para escribir aplicaciones de línea de comandos. Proporciona el comando *flask* y permite añadir comandos de gestión personalizados.
- **Blinker** proporciona soporte para Señales.

En el contexto actual, haremos referencia a las librerías que se han utilizado para el caso de estudio; es importante indicar que existen más librería, por no estar dentro del alcance de la implementación, no se tomarán en cuenta.

3.3.1 Flask-WTF

Es una librería de *Flask* para *Python*, es un conjunto de herramientas para la creación de formularios web de manera rápida, sencilla y segura. Es muy útil por la protección contra ataques CSRF[5] (Cross-Site Request Forgery) y su versátil gestión de validaciones.

[5] Son medidas de seguridad para prevenir ataques cuando los usuarios ejecutan acciones no deseadas y sin permisos.

Características

- Integración con WTForms.
- Formulario seguro con token CSRF.
- Protección global CSRF.
- Soporte reCAPTCHA.
- Carga de archivos que funciona con Flask-Uploads.
- Internacionalización mediante Flask-Babel.

Para instalar esta librería, se ejecuta la siguiente instrucción:

```
pip install Flask-WTF
```

3.3.2 Flask-snippets

Es una librería de *Flask* que cuenta con un conjunto de fragmentos de código para las aplicaciones web construidas con *Python* y *Flask*.

Esta librería contiene una amplia gama de fragmentos de código que permite el uso e implementación de decoradores de rutas, renderizado de plantillas, integración de base de datos, entre otros.

Por otro lado, permite generar un esqueleto o estructura básica de directorios y archivos para empezar el desarrollo de un proyecto de software.

El proceso de instalación de esta librería es el siguiente:

```
pip install Flask-snipperts
```

3.3.3 Flask-cors

Es otra de las librerías de *Flask* que permite el manejo de intercambio de recursos de origen cruzado, instalando esta librería se puede hacer peticiones *HTTP* desde otros dominios distintos al que usa la aplicación del proyecto.

Mehmood (2022) indica que a veces creamos una API, pero no tenemos ningún problema para servir esto al usuario, pero cuando vamos a integrarnos con el front-end, surge este problema; esta es una solicitud de origen cruzado que debemos resolver (p. 2).

Es el momento donde la librería Flask-cors, permite resolver el problema informado desde el front-end y aplica el intercambio de recursos de origen cruzado, la clase *CORS* deberá responder a cualquier solicitud de rutas directas y otras solicitudes (p. 3).

Para instalar la librería se procede de la siguiente forma:

```
pip install Flask-Cors
```

3.3.4 Flask-paginate

Es otra de las librerías que se utilizarán en este caso de estudio, está orientada a crear páginas en la vista para las consultas que son extraídas de la base de datos, hace referencia a will_paginate y usa bootstrap como marco *CSS*.

Para instalarla, se debe ejecutar la siguiente instrucción:

```
pip install flask-paginate
```

3.4 Motor de base de datos MySQL

MySQL es uno de los motores de base de datos más utilizado en proyectos de software medianos y grandes a nivel mundial, la integración con proyectos es bastante sencillo ya que basta con los parámetros de configuración generales (host, puerto, username, password) para tener una conexión fiable.

El motor de base de datos *MySQL*, es el encargado de gestionar el almacenamiento, recuperación y actualización de la información que se genera en nuestra aplicación software o sistema informático.

Muchas veces se ha visto a programadores o incluso a arquitectos de software referirse a *MySQL* como un motor de almacenamiento, pero esto no es necesariamente correcto, ya que los motores de almacenamientos son módulos de las bases de datos; en el mercado existen una gran variedad de estos motores entre los más conocidos tenemos:

- **InnoDB:** es el motor de almacenamiento que cuenta con soporte para transacciones (Commits y Rollbacks) entre sus características principales tenemos:
  1. Admite bloqueo de nivel de fila,
  2. Cuenta con recuperación de fallos y control de concurrencia de múltiples versiones,
  3. Proporciona una restricción de integridad referencial de clave externa.

- **MyISAM:** es el motor de almacenamiento original de *MySQL*, cuenta con las siguientes características:
  1. Cuenta con almacenamiento rápido,
  2. No admite transacciones,
  3. Proporciona bloqueo a nivel de tabla.

- **Memory:** es conocido por su capacidad de crear tablas en la memoria, sus características son:
  1. Cuenta con almacenamiento rápido,
  2. Proporciona bloqueo a nivel de tabla,
  3. No admite transacciones,
  4. Es ideal para crear tablas temporales o búsquedas rápidas, esto provoca que los datos se pierdan cuando se reinicia el motor de base de datos.

- **CSV:** es muy conocido por la mayoría de los usuarios de aplicaciones de escritorio ya que los aplicativos de hojas de cálculo lo utilizan y sirve para almacenar datos en archivos *CSV*, sus características son:
  1. Proporciona una gran flexibilidad porque los datos en este formato se integran fácilmente en otras aplicaciones.

- **Merge**: utiliza en tablas *MyISAM* subyacentes, esta condición permite administrar grandes volúmenes de datos con mayor facilidad, sus características son:
  1. Agrupa lógicamente una serie de tablas *MyISAM* idénticas y las referencia como un solo objeto.

- **Archive:** sus características son:
  1. Permite comprimir datos a medida que se insertan,

2. No admite transacciones,
3. Es ideal para almacenar y recuperar grandes cantidades de datos históricos.
4. Es una alternativa muy eficiente para optimizar la inserción de alta velocidad.

- **Federated**: cuenta con la capacidad de separar los servidores *MySQL* para crear una base de datos lógica a partir de muchos servidores físicos, sus características son:
  1. Las consultas desde un servidor local se pueden ejecutar automáticamente en las tablas remotas o federadas,
  2. No almacena datos en las tablas locales,
  3. Es utilizado generalmente para entornos distribuidos.

Al momento de elegir un motor de almacenamiento, es muy importante establecer el ámbito del proyecto de software y como deben fluir los datos en su aplicación o sistema informático.

Como se ha visto, cada motor de almacenamiento tiene su particularidad que lo hace único y orientado para un entorno específico, es por ello que la decisión de utilizar uno u otro tiene que estar basado en un criterio técnico y flexible para los datos y la construcción de la información en beneficio de los usuarios o clientes finales.

Para el caso de estudio se va a utilizar el motor de almacenamiento InnoDB, por su capacidad de proporciona una restricción de integridad referencial de clave externa y el soporte y control de transacciones (Commits y Rollbacks).

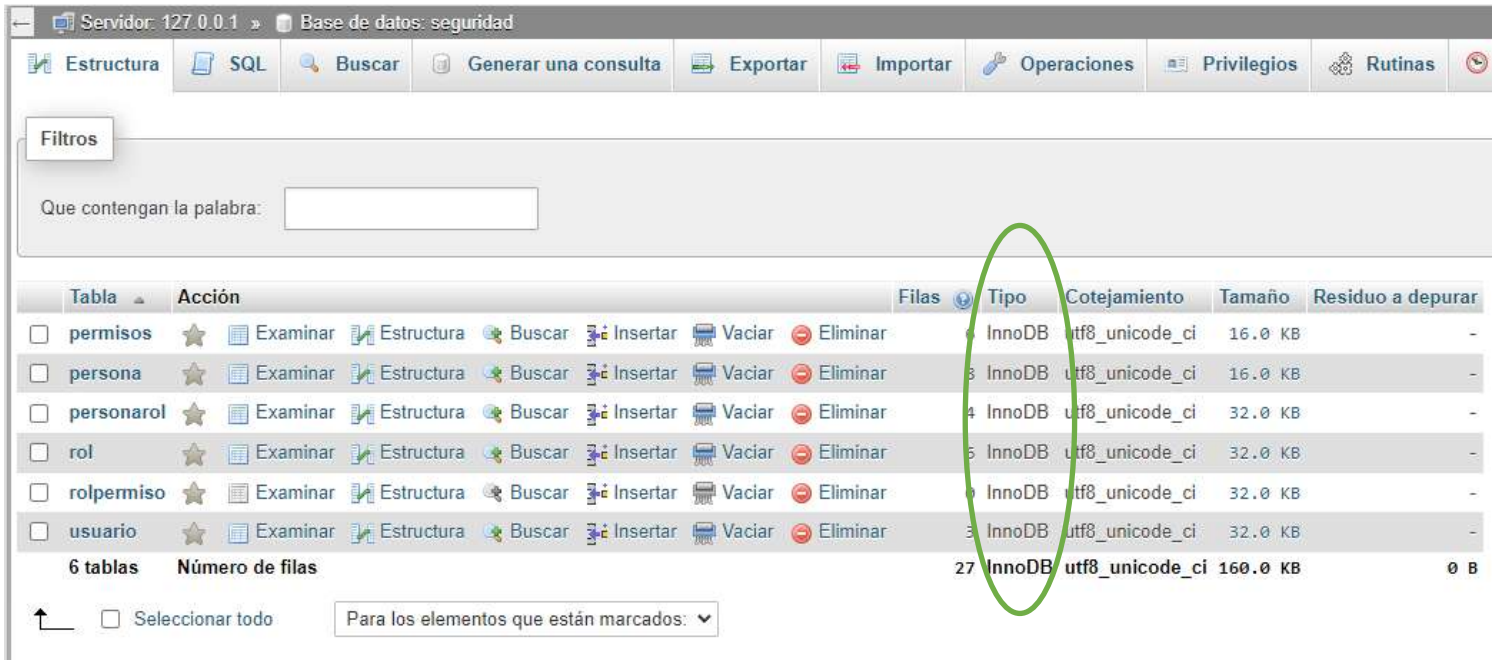
Lo primero que tenemos que crear en nuestro proyecto es la base de datos con el nombre de su preferencia (para el caso de estudio la base de datos se llamará *seguridad*) junto al tipo de cotejamiento de los datos, por lo general podemos utilizar el cotejamiento *utf8_general_ci.*

Lo siguiente es la creación de tablas y es aquí donde se alineará con el motor de almacenamiento InnoDB, se puede realizar por instrucción o utilizando el wizard de un editor de base de datos:

```
CREATE TABLE `usuario` (
  `id` int(11) NOT NULL,
  `idPersona` int(11) DEFAULT NULL,
  `username` varchar(80) DEFAULT NULL,
  `password` varchar(200) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8
COLLATE=utf8_unicode_ci;
```

Figura 12

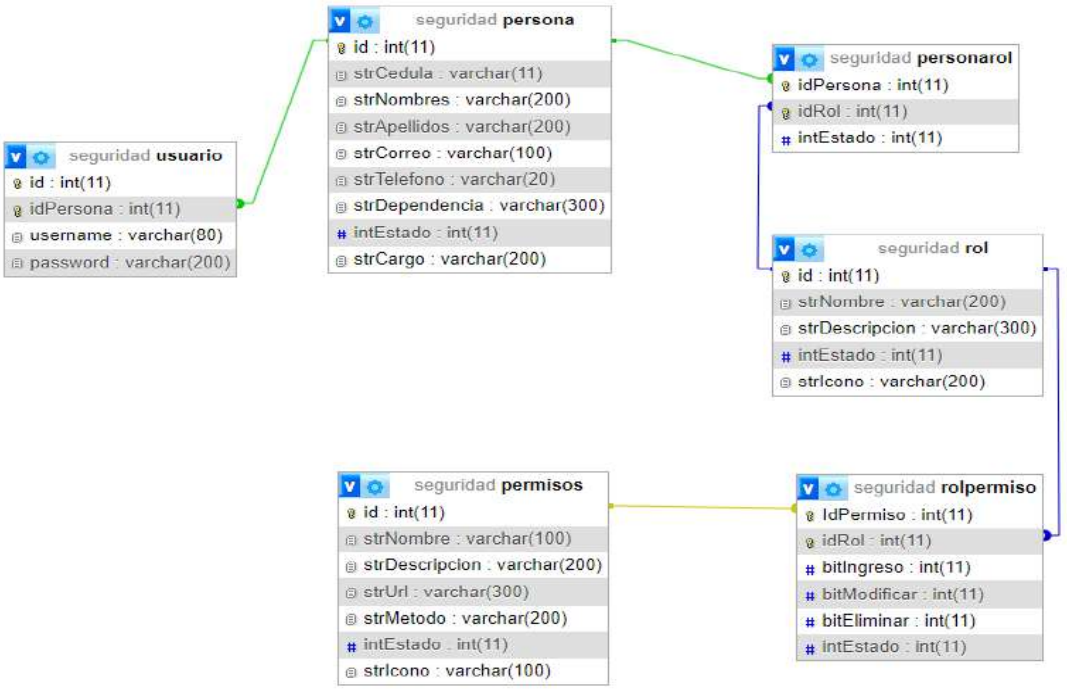
Motor de almacenamiento en MySQL.



Nota. Elaborado por los autores.

Figura 13

Esquema de la base de datos.



Nota. Elaborado por los autores.

En este capítulo se había establecido el modelo de datos que se va a utilizar en este caso de estudio, es por ello que la construcción de la base de datos debe estar basado en ese modelo.

Finalmente, tenemos que establecer el método de conexión del motor de base de datos con la aplicación o proyecto de software del caso de estudio propuesto. Para el efecto, tenemos que conocer las librerías que permitirán la interconexión de la capa de negocios con la capa de datos. *Python* y *Flask* ofrecen una variedad de librerías que permiten esta funcionalidad de intercambio de datos entre dichas capas, para el caso de estudio utilizaremos *mysql-connector-python y flask-MySQL*.

3.4.1 Librería MySQL-Connector-Python

Es una librería escrita totalmente en *Python* y no tiene ninguna referencia para su funcionamiento ideal, esta implementada para permitir la conexión de aplicaciones o sistemas informáticos escritos en *Python* puedan conectarse y acceder a los recursos de una base de datos *MySQL*.

Como se ha mencionado anteriormente, la instalación de esta librería no dista de mucho de las otras, en el terminal del editor de código o del sistema operativo huésped, hay que digitar la siguiente instrucción:

```
> pip install mysql-connector-python
```

Para estar seguros que la librería se instaló correctamente se requerirá ejecutar la siguiente instrucción:

```
> pip freeze

flask-paginate==2023.10.24
importlib-metadata==6.8.0
itsdangerous==2.1.2
Jinja2==3.1.2
MarkupSafe==2.1.3
mysql-connector-python==8.2.0
platformdirs==3.10.0
protobuf==4.21.12
python-decouple==3.8
python-dotenv==1.0.0
typing_extensions==4.7.1
virtualenv==20.24.5
Werkzeug==2.3.7
```

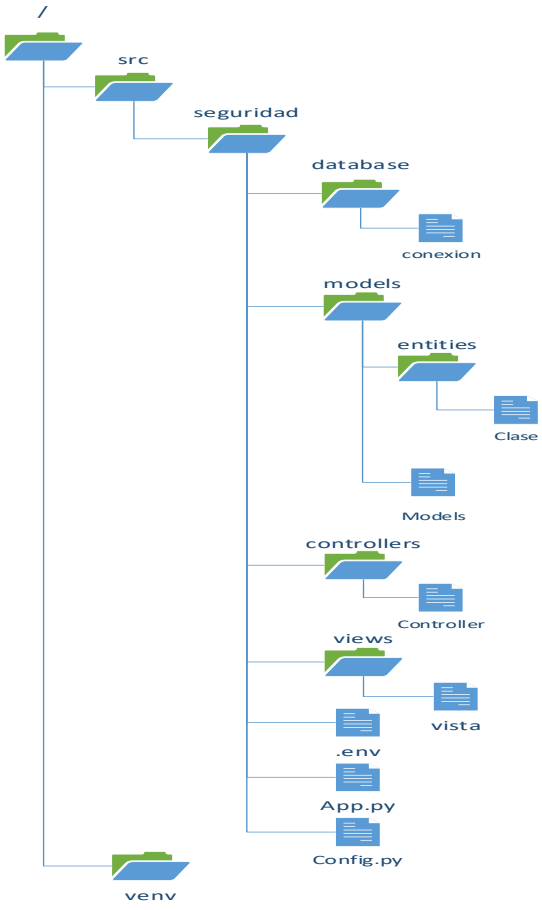
Hasta este punto, el caso de estudio ya está estructurada su capa de datos, y tiene un canal de comunicación con la capa de negocio, lo siguiente es implementar la solución respetando el patrón arquitectónico escogido para el proyecto.

3.5 Implementación del patrón arquitectónico

Como parte de la implementación del caso de estudio, el patrón arquitectónico que se va a implementar es el *MVC* (Modelo-Vista-Controlador), y su estructura funcional en el directorio, es la siguiente:

Figura 14

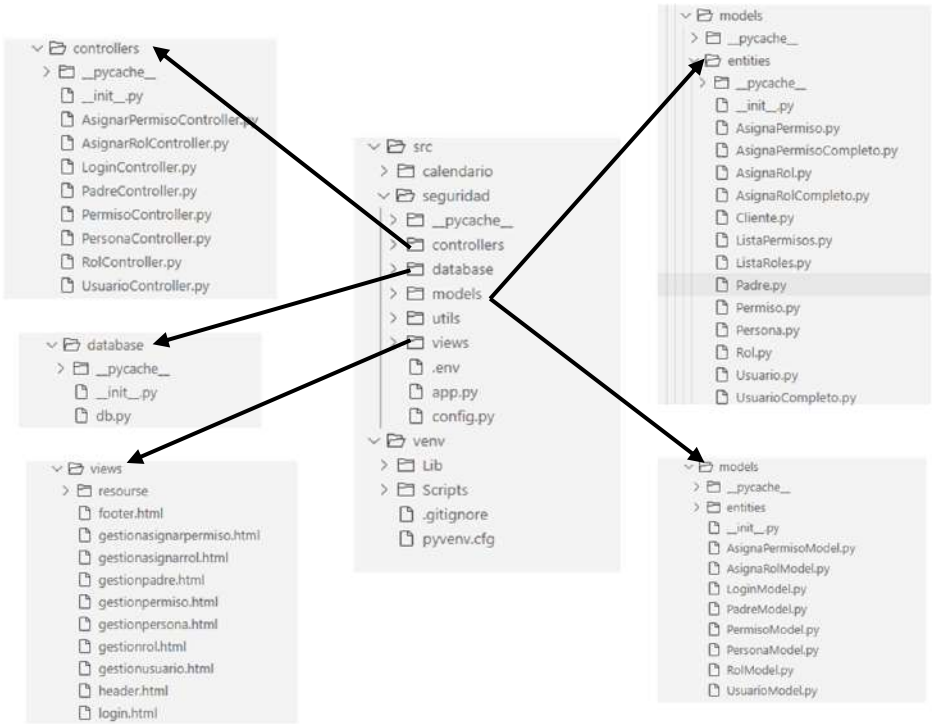
Estructura del patrón arquitectónico MVC.



Nota. Elaborado por los autores.

Figura 15

Arquitectura del patrón arquitectónico MVC.



Nota. Elaborado por los autores.

El directorio raíz es la carpeta donde estará el contenido todo el proyecto, el nombre de este directorio está sujeto a la identificación del proyecto, decisión del desarrollador o arquitecto de software. Este directorio tiene 2 sub carpetas, src y venv.

Directorio src: es la carpeta que va a contener todo el modelo *MVC* junto a sus scripts o archivos ejecutables según el módulo que implemente; es decir, dentro de esta carpeta están los módulos del sistema y su implementación.

En el caso particular, se va a implementar el módulo seguridad por ende el directorio principal del módulo se llamará, "*seguridad*".

- **Directorio seguridad**: es la carpeta que contendrá el modelo *MVC* con su implementación, pueden existir tantas carpetas de este patrón como módulos a implementar, es decir, si se va a implementar 5 módulos, deben crearse 5 directorios de este tipo.
    1. **Directorio database**: esta carpeta va a contener un archivo de conexión con la base de datos, es decir, en este directorio pueden existir múltiples archivos de conexión de acuerdo al motor de base de datos que desee conectarse.

       En el caso de estudio actual este archivo implementará la conexión con *MySQL*.

2. **Directorio models**: este directorio contendrá dos elementos, un archivo ejecutable y una carpeta para las entidades o clases.

   2.1. Carpeta entities: está destinada a contener todas las clases o entidades que se requieran para la implementación del sistema web. En este contexto es donde se aplican los patrones de diseño a las clases o entidades.

   2.2. Archivos tipo models: estos archivos los que están en constante comunicación con la capa de datos, por cada funcionalidad que implique reglas del negocio debe de contener un archivo de este tipo.

   Estos archivos tienen comunicación directa con la capa de datos y entrega información validada y filtrada a la capa de servicios, que en este caso son los controladores. Por cada controlador debe existir un archivo de tipo models.

3. **Directorio controllers**: es un contenedor de archivos que representan la capa de servicios en un patrón arquitectónico por capas, estos archivos son los que gestionan los servicios que se ofrecen a los usuarios y está en constante comunicación con la capa de negocios (models) y presentación (views), hace las funciones de enrutador de solicitudes de los usuarios y la información.

   Por cada archivo models debe existir un archivo de tipo controllers.

4. **Directorio views**: es el directorio que contiene todos los archivos o páginas de usuario, estos archivos van a interactuar con los usuarios y con la capa de servicios o archivos controladores.
5. **Archivo .env:** es el archivo que va a contener las variables de entorno del sistema informático o aplicación, aquí va estar almacenado todos los argumentos o parámetros globales que necesitamos en nuestra aplicación.
6. **Archivo app.py**: es el archivo principal de ejecución del módulo, este archivo siempre debe existir en una implementación de *Python*.
7. **Archivo config.py**: es parte de las configuraciones generales y siempre va a estar ligado al archivo "*.env*", y permitirá ejecutar algunas condiciones desde *app.py*.

- **Directorio venv**: es el directorio destinado para los entornos virtuales del proyecto, este archivo se creará automáticamente al momento de crear el entorno virtual del proyecto; es decir, el entorno virtual de este caso de estudio se llamará "*venv*"***.***

  En este directorio existe una gama de librerías instaladas que sirven para la compatibilidad de la aplicación con sus dependencias, los scripts de herramientas que requiere tener para ejecutar el entorno virtual, entre otras cosas.

3.6 Creación del entorno virtual

Como se ha mencionado, los entornos virtuales están diseñados para contener su propia instalación de *Python* y sus referencias o dependencias de bibliotecas para gestionar de manera independiente sus proyectos. Esto convierte a los entornos virtuales en una herramienta versátil para trabajar en múltiples proyectos con diferentes procesos.

Por consiguiente, el caso de estudio exige la instalación de un entorno virtual que permita contener sus propias dependencias y referencias a las librerías y herramientas con las que va a interactuar el módulo de la aplicación web que se va a implementar.

Luego de haber construido la estructura de directorios y archivos del modelo *MVC* que hace referencia a nuestro patrón arquitectónico, lo siguiente es crear el entorno virtual para este proyecto. Para ello debemos posicionar el cursor en la carpeta raíz desde el terminal o consola de línea de comandos del sistema operativo huésped y ejecutar el siguiente comando:

```
pip install virtualenv
```

Esta instrucción, ejecutará la instalación del driver o manejador de la herramienta que permitirá la creación del entorno virtual del caso

de estudio “El módulo de seguridad y manejo de usuarios, roles y permisos”.

Con la herramienta instalada, el siguiente paso es crear el entorno virtual para el módulo del caso de estudio, para ello vasta con ejecutar la siguiente instrucción:

```
python -m virtualenv venv
```

La ejecución de esta instrucción dará como resultado la creación del entorno virtual llamado *“venv”*, esto producirá la creación de un directorio con el mismo nombre y con todos los componentes necesarios para gestionar y manipular el mismo.

Al momento de revisar la configuración de este entorno, el directorio deberá estar compuesto por los siguientes componentes:

- **Directorio Lib**: este directorio va a contener todas las bibliotecas, módulos de *Python* y el subdirectorio de site-packages donde se encuentran los paquetes instalados en el entorno virtual. Las librerías de terceros se instalarán en *env/lib/pythonX.X/site-packages/*.
- **Directorio Scripts**: contiene una copia de los binarios ejecutables de *Python*, además de los scripts de activación del entorno y los de *pip*.

- **Archivo .gitignore**: es un archivo que se añade en la raíz del proyecto, sirve para controlar que archivos no hay que tener en cuenta para el control de versiones.
- **Archivo pyvenv.cfg**: en este archivo está la lista de apuntadores a las instalaciones de *Python* desde donde se ejecutó el comando de creación del entorno virtual. Este archivo concentra toda la información referente a *Python*, como ruta de instalación, versión de *Python*, versión del *virtualenv*, ruta de los *prefix*, entre otras cosas.

El siguiente paso es activar el entorno virtual, esta acción permite indicar al módulo del caso de estudio que todo lo que se ejecute se hará en el entorno virtual con las dependencias que se instalarán dentro:

```
cd .\venv\Scripts\activate
```

En consecuencia, de la ejecución de esta instrucción, el módulo ya se podrá ejecutar desde su entorno virtual, así como, proceder a instalar todas las librerías y/o bibliotecas que van a necesitar para la implementación del módulo del caso de estudio propuesto:

```
(venv) PS
D:\2023\Proyectos_Python\Calendario_académico>
```

Es el escenario indicado para la instalación de librerías y/o bibliotecas que va a requerir este caso de estudio, para ello se debe digitar las siguientes instrucciones:

Instalación de la librería *flask*.

```
(venv) PS
D:\2023\Proyectos_Python\Calendario_académico> pip
install flask
```

Instalación de la librería *Python* para *MySQL.*

```
(venv) PS
D:\2023\Proyectos_Python\Calendario_académico>pip
install mysql-connector-python
```

Instalación de la librería *Python dotenv.*

```
(venv) PS
D:\2023\Proyectos_Python\Calendario_académico>pip
```

Instalación de la librería *Python decouple*.

```
(venv) PS
D:\2023\Proyectos_Python\Calendario_académico> pip
install python-decouple
```

Instalación de la librería flask *WTF.*

```
(venv) PS D:\2023\Proyectos_Python\Calendario_académico>
pip install flask_WTF
```

Instalación de la librería *flask paginate*.

```
(venv) PS
D:\2023\Proyectos_Python\Calendario_académico> pip
```

Instalación de la librería *flask cors.*

```
(venv) PS
D:\2023\Proyectos_Python\Calendario_académico> pip
install flask-cors
```

Instalación de la librería flask cors.

```
(venv) PS
D:\2023\Proyectos_Python\Calendario_académico> pip
install flask-cors
```

En consecuencia, el entorno virtual ha sido preparado para abastecer de las funcionalidades al módulo, para poder comprobar y establecer una ruta de instalación para una posible acción de recuperación ante incidentes de daños en el entorno virtual, se fabrica un archivo llamado "*requirements*", este archivo va a contener todas las librerías y sus versiones instaladas en el entorno virtual y que son necesarias para el módulo.

Para crear este archivo se requiere ejecutar la siguiente instrucción:

```
pip freeze > requirements.txt
```

El archivo requirements.txt entonces se ha agregado a nuestro control de versiones y distribuido como parte de la aplicación. Esto permite a los programadores instalar todos los paquetes necesarios con install -r:

```
(venv) PS
D:\2023\Proyectos_Python\Calendario_académico> pip
install -r requirements.txt

Collecting blinker==1.6.3 (from -r requirements.txt
(line 1))
  ...
Collecting click==8.1.7 (from -r requirements.txt
(line 2))
  ...
Collecting colorama==0.4.6 (from -r requirements.txt
(line 3)

Successfully installed
```

Otra opción para verificar la instalación es con el comando *list o freeze.*

```
(venv) PS
D:\2023\Proyectos_Python\Calendario_académico> pip
list
```

```
(venv) PS
D:\2023\Proyectos_Python\Calendario_académico> pip
freeze
```

Luego de haber trabajado en el proyecto, es necesario y seguro desactivar el entorno virtual ejecutando el siguiente comando:

```
(venv)PS
D:\2023\Proyectos_Python\Calendario_académico>
.\venv\Scripts\deactivate.bat
```

Con la acción anterior, el sistema volverá a entorno global de *Python* y evitará conflictos de dependencias de ese entorno con otros proyectos resguardando la incompatibilidad, liberará recursos del sistema como memoria o espacio en disco, y mantendrá el entorno limpio ya que se evita que queden configuraciones o dependencias activas.

Conclusiones

- El establecimiento del entorno de trabajo para el desarrollo de un módulo, aplicación o sistema se convierte en una tarea muy importante al momento de establecer un patrón arquitectónico, este elemento de organización va a permitir a mediano y largo plazo un producto estable, eficiente y mantenible a través del tiempo.
- Los entornos virtuales están diseñados para contener su propio conjunto de referencias o dependencias de bibliotecas para gestionar de manera independiente sus proyectos. Esto convierte a los entornos virtuales en una herramienta versátil

para trabajar en múltiples proyectos con diferentes procesos o procedimientos.

- El patrón arquitectónico *MVC* (Modelo-Vista-Controlador), permitirá tener un gran nivel de organización de nuestro código al momento de la implementación y el mantenimiento del módulo o sistema informático, permitirá identificar con rapidez, incidentes, mejoras y actualizaciones de nuestro código.

Recomendaciones

- Es importante que se proceda con la instalación y activación del entorno virtual del proyecto, antes de la instalación de las librerías o paquetes que se van a utilizar para desarrollar la solución informática.
- Luego de concluir con los trabajos de implementación de la solución informática, se recomienda proceder con la desactivación del entorno virtual del proyecto.

CAPÍTULO 4
Desarrollo del módulo

4

Capítulo

4

Desarollo del módulo

Este capítulo se enfocará en el desarrollo del módulo de seguridad y manejo de usuarios, roles y permisos, con la herramienta de desarrollo *Python*, el framework *Flask* y la herramienta de diseño bootstrap. Hasta este punto ya tenemos una idea de qué son, cómo se instalan, para qué sirven y cómo se utilizan estas herramientas en un proyecto software.

El propósito en esta fase es aprender a utilizar estas herramientas para lograr implementar una aplicación web, con un patrón arquitectónico *MVC* y orientado a objetos, de tal manera que nos permita motivar al lector en la construcción de sistemas informáticos o aplicaciones web más robustas y flexible en su vida profesional o estudiantil.

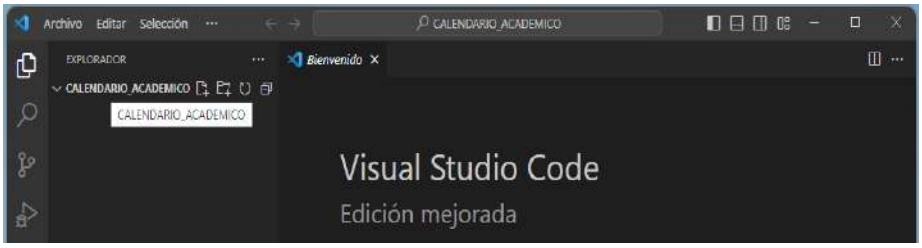
En el capítulo anterior se había mencionado la construcción del patrón arquitectónico *MVC*, mismo que se debe construirse en el editor de código de la siguiente manera:

Tenemos que crear la carpeta o directorio principal del proyecto, para el caso de estudio le hemos llamado "CALENDARIO_ACADÉMICO", ya que el módulo seguridad es parte de un sistema informático que gestiona el calendario académico de una institución educativa.

Luego de ejecutar el editor de código, arrastraremos esta carpeta dentro del editor e inmediatamente se ubica en el entorno de trabajo del editor de código lista para comenzar.

Figura 16

Directorio raíz.



Nota. Elaborado por los autores.

A continuación se creará un directorio donde se concentrará todo el código de los módulos que van a formar parte de la solución

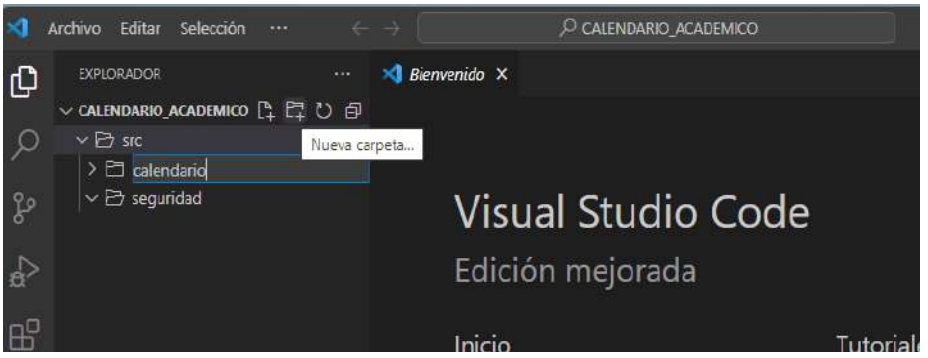
completa; este directorio es opcional y solo se utiliza para organizar de mejor forma la codificación del sistema.

Para el caso de estudio se creará el directorio *"src"* y contendrá las carpetas de cada módulo que pueda o tenga la solución completa (sistema informático), este directorio a su vez contendrá las carpetas ***"calendario"*** y ***"seguridad"***.

Es en el directorio seguridad donde se concentrará la implementación de este caso de estudio ya que el objetivo de este libro es implementar el módulo de seguridad y manejo de usuarios, roles y permisos.

Figura 17

Directorio src y directorio de módulos.

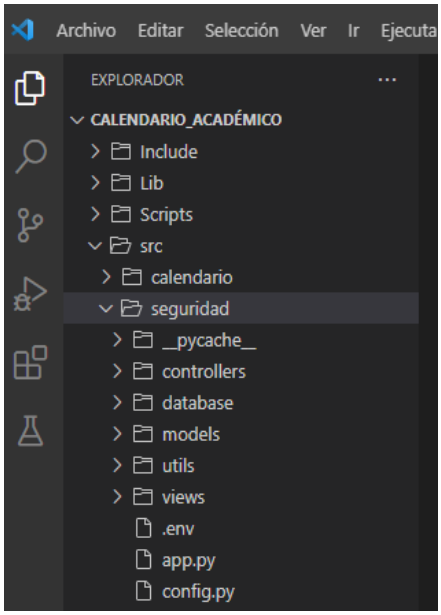


Nota. Elaborado por los autores.

En el directorio “seguridad”, se implementará el patrón arquitectónico que se mencionó en el capítulo anterior.

Figura 18

MVC implementado en el módulo seguridad.



Nota. Elaborado por los autores.

Como ya se mencionó, el directorio “seguridad” contendrá las carpetas: controllers, database, models, views, .env, app.py, config.py. Además, el directorio *models* va a contener al directorio ***entities*** compuestas por las clases o modelo de datos.

Hasta este punto hemos concretado la implementación del patrón arquitectónico *MVC*, y la estructura está lista para contener los

scripts que el sistema requiera, tal como fue diseñado en los mockups del capítulo 2.

Por cada directorio o carpeta que se crea en este punto, debemos indicarle a *Python* que van a ser componentes, para ello, luego de la creación de los directorios o carpetas se debe crear un archivo *__init__.py*.

El archivo *__init__.py*, sirve para indicar que un directorio es un paquete o componente de *Python*, eso permitirá la importación de módulos desde ese directorio, es decir, tendrá como función fundamental iniciar el paquete o componente y definir su contenido.

Además, permite definir la interfaz pública del paquete o componente con ello se podrá importar dicho paquete desde el exterior. Por último, ejecuta el código de inicialización del paquete, es en este archivo donde se puede colocar el código que requiera un paquete para ser inicializado, esto puede ser variables, configuraciones de entorno, carga de versiones o cualquier tarea que desea ejecutar y estar listas antes de ser usadas.

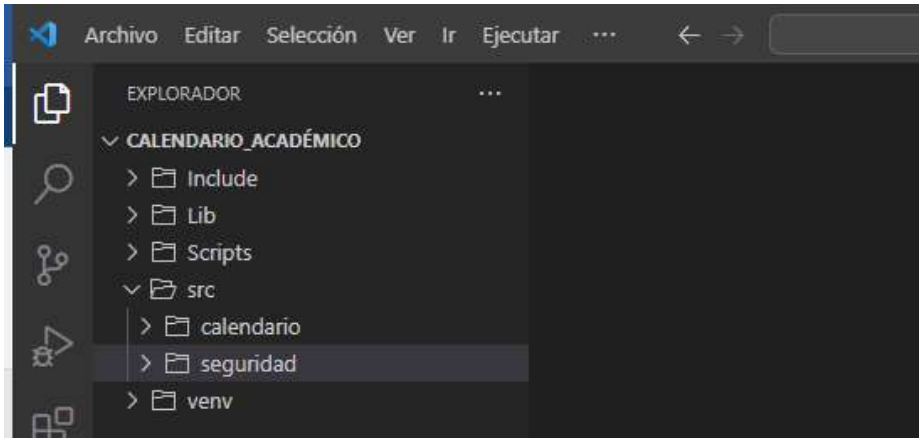
Por otra parte, también vamos a requerir de ejecutar la creación del entorno virtual en la solución, es recomendable crear un entorno virtual para la solución para garantizar que tenga las dependencias y librerías necesarias e independientes para cada módulo, de esta

manera no necesitaremos actualizar toda la aplicación en caso que algún modulo tenga una versión distinta de librerías en el servidor huésped.

De la misma manera como se explicó anteriormente debemos ejecutar la instrucción, *python -m virtualenv venv*, para luego activar el entorno virtual y empezar a instalar todas las librerías y/o bibliotecas que se requieran, es importante indicar que es aconsejable instalar solo lo que la implementación del módulo requiera. (Para más detalle sobre la instalación y creación de los entornos virtuales revisar el capítulo 3). Para comprobar que el entorno virtual esta creado, el editor de código mostrará un directorio con el nombre del entorno virtual, en este caso se llamará *"venv"***.**

Figura 19

Entorno virtual venv.

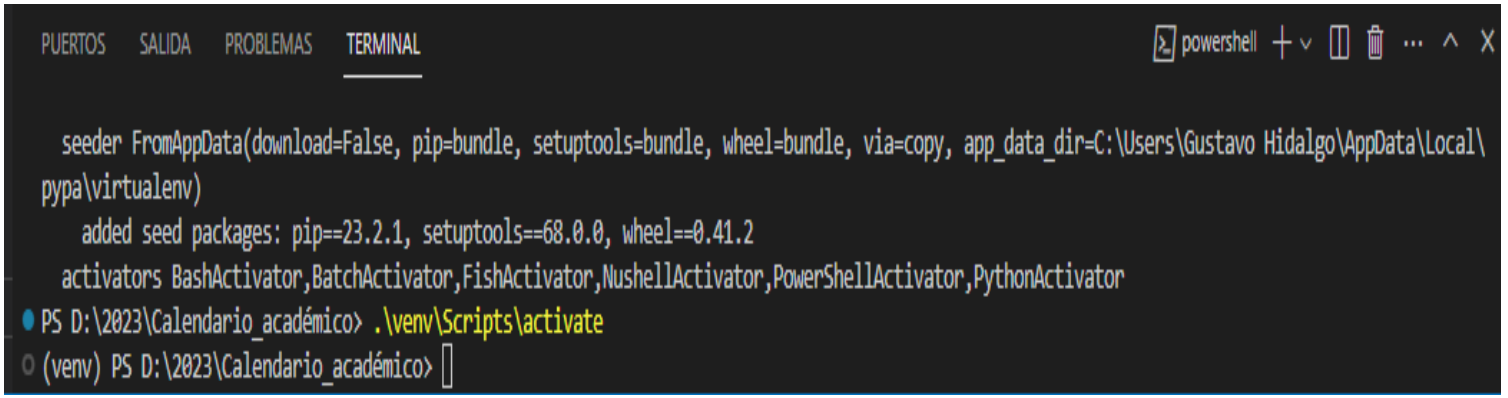


Nota. Elaborado por: Los autores.

Por último, desde una consola de línea de comandos del sistema operativo huésped o terminal del editor de código debemos ejecutar la activación del entorno virtual para poder instalar todas sus dependencias.

Figura 20

Activación del entorno virtual venv.



Nota. Elaborado por los autores.

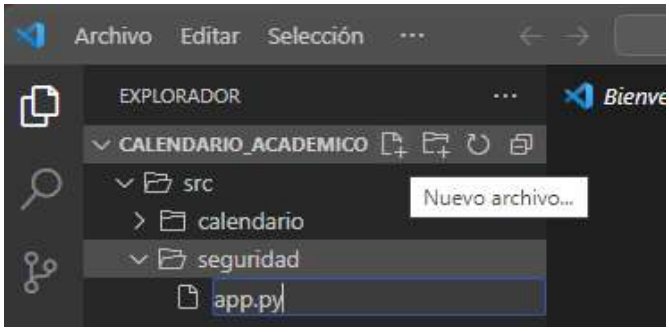
4.1. Archivo principal app.py

"*Como todo en la vida, por algo se tiene que empezar*".

Es el punto de partida de una aplicación o sistema informático escrito en *Python*, puede llamarse como el desarrollador o arquitecto lo desee, pero en este caso de estudio se llamará "*app.py*". Lo primero que se debe hacer es crear un script o archivo *Python* principal en la raíz del directorio del módulo que para esta ocasión se llamará *"seguridad"* y el archivo llevará el nombre de "*app.py*", es importante indicar que los scripts de *Python* llevan la extensión "*py*". Esta extensión indicará al depurador que ejecute un archivo *Python*. Para crear un archivo o carpeta basta con dar clic en el icono que aparece junto al nombre del directorio raíz y posicionarse en la carpeta donde desea el archivo o carpeta.

Figura 21

Creación de archivos.



Nota. Elaborado por los autores.

Este archivo script, está escrito de manera que sirva como punto de partida para la ejecución de los demás archivos o scripts según sean convocados, es por ello, que se requiere tener establecido 3 criterios importantes.

El primero es el acceso a los archivos donde se encuentran los scripts, el segundo es la instanciación del objeto *Flask* y la tercera es la ejecución de la aplicación.

Es importante indicar que todos los archivos *Python*, necesitan importar ciertas librerías para que puedan ejecutar determinadas instancias según el rol que cumplan estos archivos. En este caso particular, como es el archivo principal requeriremos librerías específicas para los 3 criterios antes mencionados.

Es importante indicar que no necesariamente va a existir 3 criterios que controlar en este archivo principal, depende mucho del criterio y la orientación que el desarrollador (programador) y/o arquitecto le dé al proyecto software.

Para manejar los 3 criterios antes mencionados, el caso de estudio requiere la importación de las siguientes librerías y componentes:

Antes de continuar es necesario aclarar la diferencia entre *from* e *import*. *Import*. es una función que nos ayuda a traer todos los componentes y/o funciones de una determinada biblioteca, mientras que *from …. import*, traerá solo el o los componentes que están descrito en la línea.

```
#Importación d librerías

from flask import Flask
from config import config
from flask_cors import CORS
import os
from controllers import PersonaController,RolController,
PermisoController,AsignarRolController,
AsignarPermisoController,UsuarioController,LoginController
```

La importación de la librería *flask* y su componente *Flask*, va a permitir poder instanciar un objeto de tipo clase *Flask*, elemento fundamental y primordial para el desarrollo de aplicaciones web con el lenguaje *Python* y el framework *flask*.

Con la importación de la librería *config* y su componente *config*, hará más eficiente la actualización del código en tiempo de ejecución, es decir que no requerirá de reiniciar la ejecución del programa para que los cambios se actualicen. Esto se debe a que gracias a la implementación del archivo *Python, config.py* se podrá acceder al diccionario *config* con la clave *'development'*, cuyo valor es el nombre

de la clase *"DevelopmentConfig"* que a su vez tiene como argumento la constante "*DEBUG"* igual a *"True"*. Esto quiere decir, que cada vez que se ejecute esta propiedad el compilador siempre ejecutará los cambios sin necesidad de reiniciar el programa.

Por lo general, los servidores bloquean cualquier solicitud que venga de otro dominio, este procedimiento lo realiza por seguridad ya que es poco probable que alguna aplicación externa necesita hacer operaciones sobre nuestra aplicación, para controlar el intercambio. En casos de este ejemplo, si alguna operación externa necesita algo, puede abrir la puerta para comunicarse de manera controlada.

Es verdad que en el caso de estudio actual no requiere este procedimiento ya que es un módulo de aprendizaje y no va a estar por ahora comunicándose con ningún ente externo ni interno, no es necesario importar esta librería, pero si es importante conocer cómo funciona.

La importación del módulo *"os"*, va a permitir al script usar funcionalidades del sistema operativo que en este caso de estudio nos va a servir para acceso a las rutas de los directorios donde van a estar nuestros scripts.

Por último, importaremos el paquete o componente *"controllers"* construido por el desarrollador, y es parte del patrón

arquitectónico *MVC*, en este componente, están todos los controladores de las funcionalidades que se vayan a implementar, es decir, va a contener por cada vista un controlador que se comunica con un modelo de una determinada funcionalidad o tarea que se va a implementar para ser parte de la aplicación web en construcción.

4.1.1. Acceso a los archivos

El caso de estudio requiere tener una variable que nos permita conocer el directorio raíz, así como el directorio del módulo seguridad y sus vistas (views), esto permitirá de manera eficiente y sin equivocaciones encontrar los archivos *Python* o scripts que se ejecutará.

Para ello se implementa las siguientes líneas de código.

```
# directorio donde esta nuestra aplicación para acceder a
nuestro archivo
dir =
os.path.dirname(os.path.abspath(os.path.dirname(__file__)))
# rescribir el directorio y concatenarlo con las otras
carpetas
dir = os.path.join(dir,'seguridad','views')
```

os.path.dirname, es un método de *Python* que se utiliza para obtener el nombre del directorio de la ruta especificada, por lo que su ejecución devolverá un valor de cadena que representa el nombre del directorio

de la ruta especificada. Luego la función *os.path.abspath()* se usa para convertirla en una ruta absoluta antes de eliminar sólo el nombre del archivo y almacenar la ruta completa al directorio en el que se encuentra el módulo; cuando volvemos a utilizar *os.path.dirname* se obtendrá, la ruta específica del archivo *__file__* que es el nombre de ruta del archivo desde el cual se cargó el módulo, en caso que se haya cargado desde un archivo.

La línea anterior dará como resultado: *dir = D:\2023\Calendario_académico\src*, ya que el archivo ejecutado es el *app.py*.

os.path.join(), es otro método de Python que concatena uno o más componentes de rutas, con un separador de directorio ('/') después de cada espacio vacío, excepto el último componente de la ruta. Si el último componente está vacío, se coloca un separador de directorio ('/') al final.

Si un componente representa una ruta absoluta, y todos anteriores están concatenados, se descartan y la concatenación continúa desde la ruta absoluta. Este método devuelve una cadena que representa los componentes de la ruta concatenada.

Como parámetro se ha enviado el directorio de la ruta específica y los dos directorios que se requieren concatenar, entonces

el resultado de la nueva variable ***dir*** será: *D:\2023\Calendario_académico\src\seguridad\views.*

4.1.2. Instanciación de la clase Flask

En Python el término lanzar la aplicación, es la acción de ejecutar un conjunto de instrucciones con la instancia de un objeto, en este caso particular esta instancia se llama "*app*" y es de la clase *flask*.

```
#inicializar flask para poder lanzar nuestra aplicación
app = Flask(__name__,template_folder=dir)
```

Esta instrucción refleja la instancia de un objeto *app* de tipo *Flask,* el argumento *__name__* es el constructor de la clase y el argumento *template_folder* en cambio busca un conjunto de plantillas en un directorio, esta configuración se puede cambiar para adaptarse a la necesidad del proyecto, en este caso de estudio en la variable *template_folder*, se enviará la variable *dir,* que contiene la ruta específica (*D:\2023\Calendario_académico\src\seguridad\views*) de nuestro directorio de plantillas.

Esta instrucción cambia la ubicación predeterminada de las plantillas, a la ubicación contenida en la variable ***dir*** y hace referencia al directorio donde están las plantillas personalizadas para la implementación del proyecto.

4.1.3. Función de iniciación if __name__=='__main__'

La función de inicialización de la aplicación *if __name__=='__main__',* siempre debe de estar invocada en una aplicación con codificación del lenguaje *Python*, el intérprete siempre va a asignar un atributo para cada módulo que se esté ejecutando, por lo general este atributo es el propio nombre del módulo, por esa razón se referencia la variable *__name__*.

Luego que el intérprete asigna el valor del módulo con el valor de *__main__*, se está indicando que es el archivo principal del módulo. Por lo expuesto, cuando se utiliza la condición *if __name__=='__main__'*, el interprete está en la obligación de ejecutar desde el archivo principal, todas las instrucciones que estén dentro de esa condición.

```
if __name__=='__main__':
    app.config.from_object(config['development'])
    #Blueprints
    app.register_blueprint(LoginController.main,url_prefix=
'/')
    app.register_blueprint(PersonaController.main,url_prefi
x='/seg/persona')
    app.register_blueprint(RolController.main,url_prefix='/
seg/rol')
    app.register_blueprint(PermisoController.main,url_prefi
x='/seg/permiso')
    app.register_blueprint(AsignarRolController.main,url_pr
efix='/seg/asignarrol')
```

```
    app.register_blueprint(AsignarPermisoController.main,url_prefix='/seg/asignarpermiso')
    app.register_blueprint(UsuarioController.main,url_prefix='/seg/usuario')

    app.run(port=4000)
```

Con la instancia de la clase *Flask* es posible utilizar los archivos de configuración el cual puede contener un sin número de parámetros que ayudan a la configuración del sistema, estos valores de configuración también pueden ser ingresados o creados por el desarrollador, en este caso de estudio, hemos configurado el archivo *config.py* que va a contener entre otras cosas el diccionario *config* cuya clave es el nombre de la clase que permite ejecutar los cambios del código *Python* de la aplicación sin necesidad de reiniciar todo el módulo que se encuentra en ejecución.

De esta manera, al invocar la instrucción *app.config.from_object(config['development'])*, el compilador responderá ejecutando los cambios al código sin necesidad de reiniciar o parar la ejecución. Esta acción se puede realizar ya que la función *config.from_object*, permite actualizar los valores de un objeto dado, es decir, como el objeto *config['development']* contiene el atributo *DEBUG = True*, indicará que debe actualizar sus acciones y modificaciones cada vez que es invocado.

Como hemos mencionado anteriormente, la aplicación que se está implementando es de carácter modular y precisamente pre ello, se requiere trabajar con los *Blueprints*. El *Blueprints* permite crear componentes para soportar patrones arquitectónicos dentro de nuestra aplicación, es decir, permite extender una aplicación.

Para utilizar *Flask Blueprint*, se requiere importar y registrar los end-point dentro de la aplicación o sistema informático en construcción, con ello se permitirá extender nuestra aplicación de manera eficiente y cubriendo la necesidad de implementación que requiera un proyecto.

La *app.register_blueprint(LoginController.main,url_prefix='/')*, instrucción permite registrar un end-point de la página principal de nuestra aplicación, que en este caso es la página de *login*, la función *register_blueprint* en esta ocasión recibirá dos parámetros; el *LoginController.main,* como representación de la función principal del archivo controlador del componente *Login (*cuando el usuario digite en un explorador de internet la dirección url http://miservidor:puerto/), y el depurador que internamente ejecutará el código contenido en el archivo *LoginController.py**.***

Luego *Blueprints,* buscará el parámetro *url_prefix='/'*, inmediatamente el depurador que ya se encuentra en el archivo

LoginController.py ejecutará la función que contenga de decorador con el prefijo (‘/’).

Cuando un usuario determinado digite en un explorador de internet la dirección url http://miservidor:puerto/seg/persona, el sistema inmediatamente ejecutará esta instrucción *app.register_blueprint(PersonaController.main,url_prefix='/seg/persona')* y el depurador internamente ejecutará el código contenido en el archivo *PersonaController.py* luego buscará el parámetro *url_prefix='/'*, con este valor el depurador que ya se encuentra en el archivo *LoginController.py* ejecutará la función que contenga de decorador con el prefijo (‘/’).

Siguiendo el mismo procedimiento se ejecutan las instrucciones:
app.register_blueprint(RolController.main,url_prefix='/seg/rol'),
app.register_blueprint(PermisoController.main,url_prefix='/seg/permiso'),
app.register_blueprint(AsignarRolController.main,url_prefix='/seg/asignarrol'),
app.register_blueprint(AsignarPermisoController.main,url_prefix='/seg/asignarpermiso'),
app.register_blueprint(UsuarioController.main,url_prefix='/seg/usuario'), siempre ejecutando el controlador y su prefijo correspondiente a

su invocación, a partir del ingreso por teclado que el usuario haya indicado en el explorador de internet.

Por último, se requiere lanzar o ejecutar la aplicación para ello utilizamos la instrucción *app.run(port=4000)*, indicándole el puerto donde se ejecutará dicha aplicación web. En este punto se puede ejecutar el archivo *Python* principal, para ello se requiere abrir un terminal en el editor de código, luego posicionarse en el directorio *cd /src/seguridad/* y ejecutar el comando *python app.py*. De esta manera se ejecutará de manera local la url http://127.0.0.1:4000/, desde el explorador de internet del cliente.

Figura 22

Página de inicio.

Nota. Elaborado por los autores.

4.2. Implementación de la clase get_connection

La función *get_connection* también es un componente del sistema y está diseñada e implementada para ser el puente de conexión entre el modelo y la capa de datos, esta clase es la encargada de conectarse a la base de datos y gestionar las consultas que vienen del modelo (capa modelo).

```
import mysql.connector
from decouple import config

def get_connection():
    try:
        conexion = mysql.connector.connect(
            host = config('MYSQL_HOST'),
            user = config('MYSQL_USER'),
            password = config('MYSQL_PASSWORD'),
            database = config('MYSQL_DATABASE')
        )
        return conexion
    except mysql.connector.Error as error:
        raise error
```

Cuando se empieza a escribir un script en un lenguaje de programación, lo primero que hay que invocar son las librerías o componentes que se van a utilizar en el código del script. Esta implementación está enfocada en realizar las tareas de conexión con la base de datos, por ello se requiere importar el driver de conexión *mysql.connector*, mismo que permitirá establecer el controlador o canal de conexión con la base de datos; y la librería *decouple* con su

componente *config*, que permitirá extraer las variables globales contenidas en el archivo *.env*.

El contenido del archivo. env es el siguiente:

```
KEY=N1C0L4S
MYSQL_HOST = localhost
MYSQL_USER = gus
MYSQL_PASSWORD = gus123
MYSQL_DATABASE = seguridad_db
```

La función *get_connection()*, ejecutará *mysql.connector.connect()* misma que realizará la conexión y para ello utilizará la variables del archivo *.env*, invocándolos por medio de la librería *decouple* y su función *config*, en este punto va a extraer los valores de las variables 'MYSQL_HOST', 'MYSQL_USER', 'MYSQL_PASSWORD', 'MYSQL_DATABASE'. Con ello establecerá la conexión y retornará la cadena de conexión en la variable *conexión.*

En caso de existir algún problema, la función devolverá un error.

4.3. Implementación de la gestión de usuarios del sistema

El componente de gestión de usuarios del sistema, es el responsable de crear, actualizar y eliminar usuarios. Es la gestión

encargada de interactuar con el login para permitir el acceso a los recursos del sistema.

4.3.1. Clase Usuario

La clase *Usuario* es la estructura de datos que contiene los parámetros y métodos necesarios para almacenar información referente a los usuarios. Esta clase se utilizará para gestionar los datos de los usuarios en las diferentes capas del patrón arquitectónico *MVC*.

La implementación de la clase *Usuario*, debe estar acorde a la estructura de la tabla usuario de la base de datos, por ello su implementación es la siguiente:

```
from werkzeug.security import check_password_hash

class Usuario():
    def __init__(self,id,idPersona,username,password) ->
None:
        self.id = id
        self.idPersona = idPersona
        self.username = username
        self.password = password

    def __str__(self) -> str:
        return {
            'id': self.id,
            'idPersona': self.idPersona,
            'username': self.username,
            'password': self.password
```

```
        }

    def __hash__(self, pasword_hashed, password) -> int:
        return check_password_hash(pasword_hashed,password)
```

La importación de la librería *werkzeug.security,* proporciona el componente *check_password_hash* que contiene la función *check_password_hash* misma que permite encriptar una cadena de caracteres con el algoritmo *hashed md5 y sha1.*

La declaración de la clase *Usuario* siempre debe de terminar con paréntesis y los dos puntos ":", es importante establecer las sangrías en el código, ya que le indican al compilador qué líneas pertenecen a una estructura de control en el código.

El siguiente paso es establecer el constructor mediante la función *def __init__():* esta función permitirá establecer la construcción del objeto en el momento de su instanciación. En la clase *Usuario*, se requiere tener como parámetros los mismos campos con los que cuenta la tabla *usuario* de la base de datos *seguridad_db*, entonces la función construirá un objeto con los parámetros *def __init__(self, id,idPersona,username,password).*

El parámetro self, siempre va a estar presente en la definición de una clase ya que es un puntero que hace referencia al objeto que se está manipulando cuando se llama aun método de dicha clase.

Por cada valor enviado a la clase *Usuario*, se debe asignar los valores recibidos por cada parámetro de la clase, esto provocará la construcción del objeto instanciado.

En la estructura de la clase, es necesario definir el método *__str__* como manera de mostrar los valores que están presentes en el objeto instanciado. La función *__str__*, no recibe parámetros, pero si referencia al puntero *self*, que permitirá extraer los datos del objeto, retornando un diccionario que está estructurado de manera que las claves serán siempre los nombres de los parámetros de la función y los valores la correspondencia que fue enviada al momento de instanciar el objeto.

Por último, y como método particular de esta clase se implementa la función *def __hash__(self, pasword_hashed, password)*, que permite ejecutar la función *check_password_hash(pasword_hashed,password)* destinada a encriptar o hashed el password de un usuario del sistema. Es decir, cada vez que crea un usuario y se invoca a esta función, la implementación envía el password del usuario y esta función la modifica o realiza hashed para convertirla en una cadena encriptada de caracteres.

Es importante indicar que este patrón de diseño estará implementado en todas las clases que formarán parte de las entidades

(*entities*) de este módulo para el caso de estudio a excepción de la función *def __hash__(self, pasword_hashed, password)*, ya que esta es parte de la librería *werkzeug.security*.

4.3.2. Usuario Controlador (UsuarioController.py)

La capa de controladores es el puente de conexión entre el modelo y la vista, nos permitirá establecer las reglas para el negocio que queremos implementar en el módulo de seguridad de la aplicación web.

```
#Importación de flask y sus componentes
from flask import
Blueprint,redirect,jsonify,request,url_for,session

# importar los models
from models.UsuarioModel import UsuarioModel
from models.entities.Usuario import Usuario
```

Lo primero que tenemos que contemplar, es la importación de las librerías y componentes que se van a necesitar en la implementación del controlador para los usuarios. Por ello, se debe importar el framework *flask* y sus métodos *Blueprint, redirect, jsonify, request, url_for, session*.

- **Blueprint**: esta librería permitirá soportar patrones comunes dentro de la aplicación, es decir, permitirá establecer un filtro

de la página que contendrá sus funcionalidades, permitiendo instanciar un objeto que podrá encaminar las rutas que hacen referencia al componente Usuario.

- **redirect**: esta librería se utilizará para redirigir a otras rutas en el mismo módulo, es decir, cada vez que se requiera ir a una determinada página esta función redirigirá el funcionamiento a la ruta especificada.
- **jsonify**: permite serializar los valores a un tipo de respuesta JSON, de esta manera cualquier lista o diccionario que genere una respuesta de un código específico se devolverá en un formato JSON, siempre será un método que nos sirve para entregar una respuesta al usuario.
- **request:** este método es utilizado para capturar los valores que envía la vista al controlador ya sea por el método POST o GET. Es decir, el contenido del Cliente Web es receptado por el servidor mediante el método ***request***.
- **url_for:** permite crear y generar URL en una aplicación, de esta manera la función se utiliza para codificar y visualizar las plantillas que son invocadas en sus parámetros.
- **session:** esta extensión permite manejar las variables de sesión de una manera fácil y dinámica. Una sesión es el tiempo que un usuario accede al servidor hasta que sale. Los datos que deben guardarse en la sesión se almacenan en un directorio temporal en el servidor.

La importación del modelo *models.UsuarioModel* y su componente *UsuarioModel*, va a permitir la comunicación del controlador con el modelo. De esta manera, el desarrollador podrá utilizar todos los métodos que se encuentran implementados en el componente *UsuarioModel.*

De la misma manera, la importación de la entidad *Usuario* mediante *models.entities.Usuario* permitirá instanciar los objetos de este tipo para poder realizar las operaciones que requiere el patrón arquitectónico *MVC*.

Lo primero que hay que hacer en este y todos los controladores que se van a implementar, es la instanciación de la variable *Blueprint*, esta variable permitirá establecer las rutas y enlazarlas con su respectiva función.

```
main = Blueprint('Usuario_blueprint',__name__)
```

El parámetro *Usuario_blueprint*, nos indica el punto inicial de la ruta correspondiente a la gestión del usuario, mientras que *__name__* especifica el módulo o paquete al que corresponde este blueprint, en este caso apuntará a la carpeta *D:\2023\Calendario_académico\src\seguridad\views*.

```
@main.route('/')
def home():
```

```
    try:
        if "username" in session:
            return UsuarioModel.get_page()
        else:
            return
redirect(url_for('Login_blueprint.home'))
    except Exception as ex:
        return jsonify({'mensaje':str(ex)}),500
```

Es necesario indicar que *@main.route('/'),* se lo conoce como decorador y sirve para enrutar las peticiones a un determinado path.

Cuando el cliente web (usuario), ingresa a la ruta *http://127.0.0.1:4000/seg/usuario*, el compilador inmediatamente enruta la ejecución a la función *def home()*, ya que en este controlador, la ruta raíz es */seg/usuario*, que fue registrada en el archivo de ejecución principal app.py, y ejecutada en la línea *app.register_blueprint(UsuarioController.main,url_prefix='/seg/usuario')*. En el archivo controlador esa representación equivale a la ruta (/).

Dicho de otra manera, cuando se registrar una *url_prefix* en el archivo de ejecución principal significa que va a corresponder a la raíz del archivo controlador, esto significa que cuando el usuario escriba el *url_prefix =/seg/usuario,* realmente el compilador está ejecutando el prefijo *('/')*.

Con el decorador identificado, el compilador de *Python* se dirige a la función *def home()*, esta función tiene la responsabilidad de mostrar la página principal del componente “Gestión de Usuario”, es decir, va a mostrar los valores contenidos en la tabla tabla *usuario* de la base de datos *seguridad_db,* además de presentar los eventos de tipo botón “Nuevo”, “Editar” y “Eliminar”.

Este procedimiento está contemplado en la función *get_page(),* que forma parte del modelo *UsuarioModel*, es la encargada de comunicarse con la base de datos y realizar todas las transacciones que el controlador le solicite (más adelante se detallará su funcionamiento).

Pero antes, realiza una validación de la sesión del usuario, es decir pregunta si algún usuario esta validado, en caso de que lo esté, permite el acceso al procedimiento antes mencionado, caso contrario lo redirecciona a la página de login.

La importancia de manejar los errores radica en tener un código más estable y manejable, es por eso que todas las funciones que se implementarán en este caso de estudio van a contener la estructura *try-except*, para manejo de errores. Es por eso, que en caso de algún error en el código, el *try-except* responderá con un mensaje tipo *JSON* en la variable *ex*.

De la misma forma, cuando el cliente web presione el botón" Nuevo", internamente hace referencia a la ruta */add*, inmediatamente el compilador regresa al controlador a buscar el decorador con el nombre */add* y ejecutará la función que le precede a la línea encontrada, en este caso ejecutará la función *def add()*.

```
@main.route('/add', methods=['POST'])
def add():

    try:
        id = None
        idPersona = request.form['idPersona']
        username = request.form['username']
        password = request.form['password']
        usuario = Usuario(id,idPersona,username,password)
        ingresos = UsuarioModel.set_entidad(usuario)

        if ingresos >= 1:
            return
redirect(url_for('Usuario_blueprint.home'))
        else:
            return jsonify({'menssage':"Error al
insertar"}),500
    except Exception as ex:
        return jsonify({'mensaje':str(ex)}),500
```

La función *def add()*, es la encargada de controlar las transacciones entre la vista y el modelo para realizar ingresos o registro de nuevos usuarios del sistema.

Mediante el método *POST*, se recibe los datos *idPersona, usarname, password* que son enviados desde la vista o cliente web y

los almacena en las variables del mismo nombre con el método *request.form*, luego los guarda en un objeto llamado *usuario* de tipo clase *Usuario*.

Es importante indicar que se puede guardar los datos que vienen desde la vista en el objeto ingresando el request.form[<<'nombre_variable'>>], directamente sin necesidad de asignar a una variable, pero por motivos académicos se hizo necesario implementarlo de esa manera. Ejemplo:

```
usuario=Usuario(id,request.form['idPersona'],request.form['username'],
request.form['password'])
```

A continuación, en la variable *ingresos* se ejecuta la función *UsuarioModel.set_entidad(usuario)*, que pertenece al modelo y recibe como parámetro el objeto que se construyó anteriormente. Esta función devolverá un valor 1 si logró registrar los valores del objeto en la tabla usuario de la base de datos *seguidad_db*, y *0* si no logró el registro de los valores.

En caso de lograr el registro de los datos o valores del objeto en la tabla *usuario*, el compilador retornará a la página principal de la gestión de usuarios y mostrará el registro ingresado, caso contrario enviará un mensaje tipo *JSON* con el texto "Error al insertar".

De la misma manera esta función manejará los errores con la estructura *try-except*.

En lo que respecta a la edición de datos en la tabla *usuario*, el controlador proporciona una función que tiene la responsabilidad de gestionar los cambios que pueden ocurrir en un determinado usuario del sistema. Por lo general lo que se cambia en este tipo de gestión es el password o contraseña del usuario.

```
@main.route('/edit/<int:id>/<int:idPersona>',
methods=['POST'])
def edit(id,idPersona):
    try:

        username = request.form.get('username')
        password = request.form.get('password')
        if password is None:
            return jsonify({'menssage':"password
vacio"}),404
        else:
            usuario =
Usuario(id,idPersona,username,password)
            ingresos = UsuarioModel.up_Endidad(usuario)

        if ingresos == 1:
            return
redirect(url_for('Usuario_blueprint.home'))
        else:
            return jsonify({'menssage':"Error al
actualizar"}),500
    except Exception as ex:
        return jsonify({'mensaje':str(ex)}),500
```

La ruta de acceso a la función de edición en este caso es: */edit/<id>/<idPersona>*, mediante esta ruta se envían dos parámetros, *id* de la tabla y el *idPersona*, que es el identificador de la persona. Por medio de estos argumentos, la función *UsuarioModel.up_Entidad(usuario)*, identificará al usuario y ejecutará las acciones de actualización de datos en la base de datos; de la misma manera como en la función *UsuarioModel.set_entidad(usuario)*, la función *UsuarioModel.up_Entidad(usuario)* recibe como parámetro un objeto de tipo *Usuario* y retorna un valor 1 si logró actualizar el registro en la tabla *usuario* de la base de datos seguidad_db, y 0 si no logró el objetivo.

En caso de lograr la actualización del registro en la tabla *usuario*, el compilador retornará a la página principal de la gestión de usuarios y mostrará el registro actualizado en el entorno de trabajo correspondiente a la funcionalidad antes mencionada, caso contrario enviará un mensaje tipo JSON con el texto “Error al actualizar”.

Por último, la función *def delete()*, se ejecutará cuando el cliente web referencie la ruta */delete/<id>*.

Código:

```
@main.route('/delete/<int:id>')
def delete(id):
    try:
        if id is None:
```

```
            return jsonify({'menssage':"Id vacio"}),404
        else:
            fila_afectada = UsuarioModel.del_entidad(id)
            if fila_afectada == 1:
                return
redirect(url_for('Usuario_blueprint.home'))
            else:
                return jsonify({'menssage':"Ninguna persona
ha sido eliminada"}),404
    except Exception as ex:
        return jsonify({'mesage':str(ex)}),500
```

Esta función recibe un parámetro entero *id*, que representa el identificador de la fila o tupla de la tabla usuario y que fue seleccionada al momento de dar clic en el botón eliminar, en esta función es obligatorio verificar si la variable *id* tiene valor o esta vacía, ya que si su valor es nulo o vacío, no debe permitir la ejecución del código de su implementación, caso contrario, se ejecutará el código contenido en la función *UsuarioModel.del_entidad(id)* de esta manera devolverá 1 si logró eliminar el registro, caso contrario deberá emitir un mensaje tipo *JSON* con el error “El registro no ha sido eliminado”.

Luego de lograr el objetivo, retornará a la página principal y mostrará todos los registros a excepción del registro eliminado. Antes de ejecutar las acciones descritas, deberá preguntar si desea o no eliminar el registro.

4.3.3. Usuario Modelo (UsuarioModel.py)

El modelo es la capa encargada de comunicarse con la base de datos para realizar transacciones como son: registro, extracción, modificación y eliminación de datos. En lo que respecta a la gestión de usuarios del sistema, se cuenta con un *UsuarioModel*, este tiene como responsabilidad realizar todas las transacciones que se requieren para registrar, extraer, modificar y eliminar datos de los usuarios en la tabla *usuario* de la base de datos *seguridad_db*.

```
from database.db import get_connection
from .entities.Usuario import Usuario
from .entities.UsuarioCompleto import UsuarioCompleto
from .entities.Persona import Persona
from flask import request,render_template
from flask_paginate import Pagination

from flask import render_template,request
from werkzeug.security import generate_password_hash
```

Lo primero que tenemos que importar en este script, es el componente para la conexión con la base de datos. Por esa razón se importa el componente *database.db* que contiene la función *get_connection*, cuya única responsabilidad es generar la conexión a través de la cadena de conexión antes mencionada.

Luego se requiere importar las entidades que nos permitirán instanciar los objetos para la manipulación de datos. Para ello se

importará, *.entities.Usuario*, *.entities.UsuarioCompleto* y *.entities.Persona*, esto nos permitirá instanciar objetos de tipo Usuario con la estructura establecida equivalente a la estructura de la tabla *usuario*, luego se establecerá una estructura especial implementada con el objetivo de manipular datos especiales que resultan de la unión de varias tablas de la base de datos *seguridad_db*. Por último, también ser requiere importar la clase Persona que se usará como comodín especial de presentación de datos legibles para los usuarios, por ejemplo, nombres completos de los usuarios.

Del framework *flask*, se requiere importar los métodos *request y render_template,* para realizar los respectivos direccionamientos de datos a las vistas.

Por último, de *flask_paginate* se importará el método *Pagination* exclusivamente para realizar las paginaciones en las interfaces de usuario.

La clase *UsuarioModel()*, contendrá todas las funciones encargadas de mostrar, registrar, eliminar y modificar de datos en la tabla *usuario*.

@classmethod, es un decorador que permiten realizar acciones para enviar la clase como primer argumento en lugar de la instancia de esa clase. Es decir, que puede utilizar la clase y sus propiedades dentro de ese método sin tener que instanciar.

```
 @classmethod
    def get_page(self):
        num_page = 0
        try:
            db = get_connection()
            cursor = db.cursor()
            cursor = db.cursor(buffered=True)
            cursor.execute("SELECT COUNT(*) FROM `usuario`;")
            total_reg = cursor.fetchone()[0]
            num_page = request.args.get('page',1,type=int)

            zise_page = 3
            inicio = (num_page - 1) * zise_page + 1

            cursor.execute("SELECT id,idPersona FROM
`usuario`;")
            result = cursor.fetchall()
            List_usuarios = []
            for row in result:
                cursor.execute("""SELECT
`usuario`.`id`,`persona`.`id` AS
                 idPersona,`persona`.`strNombres`,
`persona`.`strApellidos`,
                 `usuario`.`username`, `usuario`.`password`
                 FROM `persona` INNER JOIN `usuario` ON
`persona`.`id` = %s
                 AND `usuario`.`id` = %s; """%(row[1],
row[0]))
                res = cursor.fetchone()
                usuarioCompleto =

UsuarioCompleto(res[0],res[1],res[2],res[3],res[4],res[5])
                List_usuarios.append(usuarioCompleto.to_JSON()
)

            cursor.execute(f"SELECT * FROM `usuario` WHERE id
>= 1 ORDER BY
```

```
                            id ASC LIMIT {zise_page} OFFSET
{inicio - 1}")
            result = cursor.fetchall()
            usuarios = []
            for row in result:
                usuario = Usuario(row[0],row[1],row[2],row[3])
                usuarios.append(usuario.__str__())

            fin = min(inicio + zise_page,total_reg)

            if (inicio > total_reg):
                fin = total_reg

            cursor.execute("SELECT * FROM `persona` WHERE
`id`>0")
            result = cursor.fetchall()
            list_personas = []
            for row in result:
                persona = Persona(row[0],row[1],row[2],row[3]

,row[4],row[5],row[6],row[7],row[8])
                list_personas.append(persona.__str__())


            pagination = Pagination(page = num_page,
per_page=zise_page,
            total = total_reg, display_msg = f"Mostrando
registros {inicio}
                        - {fin} de un total de {total_reg}")
            db.close()
            return render_template('gestionusuario.html',dataU
=
                            List_usuarios,dataP =
list_personas, paginacion =
                            pagination)
        except Exception as ex:
            raise Exception(ex)
```

Función *get_page(self)*

Esta función esta implementada para hacer la principal funcionalidad de este módulo, es decir, es la encargada de extraer la información de la tabla *usuario*, presentarla en pantalla y permitir establecer las otras funcionalidades descritas antes (registrar, modificar, eliminar).

Lo primero que hay que hacer, es establecer la conexión a la base de datos, para el efecto se invoca a la función *get_connection()*, y se almacena en la variable *db,* con ello se podrá establecer el *cursor* que se encargará de gestionar las transacciones con la base de datos. Durante la implementación de esta funcionalidad, se requiere conocer varias funciones contempladas en el código que se detallan a continuación:

Pagination: la ejecución de la consulta para contar el número de registros de la tabla *usuario*, permitirá conocer cuántas páginas se van a presentar al usuario, este *num_page* (número de páginas) junto con el tamaño de las páginas por vista (*zise_page*), calculará la variable con que inicia la página de la presentación de datos, aplicando la fórmula; *inicio = (num_page - 1) * zise_page + 1.*

Luego, para el número total de registro utilizaremos la función *min* de *Python* para establecer el fin de la numeración de la paginación, aplicando la formula *fin = min(inicio + zise_page,total_reg)*.

La función *num_page = request.args.get('page',1,type=int)*, se utiliza para usar el atributo del objeto de solicitud para acceder a los parámetros de *URL*. Estos parámetros se agregan al final de esta URL en forma de clave (page) = valor (1 o el número de página que escoja). En consecuencia, la paginación quedará de la siguiente manera cada vez que se requiera navegar por el número de página *http://127.0.0.1:4000/seg/usuario/?page=2.*

Es responsable de presentar la información de manera legible para el usuario, de tal manera que se pueda entender la información que se presenta en la vista principal.

Al final, para establecer la ejecución de la paginación se invoca la instrucción, *pagination = Pagination(page = num_page, per_page=zise_page, total = total_reg, display_msg = f"Mostrando registros {inicio} - {fin} de un total de {total_reg}")*. Los parámetros que se envían por medio de esta instanciación llegarán a la vista para su respectiva presentación.

Por último, esta función enviará varios objetos y/o listas con datos específicos para que sean manipulados en la vista por intermediación del controlador. Es por ello que se generan varios objetos en la ejecución de las distintas instrucciones del script en mención.
Entre los objetos y/o listas se tendrán: List_usuarios, list_personas y paginación.

List_usuarios: es una lista de tipo *UsuarioCompleto* que tiene como argumentos los siguientes parámetros: *id, idPersona, strNombres, strApellidos, username, password*.

Para llenar esta lista se requiere navegar por varias tablas de la base de datos *seguridad_db*, la primera es la tabla *usuario*, de ella se extraerán todos los *id e idPersona* de los usuarios del sistema. Por cada *id* e *idPersona* de los usuarios, se requiere conseguir los nombres, apellidos, username y password; se creará una consulta anidada donde intervienen las tablas; *persona* (para extraer nombres, apellidos) y *usuario* (para extraer username y password), dentro de un ciclo se irá extrayendo registro por registro almacenándolo en el objeto y añadiéndolo a la lista *List_usuarios*. Esta lista va contener todos los usuarios del sistema con sus datos completos (id, idPersona, Nombres, Apellidos, username, password).

list_personas: lo siguiente es extraer la lista de personas (*list_personas*), esta lista es solamente para llenar información de los nombres y apellidos de las personas en un *select o combobox*. Este procedimiento será utilizado en el registro y actualización de información de la tabla.

Lo que resta, es enviar esta información a la vista. Para ellos se utiliza el método *render_template()*.

Función *set_entidad(self,usuarioCompleto)*

Esta función recibe como parámetro un objeto de tipo *UsuarioCompleto* ya que requiere saber los datos del nuevo *usuario* que se va a registrar en la tabla *usuario* de la base de datos *seguridad_db*.

Esta función tiene la responsabilidad de registrar o insertar nuevos usuarios, siempre y cuando no exista en la tabla *usuario*.

```
@classmethod
    def set_entidad(self,usuarioCompleto):
        try:
            filas_ingresadas=0
            db = get_connection()
            cursor = db.cursor(buffered=True)
            cursor.execute("SELECT id FROM `usuario` WHERE
idPersona = %s;
                           "%(usuarioCompleto.idPersona))
            result = cursor.fetchone()
```

```
            if result is not None:
                return filas_ingresadas
            else:
                with db.cursor() as cursor:
                    sql = """INSERT INTO `usuario`
(`idPersona`, `username`,
                          `password`) VALUES (%s,%s,%s);"""
                    dato =
(usuarioCompleto.idPersona,usuarioCompleto
                           .username,generate_password_hash
                           (usuarioCompleto.password))
                    cursor.execute(sql,dato)
                    filas_ingresadas = cursor.rowcount
                    db.commit()
                    db.close()
                    return filas_ingresadas
        except Exception as ex:
            raise Exception(ex)
```

Como en la función anterior lo primordial es realizar la conexión a la base de datos y generar un cursor que permita ejecutar las transacciones de manera eficiente; luego, se necesita verificar si el usuario, que el cliente web, esta enviando existe en la tabla *usuario* de la base de datos *seguridad_db*, para ello se implementa una consulta de búsqueda con la condición de encontrar aquellos usuarios que poseen el *idPersona = usuarioCompleto.idPersona*, dicho de otra manera, se verifica sí el *idPersona* que están en la tabla *usuario* es igual al *idPersona* que envía el cliente web, en el objeto *usuarioCompleto*. Si el resultado es positivo, el compilador no procede a ejecutar la sentencia de inserción, caso contrario si la ejecuta.

Con la ejecución de la instrucción de inserción (insert), está incluido el método *generate_password_hash(usuarioCompleto.password)*, este método es el encargado de encriptar el password o como habíamos mencionado antes hacer el ***hashed***[6].

Cuando el compilador ejecuta la sentencia de inserción, el cursor verifica el número de columnas insertadas y lo almacena en la variable *filas_ingresadas*, luego, con la función *set_entidad* retornará el valor de esa variable al controlador.

Función *del_entidad(self,id)*

Toda implementación de un proyecto de software requiere un control de eliminación, esta función esta implementada para ejecutar ese requerimiento. La función *del_entidad(self,id)*, recibe como parámetro el *id* del usuario del sistema que seleccionó en la vista principal o cliente web.

```
@classmethod
    def del_entidad(self,id):
        try:
            db = get_connection()
            with db.cursor() as cursor:
                sql = "DELETE FROM `usuario` WHERE `id`=%s"
```

[6] Proceso de convertir un dato a un valor de longitud fija encriptado en MD5, SHA-256 o bcrypt.

```
                dato = (id,)
                cursor.execute(sql,dato)
                fila_afectada = cursor.rowcount
                db.commit()
            db.close()
            return fila_afectada
        except Exception as ex:
            raise Exception(ex)
```

Nuevamente, es importante establecer la conexión a la base de datos mediante la función *get_connection* para establecer el cursor, mediante la condición *`id`=id* se verifica si el usuario que envía el cliente web, existe en la tabla *usuario*, si la respuesta es positiva, el compilador procede a eliminar de la tabla *usuario* el registro que cuente con ese *id*. Cuando el compilador ejecuta la sentencia de eliminación, el *cursor* verifica el número de columnas eliminadas y lo almacena en la variable *filas_afectadas*, luego la función *del_entidad* retornará el valor de esa variable al controlador.

Función *up_Entidad(self,usuario)*

Es la función destinada a ejecutar las modificaciones que el rol con permisos de gestión de usuarios del sistema, ha ejecutado sobre un registro determinado. En esta ocasión la función recibirá un objeto de tipo *usuario*, ya que lo único que podrá cambiar aquí es la contraseña, por ende, no se requiere más información que el *idPersona* y el *password* que desea actualizar.

```
@classmethod
    def up_Endidad(self,usuario):
        try:
            db = get_connection()
            with db.cursor() as cursor:
                sql = """UPDATE `usuario` SET `password`=%s
                    WHERE
`idPersona`=%s"""
                dato =
(generate_password_hash(usuario.password)
                        ,usuario.idPersona)
                cursor.execute(sql,dato)
                fila_actualizada = cursor.rowcount
                db.commit()
            db.close()
            return fila_actualizada
        except Exception as ex:
            raise Exception(ex)
```

La función realiza su respectiva conexión a la base de datos y genera el *cursor*; luego, ejecuta las instrucciones para la de actualización del registro, dependiendo si existe el *idPersona* que fue enviado desde el cliente web o capa de vista; es importante indicar que si el *idPersona* enviado concuerde con el *idPersona* de la tabla *usuario*, el compilador procederá a ejecutar la función *generate_password_hash()* del método *security.py*, con la finalidad de cifrar la contraseña antes de almacenarla en la base de datos.

Cuando el compilador ejecuta la sentencia de actualización, el *cursor* verifica el número de columnas actualizadas y lo almacena en la

variable *fila_actualizada*, luego la función *up_Entidad* retornará el valor de esa variable al controlador.

4.3.4. Gestión usuario (gestionusuario.html)

En relación a la capa de presentación que en este patrón arquitectónico *MVC* es conocido como vista (*View*), es la encargada de visualizar toda la funcionalidad que ha sido implementada en las capas anteriores.

Figura 23

Gestión de usuarios.

Nota. Elaborado por los autores.

La gestión de usuarios va a consistir en un listado de los usuarios del sistema junto a su nombre de usuario y su password, así como los eventos funcionales para el registro de nuevos usuarios, edición y la eliminación de usuario.

Nota: Por motivos académicos esta vista, muestra el contenido del parámetro password, con el único objetivo de presentar al lector como se almacenan esos valores en la base de datos.

Lo primero que hay que hacer a la hora de utilizar los estilos de bootstrap, es invocar el framework en el documento *HTML*, para esto, se cuenta con el procedimiento de agrega vía ***CDN*** las dos líneas de instrucciones que se encuentran en la página oficial de bootstrap, luego cuestión de copiar y pegar dentro de la etiqueta *<head>.* Este procedimiento permitirá utilizar todas las clases que han sido implementadas en el framework *css* de bootstrap sin necesidad de una instalación previa.

```
<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">
    <title>Gestión de usuarios</title>
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-
          alpha1/dist/css/bootstrap.min.css"
rel="stylesheet"
```

```
            integrity="sha384-

GLhlTQ8iRABdZLl6O3oVMWSktQOp6b7In1Zl3/Jr59b6EGGoI1aFkw7cmDA
6j6gD"
            crossorigin="anonymous">
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-
            alpha1/dist/js/bootstrap.bundle.min.js"
integrity="sha384-

w76AqPfDkMBDXo30jS1Sgez6pr3x5MlQ1ZAGC+nuZB+EYdgRZgiwxhTBTkF
7CXvN"
            crossorigin="anonymous"></script>
</head>
```

A continuación, dentro de la etiqueta *<body>*, estableceremos toda la vista (*View*) que queremos mostrar al cliente web. Estas están divididas en tres partes: el contenedor del botón nuevo, el contenedor de la tabla donde están la lista de datos y el contenedor de la paginación.

Contenedor del botón Nuevo

Es un espacio destinado a mostrar el botón de acción "Nuevo", este botón permite gestionar el registro de datos del usuario a la tabla *usuario*, dicho de otra manera, es el que permite el ingreso por teclado, de los datos del usuario para luego enviarlos al controlador.

La siguiente sección dividirá la página *HTML* en partes mediante la utilización de la clase *"card-body"*, un recuadro o tarjeta con bordes

y un margen superior, esto evitará que estén muy pegados al margen de la página.

```
<div class="card-body">
```

Es importante indicar que esta porción de código *HTML*, va a permitir varias acciones, como dibujar un botón con el nombre de *"Nuevo"*; luego, establecerá un modal para presentar una ventana emergente para el ingreso de datos del usuario.

La etiqueta *<button>,* permitirá establecer el botón dentro del * * antes mencionado y lanzará una ventana emergente mediante un modal. Estas acciones se pueden realizar mediante el uso de los parámetros *data-bs-toggle="modal"* y *data-bs-target="#modal"*.

El argumento *data-bs-toggle*, define el tipo de recuadro que se va a lanzar, mientras que el argumento *data-bs-target* especifica el identificador del modal, es decir, el nombre con el que será conocido posteriormente.

```
<td><button class="btn btn-primary btn-sm" id="btn-primary"
data-bs-
                    toggle="modal" data-bs-
target="#modal">Nuevo</button></td>
<br>
```

El modal es una estructura tipo form, que permite dibujar una tarjeta o recuadro que contenga los parámetros que se requieren enviar al controlador para su almacenamiento en la tabla *usuario* de la base de datos *seguridad_db*.

```
<!--Modal de ingreso-->
  <div class="modal fade" id="modal" tabindex="-1" aria-
                  labelledby="exampleModalLabel" aria-
hidden="true">
     <div class="modal-dialog">
       <div class="modal-content">
         <div class="modal-header">
           <h1 class="modal-title fs-5"
id="exampleModalLabel">Agregar
                Usuario</h1>
           <button type="button" class="btn-close" data-bs-
dismiss="modal" aria-
                   label="Close"></button>
          </div>
          <div class="modal-body">
          <form action="/seg/usuario/add" method="post">
                <div class="row mb-3">
                   <label> Nombre completo</label>
                   <select name="idPersona" class="form-
select" aria-
                           label="Default select example" >
                     {% for dp in dataP %}
                     <option  value="{{dp.id}}">{{dp.strNombr
es}}
                              {{dp.strApellidos}}</option>
                     {% endfor %}
                   </select> <br>
                   <label>Nombre de Usuario</label>
                   <input type="text" class="from-control mb-
3"
                          name="username">
```

```
                    <br>
                    <label>Contraseña</label>
                    <input type="password" class="from-control
mb-3"

name="password">

                </div>
                <div class="modal-footer">
                  <button type="submit" class="btn btn-
primary">Guardar</button>
                  <button type="button" class="btn btn-
primary" data-bs-
                          dismiss="modal">Cancelar</button>
                </div>
                </form>
               </div>
              </div>
```

Por lo tanto, la *class="modal fade"*, está compuesto por dos argumentos fundamentales; el argumento *modal*, que permite activar las funcionalidades de los modales o diálogos y el argumento *fade* que permite al modal o diálogo aparecer lentamente en la pantalla.

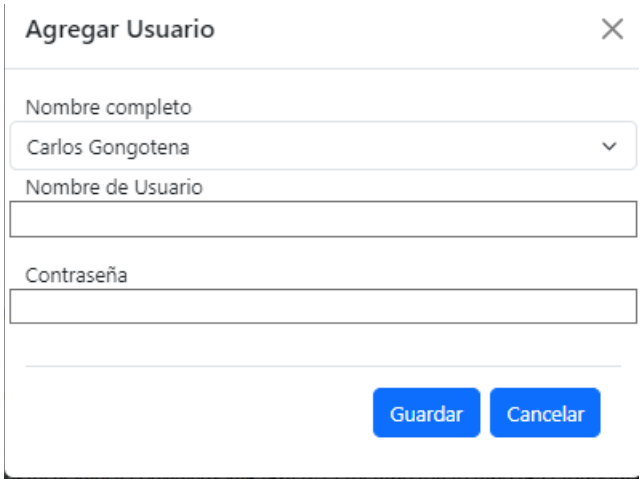
Otro elemento importante de esta etiqueta es *id*, que permite establecer un identificador o nombre único al modal o diálogo. Este identificador permitirá establecer conexiones al momento de accionar el botó *"Nuevo"*.

A esta acción se le llama lanzamiento del modal. Con el modal presente en la pantalla del cliente web, las etiquetas HTML están

obligados a dibujar los diferentes elementos para el ingreso de datos en un formulario que está dispuesto dentro de dicho modal.

Figura 24

Agregar Usuario.



Nota. Elaborado por los autores.

La descripción del modal es “Agregar usuario” y está compuesto por un <*select*> y dos <*input*>. El <*select*> que está identificado con el *name = idPersona* y destinado a la selección de los nombres y apellidos del usuario, con ello mediante la utilización de codificación *jinja2*, se consume la lista de usuarios que es enviada desde la capa *models* y recorrida mediante un ciclo *for* para ser almacenada en una variable *dp* que contiene los datos de los usuarios, esta información es agregada ordenadamente en la etiqueta <*option*> del <*select*>. Este procedimiento es implementado para que desde el cliente web pueda

seleccionar el nombre completo del usuario. Por cada selección, el compilador colocará en *value* el *id* del usuario seleccionado.

Tanto el *name* del *<select>* como el *value* de la etiqueta *<option>* son de suma importancia para el controlador, ya que mediante esa variable se puede extraer el valor que contiene la opción seleccionada. Es decir, que mediante el método *POST*, la vista envía la variable *name* del *<select>* y el *value* de *<option>* al controlar para manipular los datos en las diferentes capas del patrón *MVC*.

Luego, el formulario que está dentro del modal, tiene dos formularios tipo *<input>*, el primero está destinado para el ingreso del "*Nombre de usuario"* (su identificador o *name* es *username*), mientras que el otro *<input>*, es para el ingreso de la contraseña (su identificador o *name* será *password*). Estos dos elementos se comportan de manera diferente, mientras el *<select>* requiere escoger un valor, en los *<input>* necesita ingresar un valor; pero ambas, van a contener un valor que será enviado al constructor por medio del método *POST*.

Al pulsar el botón "Guardar" de tipo "*submit"*, el contenido de la etiqueta que está dentro del *form*, tendrá todas las variables viajarán al controlador (UsuarioController) vía *POST* y buscarán la ruta "*/seg/usuario/add*", gracias al parámetro *"accion"* de la etiqueta *form*.

Contenedor de la tabla donde están la lista de datos

Esta sección es la encargada de mostrar los datos de la tabla ya formateados de una manera legible para el cliente web; además, por cada registro que la interfaz muestre llevará consigo los botones de acción “Editar” y “Eliminar”.

Mediante la etiqueta *<table>*, se establecerá una estructura ordenada para colocar la información que el controlador envíe a la vista. Luego, con la clase *mx-auto* se centrará horizontalmente el contenido de nivel de ancho fijo, además la clase *table-bordered* mostrará los bordes en todos los lados de la tabla y las celdas. Con estos estilos la tabla tendrá una vista más amigable para el cliente.

```
<table class="table mx-auto text-center table-bordered">
```

Lo primero que se va a dibujar es el encabezado de la tabla que por lo general es estática y permitirá establecer un esquema para los datos dinámicos que se irán dibujando bajo de ella. Por esa razón se utilizarán dos etiquetas que forman parte de la estructura de *<table>*, la etiqueta *<thead>* que es la encargada de contener la fila y columnas estáticas que constituirán el encabezado de la tabla.

```
<thead>
   <th scope="col">#</th>
   <th scope="col">Nombre completo</th>
```

```
    <th scope="col">Nombre de usuario</th>
    <th scope="col">Contraseña</th>
    <th scope="col">Edición</th>
    <th scope="col">Eliminación</th>
</thead>
```

Y la etiqueta *<tbody>*, que es la encargada de contener las filas y columnas que son parte del cuerpo de la tabla o parte dinámica, requiere tener una estructura repetitiva que permita ir dibujando fila por fila la información que el controlador le envía para su presentación al cliente; además, necesita usar código *jinja2* para poder establecer esta estructura.

Mediante el uso de *Jinja2*, *HTML* puede contener estructuras para enviar mensajes y todas las instrucciones que sean necesarias para mostrar los datos al cliente web. En eta ocasión se utilizará una estructura de repetición *for* para recorrer la lista de objetos tipo *UsuarioCompleto*, que es enviada desde el modelo.

```
<tbody>
  {% for du in dataU %}
    <tr>
     <td>{{du.id}}</td>
     <td>{{du.strNombres}} {{du.strApellidos}}</td>
     <td>{{du.username}}</td>
     <td>{{du.password}}</td>
```

A través de la variable *du*, se recorre la lista de objetos *dataU*, esta lista contiene información estructurada en un tipo

UsuarioCompleto, que contiene: Nombres, Apellidos, Nombre de usuario, contraseña.

La etiqueta *<tbody>* contiene el artificio *<tr>*, que dibuja una fila, este argumento es dinámico ya que va a ir dibujando una fila cada vez que el ciclo *for* dispare un registro. Por cada dato que la lista de objeto entregue a la variable *du*, se dibujará una columna por medio del argumento *<td>*, de esta manera se dibuja la tabla sobre el documento *HTML*.

Como se mencionó anteriormente, por cada fila de registro presentado, se encontrará dos botones de acción, "Editar" y "Eliminar". El botón "Editar", por su esencia funcional requiere de un modal que permita al cliente ingresar el o los datos que necesitan ser actualizados, de esta manera, al presionar este botón automáticamente se mostrará un modal con los parámetros necesarios para poder actualizar la tabla *usuario* en la base de datos.

```
<td><button class="btn btn-primary btn-sm" id="btn-
edit{{du.id}}"
                data-bs-toggle="modal" data-bs-
                target="#modal{{du.id}}">Editar</button></td>
```

Esta funcionalidad es parecida a la que se implementó en el botón "Nuevo" para ingresar registros a la tabla, con la diferencia que en esta ocasión se requiere enviar un parámetro para que el modal

identifique a que usuario del sistema se está refiriendo. Es decir, en el parámetro *id="btn-edit{{du.id}}"* se envía el *id* o identificar de la fila que seleccionó el cliente web, para realizar la actualización o edición; luego, el modal envía esa información a la ruta */seg/usuario/edit/{{du.id}}/{{du.idPersona}}* en el controlador.

Para completar la configuración del modal del botón *Editar*, se requiere especificar en los argumentos *data-bs-toggle="modal" data-bs-target="#modal{{du.id}}"*, el tipo de ventana emergente y su nombre para ligarlo con el *modal*, de la misma manera se envía el *id* de la fila seleccionada.

```
<div class="modal fade" id="modal{{du.id}}" tabindex="-1"
aria-
                        labelledby="exampleModalLabel" aria-
hidden="true">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <h1 class="modal-title fs-5"
id="exampleModalLabel">Edición de
                                    Usuarios</h1>
        <button type="button" class="btn-close" data-bs-
dismiss="modal"
                         aria-label="Close"></button>
      </div>
      <div class="modal-body">
        <form
action="/seg/usuario/edit/{{du.id}}/{{du.idPersona}}"
                    method="post">
        <div class="row mb-3">
```

```
          <label>Nombre completo</label>
          <input type="text" class="from-control mb-3"
name="idPersona"
                    value="{{du.strNombres}}
{{du.strApellidos}}" disabled>
          <br>
          <label>Nombre de usuario</label>
          <input type="text" class="from-control mb-3"
name="username"
                    value="{{du.username}}" >
          <br>
          <label>Contraseña</label>
          <input type="password" class="from-control mb-3"
name="password">
          </div>
                        <div class="modal-footer">
           <button type="submit" class="btn btn-
primary">Guardar </button>
           <button type="button" class="btn btn-primary" data-
bs-
                   dismiss="modal">Cancelar</button>
          </div>
         </form>
        </div>
       </div>
      </div>
     </div>
     {% endfor %}
```

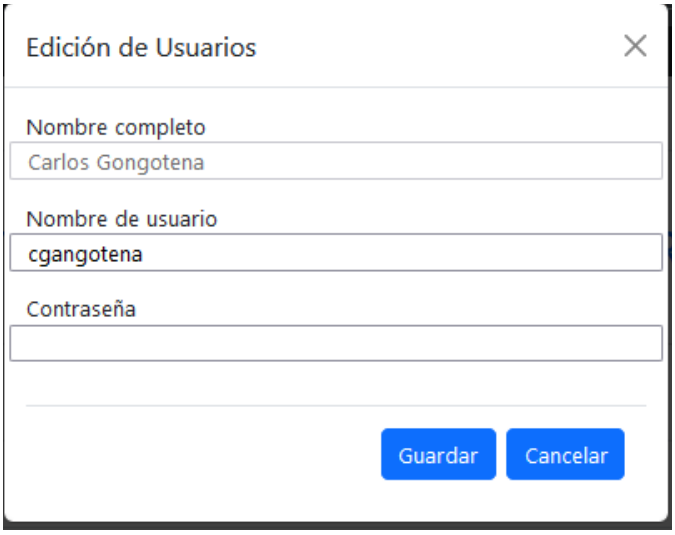
De esta manera, al construir el modal hay que tener en cuenta que el argumento *id* lleve el mismo valor con el que fue declarado en el botón "*Editar*", en ese contexto el argumento *id="modal{{du.id}}"*, ligará el botón con el modal y así al pulsar dicho botón el cliente web se presentará el modal con la leyenda "Edición de Usuarios", junto a

los elementos Nombre completo, Nombre de usuario, Contraseña ya con la información que se extrajo de la variable *du* en sus respectivos *inputs*.

Con el ingreso de los valores de parte del usuario y luego de pulsar el botón “Guardar”, el modal desaparecerá y enviará toda la información del usuario del sistema, al controlador especificando la ruta */seg/usuario/edit/{{du.id}}/{{du.idPersona}}* y los datos, luego el controlador tomará esos datos y los enviará a la función *up_Endidad* del modelo, con ello se completa todo el recorrido del patrón arquitectónico del módulo.

Figura 25

Edición de usuarios.



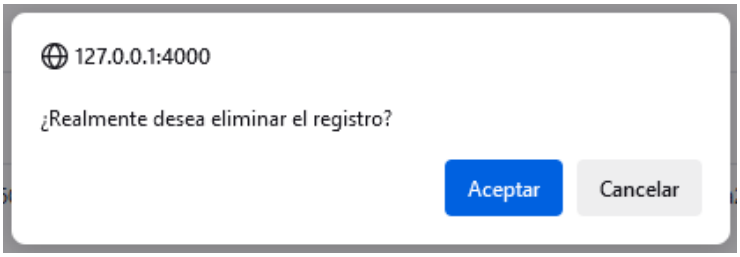
Nota. Elaborado por los autores.

A continuación, en la columna de acciones para el botón "*Eliminar*", se requiere establecer un script mediante la función *onclik()* para confirmar si se requiere eliminar o no dicho registro. El camino del "*si*", contempla la ejecución de la función *del_entidad()* en el modelo, pero antes, mediante las funciones de blueprint envía desde la vista al controlador la ruta "*Usuario_blueprint.delete*" que no es más que "*/seg/usuario/delete/*", y con el argumento *id=du.id*, la ruta se completaría de la siguiente manera; "*/seg/usuario/delete/{{du.id}}*".

```
<td><a onclick="return confirm('¿Realmente desea eliminar
el
                registro?')" href="{{
url_for('Usuario_blueprint.delete',id =
                du.id)}}" class="btn btn-danger btn-
sm">Eliminar</a></td>
    </tr>
```

Figura 26

Ventana de confirmación.



Nota. Elaborado por los autores.

Contenedor de la paginación

Por último, se requiere visualizar la paginación que de la misma manera fue enviada mediante un objeto llamado "*paginacion*", en este objeto viaja desde el modelo al controlador el número de páginas, el número de registros por páginas, la leyenda y las url de navegación.

```
<div class="row justify-content-md-center">
          <div class="col-md-auto">
            <span>
              {{ paginacion.info }}
              <hr />
            </span>
            {{ paginacion.links }}
          </div>
        </div>
```

En ese contexto, el objeto *paginacion.info* se utilizará para obtener la información de paginación como número de páginas, el número de registros por páginas y la leyenda; mientras que *paginacion.links* se obtendrá, todos los enlaces de paginación es decir, las *url* de navegación.

4.4. Implementación del componente Login

La importancia de este componente radica en que es el encargado de verificar la autenticidad de un usuario para poder acceder al sistema, este componente tiene la responsabilidad de permitir o no el acceso a los recursos del sistema.

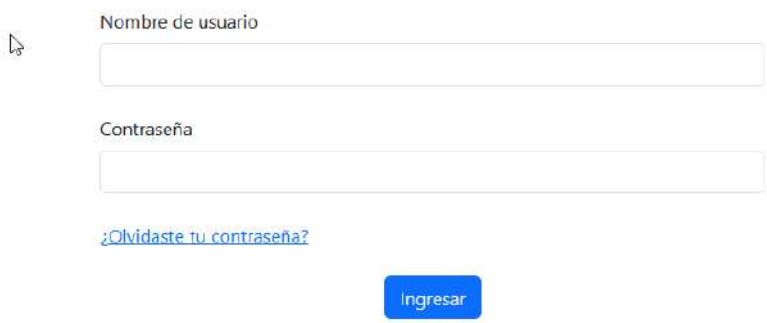
El componente de login, es la puerta de entrada del usuario a los recursos del sistema, para ello es necesario recordar que en las configuraciones de la función principal de ejecución del script *app.py*, la línea que indica al cliente web el ingreso al procedimiento de autenticación y acceso al sistema es *url_prefix = '/'*.

```
app.register_blueprint(LoginController.main,url_prefix='/')
```

Esta línea de código indica al compilador que debe ejecutar la ruta *url_prefix = '/'*, como raíz de la aplicación, y con ello el *LoginController* ejecutará la función ligada a esta ruta, en este caso será la función *def home():* la encarga de iniciar la funcionalidad de acceso.

Figura 27

Formulario de ingreso al módulo.



Nota: Elaborado por los autores.

4.4.1. Controlador del login (LoginController.py)

Para este controlador se va a requerir algunos métodos de *flask* como son:

- Blueprint: para identificar el end-point al momento de instanciar la variable main, de esta manera el sistema sabrá cuál es su ruta principal mediante la etiqueta *'Login_blueprint'*.
- redirect: para ejecutar el redireccionamiento del compilador a la ruta principal, esta acción se realiza junto con *url_for*.
- jsonify: se utilizará para enviar errores que puedan ocurrir en formato JSON.
- request: para trabajar con peticiones *HTTP*, ya sea por el método *POST* o *GET*, este método, captura las peticiones para ser gestionadas en un script determinado.
- render_template: permitirá renderizar las páginas *HTML*.
- sesión: el manejo de variables session es importante para la seguridad de la aplicación en construcción.

```
from flask import
Blueprint,redirect,jsonify,request,url_for,render_template,
flash,session
```

La importación del modelo destinado para este componente es alta, ya que se requiere importar el *LoginModel* para poder tener

conexión con la capa de datos, de la misma manera se requerirá importar la entidad *Usuario*, ya que se usará como tipo de datos para la extracción y validación de usuarios del sistema.

```
from models.LoginModel import LoginModel
from models.entities.Usuario import Usuario
```

Luego y como complemento, hay que instanciar la variable *main* de tipo *Blueprint*, ya que con ella se podrá acceder a los decoradores y sus rutas.

```
main = Blueprint('Login_blueprint',__name__)
```

Para la implementación de este controlador es un puente entre el modelo y la vista, por tal motivo al digitar la ruta: *http://127.0.0.1:4000/*, el compilador ejecutará la ruta especificada *'/'*, esta url esta enrutada mediante el decorador *@main.route('/')* y ligada a la función *home()*, esta función ejecutará un *render_template* para activar la página *"login.html"*.

```
@main.route('/')
def home():
    return render_template('login.html')
```

La función *login*, es la responsable de validar y verificar si los datos ingresados en la página de son las credenciales del usuario del sistema. Este procedimiento requiere de varios aspectos que

necesariamente no se pueden dejar de ejecutar. Por medio del método *request* se capturará el método que desde el cliente web envía la trama *HTTP*, la función está preparada para usar cualquiera de los dos métodos de transferencia de datos *HTTP*, pero el cliente lo envía por *POST*.

En la variable *usuario* tipo objeto *Usuario*, se almacenará la información que el cliente web envió y son capturadas mediante *request.form['username']* y *request.form['password']*, con este objeto *usuario* construido, es enviado como parámetro al modelo mediante la función *login_page(usuario)*, esta función retornará los datos completos de este usuario del sistema y lo asignará a la variable tipo objeto *logeado.*

Es importante asegurar que la variable *logeado* contiene información, ya que en caso de no contener datos el compilador deberá enviar un mensaje de advertencia al usuario y luego redirigir la entrada a la página principal del componente *Login*, es decir, *Login_blueprint.home*.

Por otro lado, si la variable *logeado* contiene información, se deberá comprobar si la contraseña o password corresponde al nombre de usuario, que cuenta con el objeto *logeado*. Como ya se conoce la variable password se encuentra encriptada, por tal motivo hay que ejecutar el método de la clase *Usuario* que permite comparar dos

cadenas encriptadas como es: *__hash__(logeado.password,request.form['password'])*, esta función recibe como parámetros la variable password extraído de la base de datos que esta junto al objeto logeado (*logeado.password*) y la variable request (*request.form['password']*) que envió el cliente web, este método propio del componente *check_password_hash* de la librería *werkzeug.security*, devolverá verdadero o falso.

Cuando el método responda con verdadero (*True*), el compilador creará dos variables de session con los datos del objeto *logeado*, caso contrario, enviará al usuario un mensaje tipo *flask* de password inválido y nuevamente redirigir la entrada a la página principal del componente *Login*, es decir, *Login_blueprint.home*.

```
@main.route('/login',methods = ['GET','POST'])
def login():
    try:
        if request.method=='POST':
            usuario =
Usuario(0,0,request.form['username'],request.form['password
'])
            logeado = LoginModel.login_page(usuario)
            if logeado is not None:
                if
logeado.__hash__(logeado.password,request.form['password'])
:
                    session["username"] = logeado.username
                    session["idPersona"] =
logeado.idPersona
```

```
                    return
LoginModel.get_roles()
                else:
                    flash("Password invalido .....")
                    return
redirect(url_for('Login_blueprint.home'))
            else:
                flash("Usuario no encontrado .....")
                return
redirect(url_for('Login_blueprint.home'))

        else:
            return
redirect(url_for('Login_blueprint.home'))

    except Exception as ex:
        return jsonify({'mensaje':str(ex)}),500
```

En este controlador se van a encontrar varias funciones específicas del componente *Login*, pero que es utilizado por todo el sistema.

La función *permiso*, permite gestionar la lista de permisos que posee un rol determinado, es decir, cuando un usuario ingresa al sistema mediante el cliente web y la ruta raíz, esta le pide que ingrese el nombre de usuario y la contraseña, al acceder a la aplicación el compilador le otorga los roles que tiene este usuario (tabla: personarol) dentro del sistema, estos roles a su vez tienen un conjunto de permisos que se han asignado al o los roles (tabla: rolpermiso).

Esta función recibe un parámetro *id* que representa el identificador del rol, en este caso el método de la trama *HTTP* es el método "*GET*" e invocará a la función *get_permisos(id)*. En caso que no retorne el método *GET* se redirigirá la entrada a la página principal del componente Login, es decir, *Login_blueprint.home*.

```
@main.route('/permiso/<int:id>',methods = ['GET','POST'])
def permiso(id):
    try:
        if request.method=='GET':
            return
LoginModel.get_permisos(id)
        else:
            return
redirect(url_for('Login_blueprint.home'))

    except Exception as ex:
        return jsonify({'mensaje':str(ex)}),500
```

Por último, la implementación de la función logout, que permitirá eliminar la *session* que esté activa del usuario logueado.

```
@main.route('/logout')
def logout():
    if "username" in session:
        session.pop("username")
        return redirect(url_for('Login_blueprint.home'))
```

Es importante indicar que esta implementación no es para nada obligatoria ya que existen muchas otras formas para implementar una solución de este tipo, pero el lector puede usar esta idea como una herramienta de aprendizaje e iniciación en el desarrollo de aplicaciones web con *Python*.

4.4.2. Modelo del login (LoginModel.py)

La capa modelo (*models*) del componente Login, está orientado para ser el responsable de realizar las transacciones con la base de datos, *LoginModel* es el script encargado de gestionar de forma eficiente el envío y recuperación de datos hacia y desde la base de datos *seguridad_db*.

Como el modelo (*models*), es la capa que se encarga de gestionar las transacciones con la base de datos, por esencia tendremos que importar el componente de base de datos y su método *get_connection*, con el objetivo de realizar la conexión con la base de datos *seguridad_db* y las gestiones de transacciones a ella.

```
from database.db import get_connection
```

De la misma manera se deberá importar la librería *flask* y sus métodos *render_template* y *session*.

```
from flask import render_template,session
```

Además, también hay que importar la entidad *Usuario*, misma que será utilizada para manipular los datos que van a ser parte de las transacciones entre el modelo y la base de datos.

```
from models.entities.Usuario import Usuario
```

La función *login_page(self,usuario)*, ejecuta la búsqueda del nombre de usuario, y estos datos colocarlos o asignarlos en una variable de tipo *Usuario* llamada *usuario_db*, es decir extraer información de la base de datos y almacenarla en un objeto, esta información es la que recibe el *LoginController* para saber si existe o no el usuario ingresados por el cliente web desde la página de logueo.

```
class LoginModel():

    @classmethod
    def login_page(self,usuario):
        try:
            db = get_connection()
            cursor = db.cursor()
            cursor = db.cursor(buffered=True)
            cursor.execute("SELECT `id`, `idPersona`,
`username`, `password`
                FROM `usuario` WHERE `username`= '%s';"
%(usuario.username))
            result = cursor.fetchone()
            if result is not None:
                usuario_db
=Usuario(result[0],result[1],result[2],result[3])
                db.close()
                return usuario_db
            else:
```

```
                return None
        except Exception as ex:
            raise Exception(ex)
```

A continuación, se realizará la búsqueda de roles de un determinado usuario del sistema, para ello lo primero que hay que tener en cuenta, es saber si existe una sesión activa, dicho de otra manera, se debe implementar una condición que permita conocer la sesión que esta activa mediante la condición *if "username" in session*. Es importante indicarle a la session que va a almacenar una lista de roles, ya que un usuario puede tener uno o varios roles.

Esa lista de roles se almacenará en otra lista llamada *lista_roles*, y será enviada a través de parámetros por intermedio de una variable llamada *_roles*, además del objeto usuario en una variable *user*, estos parámetros se los recibirá la plantilla '*header.html*'.

```
@classmethod
  def get_roles(self):
    try:
     if "username" in session:
        username = session['username']
        session['lista_roles'] = []
        db = get_connection()
        cursor = db.cursor()
        cursor = db.cursor(buffered=True)
        lista_roles = session['lista_roles']
        #### seleccionar el o los roles
        cursor.execute("SELECT `idRol` FROM `personarol` WHERE
```

```
                        `idPersona`='%s';"
%(session["idPersona"]))
        rRoles = cursor.fetchall()
        for row_rol in rRoles:
          cursor.execute("SELECT `id`, `strNombre` FROM `rol`
                              WHERE `id` = '%s';"
%(row_rol[0]))
          rRolNombre = cursor.fetchall()
          columnas_roles = [column[0] for column in
cursor.description]
          for row_Nombre in rRolNombre:
            lista_roles.append(dict(zip(columnas_roles,row_Nom
bre)))
            session['lista_roles'] = lista_roles
            return render_template('header.html',_roles =
lista_roles,user =
                       username)
     else:
           return render_template('login.html')
  except Exception as ex:
      raise Exception(ex)
```

Identificado el o los roles con los que el usuario cuenta, el siguiente paso es identificar los permisos o perfiles que posee cada uno de estos roles, para ello se requiere una función que a partir del identificador del rol extraer el o los permisos asociados a ese rol.

De la misma manera hay que garantizar que exista una session activa, luego a partir de *id* del rol buscar el o los *id´s* de permisos y con esta información ejecutar una consulta para extraer el o los nombres y la o las *url* de estos permisos o perfiles. Luego almacenar en una lista

llamada *lista_permisos*. Por último, por parámetros se envía a la plantilla "*header.html*".

```
@classmethod
  def get_permisos(self,rol):
     try:
       if "username" in session:
         username = session['username']
         session['lista_permisos'] = []
         db = get_connection()
         cursor = db.cursor()
         cursor = db.cursor(buffered=True)
         lista_permisos = session['lista_permisos']
         #### seleccionar el o los permisos
         cursor.execute("SELECT `IdPermiso` FROM `rolpermiso`
WHERE `idRol` =
                        '%s';" %(rol))
         rPemisos = cursor.fetchall()
         for row_Permiso in rPemisos:
           cursor.execute("""SELECT `strNombre`, `strUrl` FROM
`permisos` WHERE
                             `id` = '%s'; """ %
(row_Permiso[0]))
           rPermiso = cursor.fetchone()
           columnas_permisos = [column[0] for column in
cursor.description]
           lista_permisos.append(dict(zip(columnas_permisos,rP
ermiso)))
         session['lista_permisos'] = lista_permisos
         return render_template('header.html',_permisos =
                  session['lista_permisos'], _roles
                  =  session['lista_roles'],user = username)
      else:
        return render_template('login.html')
    except Exception as ex:
      raise Exception(ex)
```

4.4.3. Gestión del login (login.html)

En la actualidad existe una gran variedad de páginas de acceso que contienen múltiples seguridades y controles para garantizar la información de los usuarios del sistema, la página de login es la primera de un conjunto de páginas destinadas a satisfacer las necesidades del usuario del sistema, esta contiene las acciones necesarias para contener los datos que van a ser validados por las otras capas en su momento.

Esta página está escrita en *html* y formateada con el framework *Bootstrap* para una mejor presentación, como ya se había mencionado, para utilizar las propiedades de *Bootstrap* se requiere copiar la siguiente línea de código.

```
<link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css
/bootstrap.min.css" rel="stylesheet" integrity="sha384-
T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCmG1MXxSR1GAsXEV/Dwwykc2MPK
8M2HN" crossorigin="anonymous">
```

Como se había mostrado en el script del controlador (*LoginController*), existen mensajes que la aplicación enviaba a los usuarios para conocer ciertas validaciones de datos, por ejemplo, si desde el cliente web se digitaba un usuario que no existía, el controlador enviaba el mensaje "*Usuario no encontrado .....*", este

mensaje se presenta en la vista mediante código *jinja2* escrito en la página *login.html*.

```
{% with messages = get_flashed_messages() %}
{% if messages %}
{% for message in messages %}
<div class="alert alert-primary alert-dismissible"
role="alert">
   <strong>{{message}}</strong>
   <button type="button" class="btn-close" data-bs-
dismiss="alert" aria-
                label="Close"></button>
</div>
{% endfor %}
{% endif %}
{% endwith %}
```

La función *get_flashed_messages()* permite extrae todos los mensajes mostrados de la sesión y los devuelve a la página actual. Es decir, los mensajes que se crearon en el controlador viajan junto a la session y son capturados por esta función en la variable "*messages*", luego mediante un ciclo de repetición *for* se recorrerá esta lista y se almacenará por cada ciclo en la variable "*message*", luego es cuestión de publicar la variable "*message"* para que la página les presente su contenido dentro de una etiqueta <*strong*>.

Lo siguiente es dibujar los elementos del formulario para el ingreso de información, el nombre de usuario se utiliza un *input* de tipo

text, con el *id = "username"* y el *name = "username"*, esto con el objetivo de que viaje hasta el controlador con ese nombre.

```
<div class="form-outline mb-4">
   <label class="form-label" for="username">Nombre de
usuario</label>
   <input type="text" id="username" class="form-control"
name="username"/>
</div>
```

De la misma forma, el siguiente elemento del formulario es un *input,* pero este de tipo *password* donde se ingresará la contraseña, con la diferencia que no puedan visualizar los datos que se van digitando, de la misma forma se debe identificar este elemento con el *name = "password"*.

```
<div class="form-outline mb-4">
  <label class="form-label" for="password">Contraseña</label>
  <input type="password" id="password" class="form-control"
name="password"/>
</div>
```

Por último, el botón "*Ingresar*" es el encargado de enviar a la ruta "*/login*" con todos los datos que el usuario ingresó en los elementos antes descritos, además los envía con el método POST al controlador, ya en el compilador buscará la función que está inmediatamente después del decorador *@main.route('/login',methods = ['GET','POST'])* que en este caso es *def login()*.

```
<div class="text-center">
  <button type="submit" class="btn btn-primary btn-block mb-
4">Ingresar</button>
</div>
```

4.5. Implementación del componente Persona

El componente Persona, está orientado a registrar, modificar, eliminar y seleccionar una lista de datos correspondientes a la información completa de una persona, es una tabla destinada a almacenar información como es; cédula, nombres, apellidos, correo, teléfono, entre otros datos.

Figura 28

Gestión de usuarios y personas.

Nota. Elaborado por los autores.

4.5.1. Clase Persona

La clase Persona es una estructura de datos que contiene un conjunto de información referente a una persona, estos datos van desde sus nombres, apellidos, cédula, correo, teléfono hasta su cargo dentro de la empresa.

```
class Persona():
    def
__init__(self,id,strCedula=None,strNombres=None,strApellidos=N
one

,strCorreo=None,strTelefono=None,strDependencia=None,
                  intEstado=None,strCargo=None) -> None:
        self.id = id
        self.strCedula = strCedula
        self.strNombres = strNombres
        self.strApellidos = strApellidos
        self.strCorreo = strCorreo
        self.strTelefono =  strTelefono
        self.strDependencia = strDependencia
        self.intEstado = intEstado
        self.strCargo = strCargo
    def __str__(self) -> str:
        return {
            'id': self.id,
            'strCedula': self.strCedula,
            'strNombres': self.strNombres,
            'strApellidos': self.strApellidos,
            'strCorreo': self.strCorreo,
            'strTelefono': self.strTelefono,
            'strDependencia': self.strDependencia,
            'intEstado': self.intEstado,
            'strCargo': self.strCargo
        }
```

4.5.2. Persona Controlador (PersonaController.py)

Como en los casos anteriores, en el controlador es necesario importar una serie de librerías con sus respectivos métodos.

```
from flask import
Blueprint,redirect,jsonify,request,url_for,session
```

Como en los casos anteriores, se debe importar tanto el modelo *PersonaModel* como la entidad *Persona*, sin olvidar la instanciación de la variable *main* para poder manejar los decoradores de las rutas.

```
from models.PersonaModel import PersonaModel
from models.entities.Persona import Persona
main = Blueprint('Persona_blueprint',__name__)
```

Como se había explicado anteriormente, cada controlador vendrá con un decorador raíz "/" haciendo alusión a la *url_prefix = '/seg/persona'* del *app.register_blueprint* de la función principal del *app.py*. Es decir, que cada vez que el usuario escriba en el cliente web http://127.0.0.1:4000/seg/persona/, el compilador lo redirecciona al decorador *@main.route('/')* y su función *def home()*.

```
@main.route('/')
def home():
    try:
        if "username" in session:
            return PersonaModel.get_page()
        else:
            return
redirect(url_for('Login_blueprint.home'))

    except Exception as ex:
        return jsonify({'mensaje':str(ex)}),500
```

Desde aquí se ejecutará la función *get_page()*, misma que retornará la página *gestionpersona.html* con los datos de la tabla y sus acciones para registrar nuevas personas, editar datos de una persona y eliminar una persona siempre que el usuario del sistema tenga una session activa, caso contrario redireccionará a la página de *login* al compilador.

De la misma forma, el controlador va a contar con un decorador para añadir registros a la tabla *persona* de la base de datos *seguridad_db*, y seguido a este decorador la función *def add()*, encargada de llamar al modelo por intermedio de la función *set_entidad(persona)*, misma que recibirá un objeto de tipo *Persona* para su posterior registro desde el modelo.

```
@main.route('/add', methods=['POST'])
def add():

    try:
```

```
        id = None
        strCedula = request.form['strCedula']
        strNombres = request.form['strNombres']
        strApellidos = request.form['strApellidos']
        strCorreo = request.form['strCorreo']
        strTelefono = request.form['strTelefono']
        strDependencia = request.form['strDependencia']
        intEstado = request.form['intEstado']
        strCargo = request.form['strCargo']
        persona =
Persona(id,strCedula,strNombres,strApellidos,strCorreo,

strTelefono,strDependencia,intEstado,strCargo)
        ingresos = PersonaModel.set_entidad(persona)

        if ingresos >= 1:
            return redirect(url_for('Persona_blueprint.home'))
        else:
            return jsonify({'menssage':"Error al
insertar"}),500
    except Exception as ex:
        return jsonify({'mensaje':str(ex)}),500
```

Mediante la misma lógica de desarrollo basado en el estándar del ***CRUD*** (Create-Read- Update-Delete), se implementa el decorador *@main.route('/delete/<int:id>')* que implementará la función *def delete(id)*. Esta función invocará al modelo mediante la función *del_entidad(id)* que recibe el identificador de la tabla *persona* de la base de datos *seguridad_db* y procede a eliminar el registro. Luego de eliminar el registro retornará a la página principal del componente persona.

```
@main.route('/delete/<int:id>')
def delete(id):
    try:
        if id is None:
            return jsonify({'menssage':"Id vacio"}),404
        else:
            fila_afectada = PersonaModel.del_entidad(id)
            if fila_afectada == 1:
                return
redirect(url_for('Persona_blueprint.home'))
            else:
                return jsonify({'menssage':"Ninguna persona
ha sido eliminada"}),404
    except Exception as ex:
        return jsonify({'mesage':str(ex)}),500
```

Por último, y siguiendo el estándar del *CRUD*, se implementa el decorador *@main.route('/edit/<int:id>')*, que efectuará la función *def edit(id)*. Esta función invocará al modelo mediante la función *up_Entidad(id)* que recibe el identificador de la persona en la tabla *persona* de la base de datos *seguridad_db* y procede a actualizar o modificar el registro. Luego de actualizar el registro retornará a la página principal del componente *persona*.

```
@main.route('/edit/<int:id>', methods=['POST'])
def edit(id):
    try:
        strCedula = request.form['strCedula']
        strNombres = request.form['strNombres']
        strApellidos = request.form['strApellidos']
        strCorreo = request.form['strCorreo']
        strTelefono = request.form['strTelefono']
        strDependencia = request.form['strDependencia']
```

```
        intEstado = request.form['intEstado']
        strCargo = request.form['strCargo']
        persona = Persona(
id,strCedula,strNombres,strApellidos,strCorreo

,strTelefono,strDependencia,intEstado,strCargo)
        ingresos = PersonaModel.up_Endidad(persona)
        if ingresos == 1:
            return redirect(url_for('Persona_blueprint.home'))
        else:
            return jsonify({'menssage':"Error al
actualizar"}),500
    except Exception as ex:
        return jsonify({'mensaje':str(ex)}),500
```

4.5.3. Persona Modelo (PersonaModel.py)

Siguiendo el patrón arquitectónico *MVC*, el siguiente paso es la implementación del modelo para este componente, en este caso se llamará *PersonaModel.py*, a diferencia de los anteriores modelos, aquí se va a importar la entidad *Persona*.

```
from database.db import get_connection
from .entities.Persona import Persona
from flask import request,render_template
from flask_paginate import Pagination
```

La clase *PersonaModel()*, está conformada por las 4 funciones básicas del *CRUD*. La función *get_page()*, es la responsable de extraer la lista de personas con toda su información desde la tabla *persona* de la base de datos *seguridad_db*.

```
@classmethod
    def get_page(self):
        zise_page = 5
        num_page = 0
        try:
            db = get_connection()
            cursor = db.cursor()
            cursor.execute("SELECT COUNT(*) FROM `persona`;")
            total_reg = cursor.fetchone()[0]
            num_page = request.args.get('page',1,type=int)

            zise_page = 3
            inicio = (num_page - 1) * zise_page + 1

            cursor.execute(f"SELECT * FROM `persona` WHERE id
>= 1 ORDER BY id
                               ASC LIMIT {zise_page} OFFSET
{inicio - 1}")
            result = cursor.fetchall()
            personas = []
            for row in result:
                persona =
Persona(row[0],row[1],row[2],row[3],row[4]

,row[5],row[6],row[7],row[8])
                personas.append(persona.to_JSON())
            fin = min(inicio + zise_page,total_reg)
            if (inicio > total_reg):
                fin = total_reg
            pagination = Pagination(page = num_page,
per_page=zise_page, total =
                                    total_reg, display_msg =
f"Mostrando registros
                                    {inicio} - {fin} de un total
de {total_reg}")
            db.close()
            return render_template('gestionpersona.html',data
= personas,
```

```
                                                        paginacion
= pagination)
        except Exception as ex:
            raise Exception(ex)
```

La siguiente implementación es la función *set_entidad(persona)*; esta función recibe un objeto tipo *Persona* con el objetivo de registrar la información de esa persona en la tabla *persona* de la base de datos *seguridad_db*.

```
@classmethod
    def set_entidad(self,persona):
        try:
            db = get_connection()
            with db.cursor() as cursor:
                sql = """INSERT INTO `persona`
(`strCedula`, `strNombres`, `strApellidos`, `strCorreo`,
`strTelefono`,
                `strDependencia`, `intEstado`, `strCargo`)
VALUES (%s,%s,%s,%s,%s,%s,%s,%s);"""
                dato =
(persona.strCedula,persona.strNombres,persona.strApellidos,
persona.strCorreo,persona.strTelefono,persona.strDependenci
a,persona.intEstado,persona.strCargo)
                cursor.execute(sql,dato)
                filas_ingresadas = cursor.rowcount
                db.commit()
            db.close()
            return filas_ingresadas
        except Exception as ex:
            raise Exception(ex)
```

Luego, es el turno para la función *del_entidad(self,id)*, esta función recibe como parámetro el *id* de la *persona* en la tabla y procede a eliminarla luego de la aceptación del usuario del sistema.

```
@classmethod
    def del_entidad(self,id):
        try:
            db = get_connection()
            with db.cursor() as cursor:
                sql = "DELETE FROM `persona` WHERE `Id`=%s"
                dato = (id,)
                cursor.execute(sql,dato)
                fila_afectada = cursor.rowcount
                db.commit()
            db.close()
            return fila_afectada
        except Exception as ex:
            raise Exception(ex)
```

Por último, la función *def up_Entidad(self,persona)*, recibe como parámetro un objeto tipo *Persona* con todos los datos actualizará la tabla *persona* de la base de datos *seguridad_db*.

```
@classmethod
   def up_Endidad(self,persona):
     try:
       db = get_connection()
       with db.cursor() as cursor:
       sql = """UPDATE `persona` SET
`strCedula`=%s,`strNombres`=%s,
          `strApellidos`=%s,`strCorreo`=%s,`strTelefono`=%s,`s
trDependencia`=%s,
```

```
          intEstado`=%s,`strCargo`=%s WHERE `id` =
%s"""
       dato =
(persona.strCedula,persona.strNombres,persona.strApellidos

,persona.strCorreo,persona.strTelefono,persona.strDependencia
               ,persona.intEstado,persona.strCargo,persona.id)
       cursor.execute(sql,dato)
       fila_actualizada = cursor.rowcount
       db.commit()
     db.close()
     return fila_actualizada
   except Exception as ex:
     raise Exception(ex)
```

4.5.4. Gestión Persona (gestionpersona.html)

La vista (View) es la encargada de visualizar toda la funcionalidad que ha sido implementada en las capas anteriores. En esta ocasión esta vista deberá presentar todos los datos informativos de las personas que están registradas en la tabla *persona*, así como los eventos funcionales para el registro de nuevos usuarios, edición y la eliminación de usuario. Como en los casos anteriores hay que instalar el framework *Bootstrap* en el documento *html*.

```
<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">
    <title>Gestión de usuarios</title>
```

```
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-
          alpha1/dist/css/bootstrap.min.css"
rel="stylesheet"
          integrity="sha384-

GLhlTQ8iRABdZLl6O3oVMWSktQOp6b7In1Zl3/Jr59b6EGGoI1aFkw7cmDA
6j6gD"
          crossorigin="anonymous">
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-
          alpha1/dist/js/bootstrap.bundle.min.js"
integrity="sha384-

w76AqPfDkMBDXo30jS1Sgez6pr3x5MlQ1ZAGC+nuZB+EYdgRZgiwxhTBTkF
7CXvN"
          crossorigin="anonymous"></script>
</head>
```

Dentro de la etiqueta <*body*>, estableceremos toda la vista que queremos mostrar al cliente web. La vista está dividida en tres partes: el contenedor del botón nuevo, el contenedor de la tabla donde están la lista de datos y el contenedor de la paginación.

Contenedor del botón Nuevo

Es un espacio destinado a accionar el botón “Nuevo”, este botón permite gestionar el registro de datos de las personas a la tabla *persona*.

La siguiente sección dividirá la página HTML en parte mediante la utilización de la clase *"card-body"*, un recuadro o tarjeta con bordes y un margen superior, esto evitará que este pegado un margen de la página.

```
<div class="card-body">
```

Esta porción de código *HTML*, va a dibujar un botón con el nombre de *"*Nuevo*"*, luego establecerá un modal que mostrará una ventana emergente para el ingreso de datos de la persona.

La etiqueta *<button>,* permitirá establecer el botón dentro del * * antes mencionado y lanzar una ventana emergente mediante un modal. Estas acciones se realizará mediante el empleo de los parámetros *data-bs-toggle="modal"* y *data-bs-target="#modal"*.

```
<td><button class="btn btn-primary btn-sm" id="btn-primary"
data-bs-
                    toggle="modal" data-bs-
target="#modal">Nuevo</button></td>
<br>
```

Como ya se mencionó, este modal es una estructura tipo *form*, que permite dibujar una tarjeta o recuadro que contenga los parámetros que se requieren enviar al controlador para su almacenamiento en la tabla *persona* de la base de datos *seguridad_db*.

```
<!--Modal de ingreso-->
<div class="modal fade" id="modal" tabindex="-1" aria-
                              labelledby="exampleModalLabel"
aria-hidden="true">
   <div class="modal-dialog">
     <div class="modal-content">
        <div class="modal-header">
          <h1 class="modal-title fs-5"
id="exampleModalLabel">Agregar
                          Usuario</h1>
          <button type="button" class="btn-close" data-bs-
dismiss="modal" aria-
                         label="Close"></button>
        </div>
       <div class="modal-body">
          <form action="/seg/persona/add" method="post">
            <div class="row mb-3">
              <label>CÉDULA</label>
              <input type="text" class="from-control mb-3"

name="strCedula">
              <label>NOMBRES</label>
              <input type="text" class="from-control mb-3"

name="strNombres">
              <label>APELLIDOS</label>
              <input type="text" class="from-control mb-3"

name="strApellidos">
              <label>CORREO</label>
              <input type="text" class="from-control mb-3"

name="strCorreo">
              <label>TELÉFONO</label>
              <input type="text" class="from-control mb-3"

name="strTelefono">
              <label>DEPENDENCIA</label>
```

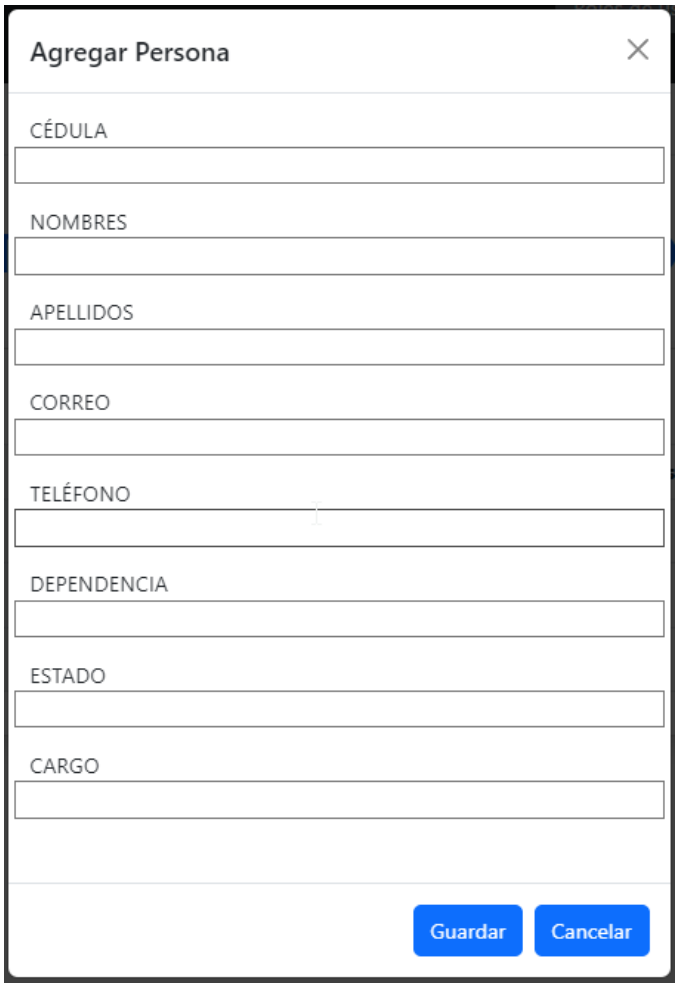
```
              <input type="text" class="from-control mb-3"

name="strDependencia">
              <label>ESTADO</label>
              <input type="text" class="from-control mb-3"

name="intEstado">
              <label>CARGO</label>
              <input type="text" class="from-control mb-3"
name="strCargo">
       </div>
     </div>
   <div class="modal-footer">
     <button type="submit" class="btn btn-
primary">Guardar</button>
     <button type="button" class="btn btn-primary" data-bs-
                    dismiss="modal">Cancelar</button>
   </div>
  </form>
 </div>
</div>
```

La etiqueta *id*, permite establecer un identificador o nombre único del modal o diálogo, con este identificador activar el modal luego de pulsar el botón “Nuevo”.

Con el modal presente en la pantalla del cliente web, las etiquetas *HTML* están obligados a dibujar los diferentes elementos para el ingreso de datos en un formulario que está dispuesto dentro de dicho modal.

Figura 29

Agregar persona.



Nota. Elaborado por los autores.

El modal lleva el nombre “Agregar persona” y está compuesto por ocho elementos *input* dentro de un formulario, cada elemento del formulario tiene su argumento *name* bien identificado, esto para que

el controlador lo pueda identificar de manera única y extraer los datos que en el existe. Pero todos siempre van a contener un valor que será enviado al constructor por medio del método *POST*.

Al pulsar el botón "Guardar" de tipo "*submit"* contenido en la etiqueta que está dentro del *form*, todas las variables viajarán al controlador (*PersonaController*) vía *POST* y buscarán la ruta "*/seg/persona/add"*, gracias al parámetro *"accion"* de la etiqueta *form*.

Contenedor de la tabla donde están la lista de datos

Esta sección muestra los datos de la tabla *persona* formateados de una manera legible para el cliente web; además, por cada registro que la interfaz muestre llevará consigo los botones de acción "Editar" y "Eliminar".

En la etiqueta <*table*>, se establecerá una estructura ordenada para colocar la información que el controlador envíe a la vista. Luego con la clase *mx-auto* permitirá centrar horizontalmente contenido de nivel de bloque de ancho fijo, además la clase *table-bordered* mostrará los bordes en todos los lados de la tabla y las celdas. Con estos estilos la tabla tendrá una vista más amigable para el cliente.

```
<table class="table mx-auto text-center table-bordered">
```

Lo siguiente es dibujar es el encabezado de la tabla que por lo general es estática. Para ello se utilizarán dos etiquetas que forman parte de la estructura de *<table>*, la etiqueta *<thead>* que es la encargada de contener la fila y columnas estáticas que son el encabezado de la tabla.

```
<thead>
  <th scope="col">#</th>
  <th scope="col">Cédula</th>
  <th scope="col">Nombres</th>
  <th scope="col">Apellidos</th>
  <th scope="col">Correo</th>
  <th scope="col">Teléfono</th>
  <th scope="col">Dependencia</th>
  <th scope="col">Estado</th>
  <th scope="col">Cargo</th>
  <th scope="col">Edición</th>
  <th scope="col">Eliminación</th>
</thead>
```

Y la etiqueta *<tbody>*, que es la encargada de contener las filas y columnas que son parte del cuerpo de la tabla, por lo general es la parte dinámica. Por su esencia dinámica, esta etiqueta requiere tener una estructura repetitiva que permita ir dibujando las filas que el controlador le envía para su publicación al cliente web; además, necesita usar código *jinja2* para poder establecer esta estructura.

Mediante el uso de *Jinja2*, *HTML* puede contener estructuras, puede enviar mensajes y todas las instrucciones que sean necesarias

para mostrar los datos al cliente. En esta ocasión se utilizará un *for* para recorrer la lista de objetos tipo *Persona*, que es enviada desde el modelo.

```
{% for d in data %}

<tr>
  <td>{{d.id}}</td>
  <td>{{d.strCedula}}</td>
  <td>{{d.strNombres}}</td>
  <td>{{d.strApellidos}}</td>
  <td>{{d.strCorreo}}</td>
  <td>{{d.strTelefono}}</td>
  <td>{{d.strDependencia}}</td>
  <td>{{d.intEstado}}</td>
  <td>{{d.strCargo}}</td>
```

A través de la variable *d*, se recorre la lista de objetos *data*, esta lista contiene información estructurada en un tipo *Persona*, como son; cédula, Nombres, Apellidos, correo, teléfono, dependencia, estado, cargo; de las personas que están registrados en la tabla *persona* de la base de datos *seguridad_db*.

La etiqueta *<tbody>* contiene el artificio *<tr>*, que permite dibujar una fila, este argumento es dinámico ya que va a ir dibujando una fila cada vez que el ciclo *for* dispare un registro. Cada fila de registro presentado, se encontrará dos botones de acción, “Editar” y “Eliminar”.

El botón “Editar”, por su esencia funcional requiere de un modal que permita al cliente ingresar el o los datos que necesitan ser actualizados, por tal motivo al presionar este botón automáticamente se mostrará un modal con los parámetros necesarios para poder actualizar la tabla *persona* de la base de datos.

```
<td><button class="btn btn-primary btn-sm" id="btn-
edit{{d.id}}" data-bs-toggle="modal" data-bs-
target="#modal{{d.id}}">Editar</button></td>
```

Esta funcionalidad requiere enviar un parámetro para que el modal identifique a que usuario del sistema se está refiriendo. Es decir, en el parámetro *id="btn-edit{{d.id}}"* se envía el *id* o identificar de la fila que seleccionó para realizar la actualización o edición, de esta manera el modal envía esa información a la ruta */seg/persona/edit/{{d.id}}* en el controlador.

Para completar la configuración del *modal* del botón editar se requiere especificar en los argumentos *data-bs-toggle="modal" data-bs-target="#modal{{d.id}}"*, el tipo de ventana emergente y su nombre para ligarlo con el *modal*, de la misma manera se envía el *id* de la fila seleccionada.

```
<!--Ventana Modal-->
<div class="modal fade" id="modal{{d.id}}" tabindex="-1" aria-
```

```
                    labelledby="exampleModalLabel" aria-
hidden="true">
   <div class="modal-dialog">
      <div class="modal-content">
         <div class="modal-header">
            <h1 class="modal-title fs-5"
id="exampleModalLabel">Edición de
                    persona</h1>
            <button type="button" class="btn-close" data-bs-
dismiss="modal"
                    aria-label="Close"></button>
      </div>
      <div class="modal-body">
        <form action="/seg/persona/edit/{{d.id}}"
method="post">
          <div class="row mb-3">
            <label>Cédula</label><br>
            <input type="text" class="from-control from-
control-xl"
                    name="strCedula"
value="{{d.strCedula}}">  <br>
            <label>Nombres</label><br>
            <input type="text" class="from-control mb-3"
name="strNombres"
                    value="{{d.strNombres}}"> <br>
            <label>Apellidos</label><br>
            <input type="text" class="from-control mb-3"
name="strApellidos"
                    value="{{d.strApellidos}}"><br>
            <label>Correo</label><br>
            <input type="text" class="from-control mb-3"
name="strCorreo"
                    value="{{d.strCorreo}}"><br>
            <label>Telefono</label><br>
            <input type="text" class="from-control mb-3"
name="strTelefono"
                    value="{{d.strTelefono}}">  <br>
            <label>Dependencia</label><br>
```

```
            <input type="text" class="from-control mb-3"
name="strDependencia"
                    value="{{d.strDependencia}}"> <br>
            <label>Estado</label><br>
            <input type="text" class="from-control mb-3"
name="intEstado"
                     value="{{d.intEstado}}"><br>
            <label>Cargo</label><br>
            <input type="text" class="from-control mb-3"
name="strCargo"
                     value="{{d.strCargo}}"> <br>
          </div>
        </div>
        <div class="modal-footer">
          <button type="submit" class="btn btn-
primary">Guardar</button>
          <button type="button" class="btn btn-primary" data-
bs-
                    dismiss="modal">Cancelar</button>
        </div>
      </form>
    </div>
  </div>
</div>
{% endfor %}
```

De esta manera cuando el cliente ingrese los valores que se requieren cambiar y pulse el botón “Guardar”, el *modal* desaparecerá y enviará toda la información del usuario del sistema al controlador especificando la ruta */seg/persona/edit/{{du.id}}* y los datos necesarios al controlador, luego los enviará a la función *up_Endidad* del modelo, con ello se completa todo el recorrido del patrón arquitectónico del módulo.

Figura 30

Edición de persona.

Nota. Elaborado por los autores.

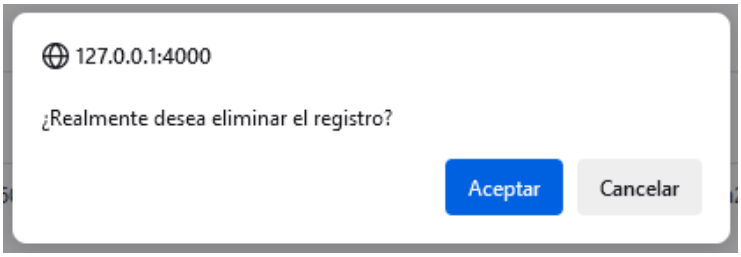
Mientras tanto, en la columna de acciones para el botón eliminar se requiere establecer un script mediante la función *onclik()* para confirmar si se requiere eliminar o no dicho registro.

El camino del "*si*", contempla la ejecución de la función *del_entidad()* en el modelo, pero antes, mediante las funciones de blueprint envía desde la vista al controlador la ruta "*Usuario_blueprint.delete*" que no es más que "*/seg/persona/delete/*", y con el argumento *id=d.id*, la ruta se completaría de la siguiente manera; "*/seg/ persona /delete/{{d.id}}*".

```
<td><a onclick="return confirm('¿Realmente desea eliminar
el registro?')" href="{{
url_for('Persona_blueprint.delete',id = d.id) }}"
class="btn btn-danger btn-sm">Eliminar</a></td>
```

Figura 31

Ventana de confirmación.



Nota. Elaborado por los autores.

Contenedor de la paginación

Por último, se requiere visualizar la paginación que de la misma manera fue enviada mediante un objeto llamado "*paginacion*", en este

objeto viaja desde el modelo al controlador el número de páginas, el número de registros por páginas, la leyenda y las *url* de navegación.

```
<div class="row justify-content-md-center">
          <div class="col-md-auto">
            <span>
              {{ paginacion.info }}
              <hr />
            </span>
            {{ paginacion.links }}
          </div>
        </div>
```

En ese contexto, el objeto *paginacion.info* se utilizará para obtener la información de paginación como número de páginas, el número de registros por páginas y la leyenda; mientras que *paginacion.links* obtendrá todos los enlaces de paginación es decir, las *url* de navegación.

Hasta este punto se ha aplicado un procedimiento estándar para la implementación de componentes que intervienen en una sola tabla, es decir, este procedimiento de implementación del patrón arquitectónico es para construir el *CRUD* de un componente cuya intervención es a una única tabla de la base de datos.

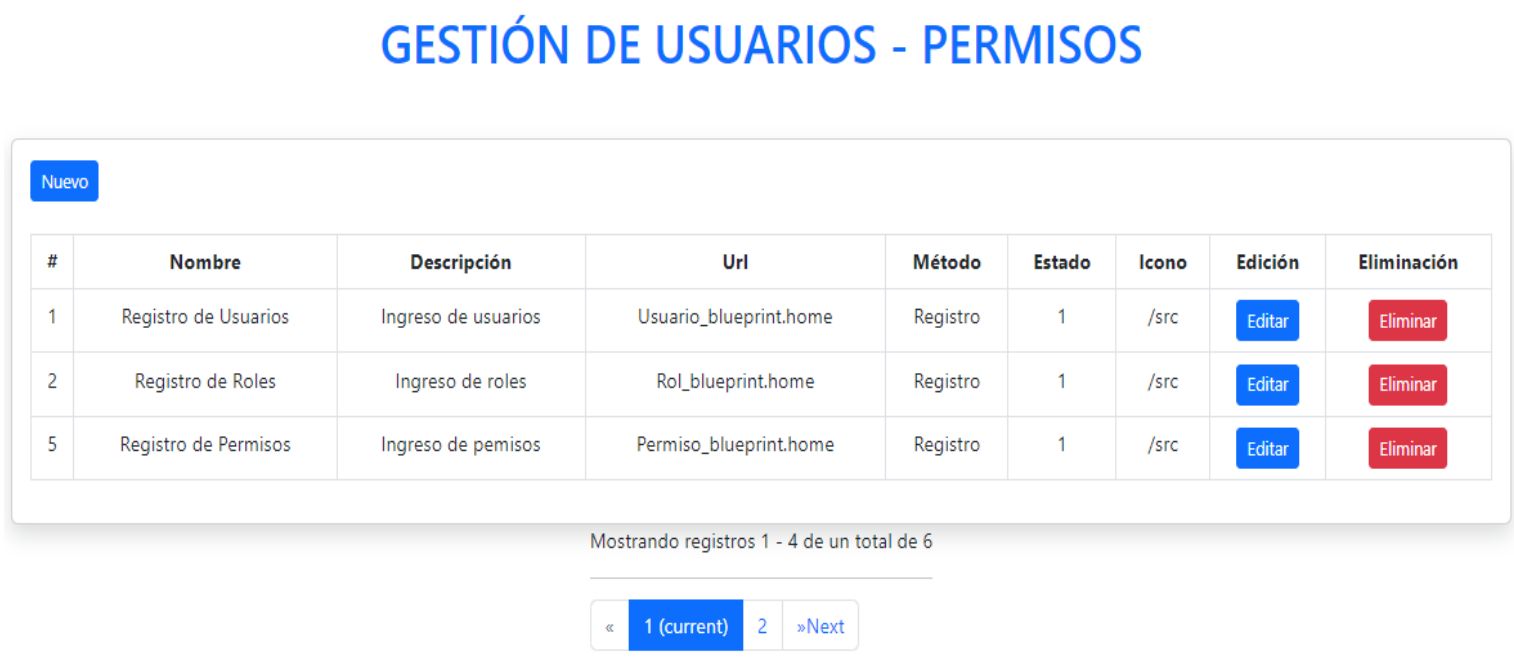
Teniendo en cuenta este procedimiento de implementación del patrón arquitectónico *MVC*, se debe implementar los siguientes componentes que tienen similar estructura, estos componentes son

aquellos que requieren el *CRUD* en una tabla específica. Entre los que se alinean a esta visión están: permisos, rol.

Para el componente de permisos intervendrá en una única tabla llamada *permiso* de la base de datos *seguridad_db*, su url será *http://127.0.0.1:4000/seg/permiso/* y aplicando el procedimiento que se aplicó al componente persona quedará de la siguiente manera.

Figura 32

Gestión de usuarios – permisos.



Nota. Elaborado por los autores.

De la misma manera para el componente rol, cuya intervención se hará en la tabla *rol* de la base de datos *seguridad_db*. su url será *http://127.0.0.1:4000/seg/rol/* y aplicando el procedimiento que se aplicó al componente *persona* quedará de la siguiente manera.

Figura 33

Gestión de usuarios - roles.

Nota. Elaborado por los autores.

En la implementación de un proyecto informático que conlleva el desarrollo de un sistema o aplicación web, siempre va a existir transacciones a más de una tabla en la base de datos. Esta implementación no es la excepción, por ello a continuación se implementará componentes que intervienen con múltiples tablas de la base de datos.

4.6. Implementación del componente Asignar Rol

Esta funcionalidad está enfocada en asignar roles a personas para luego convertirlas en usuarios del sistema. Dicho de otra manera, luego de haber construido la gestión de persona y gestión de rol lo que sigue es la asignación de roles a personas. Luego con la gestión de usuarios del sistema estas personas se convierten en usuarios del sistema con los roles asignados.

4.6.1 Clase AsignaRol

Es una clase especial ya que proviene de una estructura relacional entre la tabla *rol* y *persona* de la base de datos *seguridad_db*. Es la encargada de extraer los datos de la relación entre el rol y la persona. Esta clase estará compuesta por el *idPersona* o identificador de la persona en la tabla persona y el *idRol* o identificador del rol en la tabla rol. El parámetro *intEstado* es un control que permitiera asignar o no la funcionalidad en el sistema.

```
class AsignaRol():
    def __init__(self, idPersona,idRol,intEstado=None) ->
None:
        self.idPersona = idPersona
        self.idRol = idRol
        self.intEstado = intEstado

    def __str__(self) -> str:
        return {
            'idPersona': self.idPersona,
            'idRol': self.idRol,
            'intEstado': self.intEstado
        }
```

4.6.2. AsignarRol Controlador (AsignarRolController.py)

Este script está basado en la misma lógica que tienen los controladores ya implementados en este documento, es decir, tendrá una función para consultar, registrar, modificar y eliminar los datos de dicha asignación.

De acuerdo al estándar establecido lo primero que hay que hacer es importar las librerías y componentes necesarios para implementar este *script* tipo *Python*. Como ya hemos mencionado las librerías a importar siempre van a ser las mismas que las anteriores implementaciones de controladores.

```
from flask import
Blueprint,redirect,jsonify,request,url_for,session
```

También es necesario importar el modelo y su entidad.

```
from models.AsignaRolModel import AsignaRolModel
from models.entities.AsignaRol import AsignaRol
```

Además, se deberá instanciar la variable *main* para tener acceso a los decoradores de las rutas y funciones.

```
main = Blueprint('AsignarRol_blueprint',__name__)
```

La implementación de la función principal junto al decorador raíz es la que permitirá gestionar la lista de asignaciones de roles a personas, de esta manera se podrá extraer la información de la tabla para ser presentada por la vista en su momento.

```
@main.route('/')
def home():
    try:
        if "username" in session:
            return AsignaRolModel.get_page()
        else:
            return redirect(url_for('Login_blueprint.home'))
    except Exception as ex:
        return jsonify({'mensaje':str(ex)}),500
```

De la misma forma tiene que existir una función que permita el registro de los datos en la tabla *personarol*. Esta función deberá recibir los datos que provienen de la vista y ejecutar la función de registro de

datos que está en el modelo. Del mismo modo, manejar las redirecciones en caso de registros éxitos o registros rechazado.

```
@main.route('/add', methods=['POST'])
def add():
    try:
        idPersona = request.form.get('idPersona')
        idRol = request.form.get('idRol')
        intEstado = request.form['intEstado']
        asignaRol = AsignaRol(idPersona,idRol,intEstado)
        ingresos = AsignaRolModel.set_entidad(asignaRol)

        if ingresos >= 1:
            return
redirect(url_for('AsignarRol_blueprint.home'))
        else:
            return jsonify({'menssage':"Error al
insertar"}),500
    except Exception as ex:
        return jsonify({'mensaje':str(ex)}),500
```

En este caso, la actualización de datos no tiene mucho sentido (pese a que si se puede implementar) ya que para cambiar las características de una persona con respecto a un *rol* se lo puede hacer desde cero.

Lo siguiente es la eliminación, en este contexto es preferible eliminar que actualizar, ya que los datos no se podrán repetir.

```
@main.route('/delete/<int:idPersona>/<int:idRol>')
def delete(idPersona,idRol):
    try:
        if idPersona is None and idRol is None:
            return jsonify({'menssage':"Id vacio"}),404
        else:
            fila_afectada =
AsignaRolModel.del_entidad(idPersona,idRol)
            if fila_afectada == 1:
                return
redirect(url_for('AsignarRol_blueprint.home'))
            else:
                return jsonify({'menssage':"Ninguna persona ha
sido eliminada"}),404
    except Exception as ex:
        return jsonify({'mesage':str(ex)}),500
```

4.6.3. AsignarRol Modelo (AsignaRolModel.py)

Como se ha visto en las anteriores implementaciones, el modelo es el puente entre la aplicación y la base de datos, es por ello, que se requiere importar algunas librerías y componentes necesarios para la implementación, de igual manera que los modelos anteriores se requiere la importación de las siguientes librerías:

```
from database.db import get_connection
from .entities.AsignaRol import AsignaRol
from .entities.Persona import Persona
from .entities.Rol import Rol
from .entities.AsignaRolCompleto import AsignaRolCompleto
from flask import request,render_template
from flask_paginate import Pagination
```

En la implementación de la clase se deberá definir las funciones que tienen que ejecutar las diferentes acciones que el modelo debe coordinar con la base de datos.

De esta este modo la función *def get_page(self)*, tiene como responsabilidad extraer los datos de esta tabla para presentarlos en la vista, pero como ya hemos visto en el diseño de la base de datos, esta tabla está formada por identificadores de otras tablas. Esa información no puede ser presentada directamente al usuario de la manera como fue extraída, ya que son números y no caracteres, provocando incertidumbre y poca comprensión en ellos. Por solucionar este percance, se requiere hacer una consulta que permita mediante esas dos *ids* extraer las descripciones correspondientes desde la tabla *rol* y la tabla *persona*.

Teniendo la información formateada con las descripciones de los roles y la persona, esta debe almacenarse en un objeto de tipo *AsignaRolCompleto*, ya que fue diseñado para poder contener ese tipo de datos sin necesidad de una tabla en la base de datos. Es así que el objeto *AsignaRolCompleto* contendrá los siguientes datos: idPersona, idRol, Nombres y Apellidos de la persona, así como el Nombre del rol.

Por otra parte, la aplicación debe permitir al usuario poder escoger de una lista de nombres y apellidos de personas y de una lista de roles para evitar posibles errores al ingreso de datos.

Es por ello, que se requiere diseñar una función destinada a extraer datos de la tabla persona y de la tabla rol, para luego enviarlas a la vista.

```
@classmethod
    def get_page(self):
        try:
            db = get_connection()
            cursor = db.cursor()
            cursor = db.cursor(buffered=True)
            cursor.execute("SELECT COUNT(*) FROM
`personarol`;")
            total_reg = cursor.fetchone()[0]
            num_page = request.args.get('page',1,type=int)

            zise_page = 3
            inicio = (num_page - 1) * zise_page + 1

            cursor.execute("SELECT idPersona, idRol FROM
`personarol`;")
            result = cursor.fetchall()
            lista_asignaRoles = []
            for row in result:
                cursor.execute("""SELECT DISTINCT

`persona`.`id`,`rol`.`id`,`persona`.`strNombres`,
                            `persona`.`strApellidos`,
`rol`.`strNombre`,
                            `personarol`.`intEstado`
                            FROM `persona` INNER JOIN `rol` ON
`persona`.`id`
                            = %s AND `rol`.`id` = %s
                            INNER JOIN `personarol` ON
`idPersona` = %s AND
```

```
                            `idRol` = %s; """ % (row[0],
row[1],row[0], row[1]))

                result = cursor.fetchone()
                asignaRolCompleto =
AsignaRolCompleto(result[0]

,result[1],result[2],result[3]
                                      ,result[4],result[5])
                lista_asignaRoles.append(asignaRolCompleto)

            cursor.execute(f"""SELECT * FROM `personarol`
                               WHERE `idPersona`>= 1 ORDER BY
`idPersona` ASC
                               LIMIT {zise_page} OFFSET
{inicio - 1}""")
            result = cursor.fetchall()
            asignaRoles = []
            for row in result:
                asignaRol = AsignaRol(row[0],row[1],row[2])
                asignaRoles.append(asignaRol.to_JSON())

            fin = min(inicio + zise_page,total_reg)

            if (inicio > total_reg):
                fin = total_reg

            pagination = Pagination(page = num_page,
per_page=zise_page, total =
                          total_reg, display_msg = f"Mostrando
registros {inicio}
                          - {fin} de un total de {total_reg}")

            cursor.execute("SELECT `id`, `strCedula`,
`strNombres`,
                         `strApellidos`, `strCorreo`,
`strTelefono`,
```

```
                        `strDependencia`, `intEstado`,
`strCargo` FROM `persona`
                        WHERE `intEstado` > 0;")
            result = cursor.fetchall()
            list_usuarios = []
            for row in result:
                usuario = Persona(row[0],row[1],row[2],row[3]

,row[4],row[5],row[6],row[7],row[8])
                list_usuarios.append(usuario.to_JSON())

            cursor.execute("SELECT `id`, `strNombre`,
`strDescripcion`,
                           `intEstado`, `strIcono` FROM `rol`
                            WHERE  `intEstado`>0;")
            result = cursor.fetchall()
            list_roles = []
            for row in result:
                rol = Rol(row[0],row[1],row[2],row[3],row[4])
                list_roles.append(rol.to_JSON())
            db.close()

            return
render_template('gestionasignarrol.html',data =
                    lista_asignaRoles,dataU =
                    list_usuarios,dataR =
list_roles,paginacion = pagination)
        except Exception as ex:
            raise Exception(ex)
```

Otra acción que se debe implementar es el registro de la información, con esa premisa se debe realizar una función para registrar la información que por intermedio del modal que dispara el botón “N*uevo”*, desde la vista, reciba un objeto de tipo *AsignaRol*.

```
@classmethod
    def set_entidad(self,asignaRol):
        try:
            db = get_connection()
            cursor = db.cursor()
            #cursor = db.cursor(buffered=True)
            cursor.execute("SELECT idPersona, idRol FROM
`personarol` WHERE
         idPersona= %s and idRol = %s;" %
(asignaRol.idPersona,asignaRol.idRol))
            result = cursor.fetchall()
            if (len(result) == 0):
                with db.cursor() as cursor:
                    sql = "INSERT INTO `personarol`
(`idPersona`,
                          `idRol`,`intEstado`) VALUES
(%s,%s,%s);"
                    dato =
(asignaRol.idPersona,asignaRol.idRol,
                                asignaRol.intEstado)
                    cursor.execute(sql,dato)
                    filas_ingresadas = cursor.rowcount
                    db.commit()
            else:
                filas_ingresadas = cursor.rowcount
            db.close()
            return filas_ingresadas
        except Exception as ex:
            raise Exception(ex)
```

La siguiente acción a implementar y siguiendo el estándar estipulado en las anteriores implementaciones de modelos, se invoca a la función *def del_entidad(self,idPersona,idRol)*, esta función recibirá el id del rol y el id de la persona ya que requiere estar completamente segura de que asignación debe eliminar.

```
@classmethod
    def del_entidad(self,idPersona,idRol):
        try:
            db = get_connection()

            with db.cursor() as cursor:
                sql = "DELETE FROM `personarol` WHERE
`idPersona`=%s AND
                        `idRol`=%s"
                dato = (idPersona,idRol,)
                cursor.execute(sql,dato)
                fila_afectada = cursor.rowcount
                db.commit()
            db.close()
            return fila_afectada
        except Exception as ex:
            raise Exception(ex)
```

Por último, se implementa la función de actualización de datos *def up_Endidad(self,asignarol)* similar al registro; esta función recibe un objeto de tipo *AsignaRol*, y permitirá actualizar la información dependiendo del rol y la persona.

```
@classmethod
    def up_Endidad(self,asignarol):
        try:
            db = get_connection()
            with db.cursor() as cursor:
                sql = "UPDATE `personarol` SET `intEstado`=%s
WHERE `idPersona`
                      = %s AND idRol =
%s"
                dato =
(asignarol.intEstado,asignarol.idPersona,asignarol.idRol)
```

```
                cursor.execute(sql,dato)
                fila_actualizada = cursor.rowcount
                db.commit()
            db.close()
            return fila_actualizada
        except Exception as ex:
            raise Exception(ex)
```

4.6.4. Gestión de AsignarRol (gestionasignarrol.html)

Mediante la vista (View) se visualiza toda la funcionalidad que ha sido implementada en las capas anteriores.

En esta ocasión, esta vista deberá presentar todos los datos informativos de las personas que están registradas en la tabla *personarol*, así como los botones para registro de nuevos usuarios, edición y la eliminación de usuario.

```
<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">
    <title>Gestión de usuarios</title>
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-
          alpha1/dist/css/bootstrap.min.css"
rel="stylesheet"
          integrity="sha384-

GLhlTQ8iRABdZLl6O3oVMWSktQOp6b7In1Zl3/Jr59b6EGGoI1aFkw7cmDA
6j6gD"
```

```
            crossorigin="anonymous">
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-
            alpha1/dist/js/bootstrap.bundle.min.js"
integrity="sha384-

w76AqPfDkMBDXo30jS1Sgez6pr3x5MlQ1ZAGC+nuZB+EYdgRZgiwxhTBTkF
7CXvN"
            crossorigin="anonymous"></script>
</head>
```

En la etiqueta *<body>*, se establecerá la vista que se mostrará al cliente web. La vista está dividida en tres partes: el contenedor del botón nuevo, el contenedor de la tabla donde están la lista de datos y el contenedor de la paginación.

Contenedor del botón Nuevo

Es destinado al botón de "Nuevo", permite gestionar el registro de datos de las asignaciones a la tabla *personarol* para luego enviarlos al controlador. La etiqueta *<button>,* permitirá establecer el botón dentro del * * y permitirá mostrar una ventana emergente mediante un modal. Estas acciones la puede realizar mediante el uso de los parámetros *data-bs-toggle="modal"* y *data-bs-target="#modal"*.

```
<td><button class="btn btn-primary btn-sm" id="btn-primary"
data-bs-
                toggle="modal" data-bs-
target="#modal">Nuevo</button></td>
<br>
```

Este modal es una estructura tipo *form*, que permite dibujar un recuadro que contenga los parámetros que se requieren enviar al controlador para su almacenamiento en la tabla *personarol* de la base de datos *seguridad_db*.

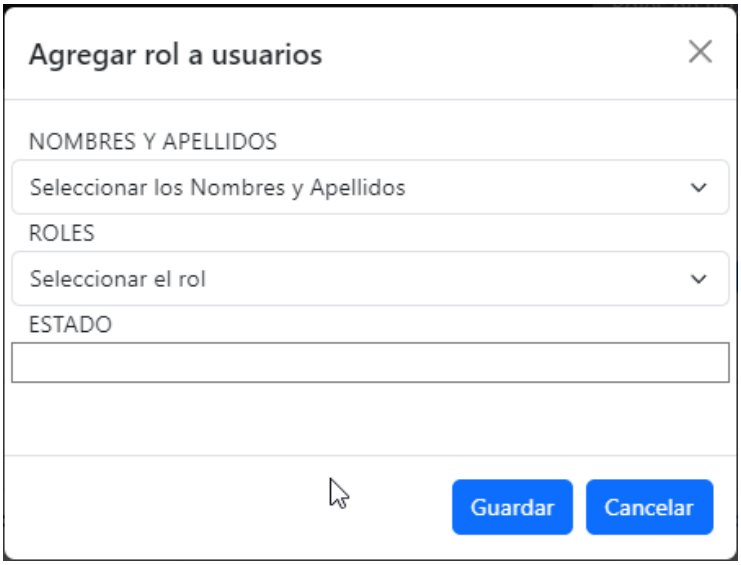
```
<!--Modal de ingreso-->
<div class="modal fade" id="modal" tabindex="-1" aria-
                        labelledby="exampleModalLabel" aria-
hidden="true">
   <div class="modal-dialog">
      <div class="modal-content">
        <div class="modal-header">
          <h1 class="modal-title fs-5"
id="exampleModalLabel">Agregar rol a
                      usuarios</h1>
          <button type="button" class="btn-close" data-bs-
dismiss="modal" aria-
                      label="Close"></button>
        </div>
        <div class="modal-body">
          <form action="/seg/asignarrol/add" method="post">
            <div class="row mb-3">
                <label> NOMBRES Y APELLIDOS</label>
                <select name="idPersona" class="form-select"
aria-label="Default
                      select example">
                  <option  selected>Seleccionar los Nombres y
Apellidos</option>
                  {% for du in dataU %}
                     <option  value="{{du.id}}">{{du.strNombre
s}}

{{du.strApellidos}}</option>
                  {% endfor %}
                  </select> <br>
```

```
                    <label> ROLES</label>
                    <select name="idRol" class="form-select"
aria-label="Default
                                select example">
                    <option  selected>Seleccionar el
rol</option>
                    {% for dr in dataR %}
                      <option  value="{{dr.id}}">{{dr.strNombre}
}</option>
                    {% endfor %}
                    </select> <br>
                    <label>ESTADO</label>
                    <input type="text" class="from-control mb-3"
                                name="intEstado"><br>
                  </div>
                 </div>
                 <div class="modal-footer">
                   <button type="submit" class="btn btn-
primary">Guardar</button>
                   <button type="button" class="btn btn-primary"
data-bs-
                            dismiss="modal">Cancelar</button>
                  </div>
                </form>
              </div>
            </div>
```

El argumento *id*, permite establecer un identificador o nombre único el modal o diálogo, con este identificador se podrá vincular cuando el botón “Nuevo” sea activado o pulsado. Con el modal presente en la pantalla del cliente web, las etiquetas *HTML* dibujarán los diferentes elementos para el ingreso de datos en un formulario que está dispuesto dentro de dicho modal.

Figura 34

Agregar rol a usuarios.



Nota. Elaborado por los autores.

El modal lleva el nombre “Agregar rol a usuario” y está compuesto un *<select>* con la lista de usuarios que están registrados en el sistema, un *<select>* que contiene la lista de los roles creados en el sistema y un elemento *input* para ingresar el estado, cada uno de ellos con un *name* único, esto para que el controlador lo pueda identificar de manera única y extraer los datos que en el existe. Pero todos siempre van a contener un valor que será enviado al constructor por medio del método *POST*.

Al pulsar el botón “Guardar” de tipo “*submit”* contenido en la etiqueta que está dentro del *form*, todas las variables viajarán al

controlador (*AsignaRolController*) vía *POST* y buscarán la ruta "*/seg/asignarrol/add"*, gracias al parámetro *"accion"* de la etiqueta *form*.

Contenedor de la tabla donde están la lista de datos

Esta sección muestra los datos de la tabla *personarol* formateados de una manera legible para el cliente web; además, por cada registro que la interfaz muestre llevará consigo los botones de acción "Editar" y "Eliminar".

Mediante la etiqueta *<table>*, se establecerá una estructura ordenada para colocar la información que el controlador envié a la vista. Luego con la clase *mx-auto* permitirá centrar horizontalmente, además la clase *table-bordered* mostrará los bordes en todos los lados de la tabla y las celdas. Con estos estilos la tabla tendrá una vista más amigable para el cliente.

```
<table class="table mx-auto text-center table-bordered">
```

Lo siguiente es dibujar el encabezado de la tabla que por lo general es estática. Para ello se utilizarán dos etiquetas que forman parte de la estructura de *<table>*, la etiqueta *<thead>* que es la encargada de contener la fila y columnas estáticas que son el encabezado de la tabla.

```
<thead>
   <th scope="col">Nombres</th>
   <th scope="col">Apellidos</th>
   <th scope="col">Rol</th>
   <th scope="col">Estado</th>
   <th scope="col">Eliminación</th>
</thead>
```

Y la etiqueta *<tbody>*, que es la encargada de contener las filas y columnas que son parte del cuerpo de la tabla que por lo general es la parte dinámica. Por su esencia dinámica, esta etiqueta requiere tener una estructura repetitiva que permita ir dibujando fila por fila la información que el controlador le envía para su publicación al cliente; además, necesita usar código jinja2 para poder establecer esta estructura.

Con la estructura *for* se recorrerá la lista de objetos tipo *AsignaRol*, que es enviada desde el modelo.

```
{% for d in data %}

<tr>
   <td>{{d.strNombres}}</td>
   <td>{{d.strApellidos}}</td>
   <td>{{d.strNombre }}</td>
   <td>{{d.intEstado}}</td>
```

A través de la variable *d*, se recorre la lista de objetos *data*, esta lista de objetos contiene información estructurada en un tipo *AsignaRol*, este contiene; Nombres y Apellidos de los usuarios, roles del sistema, estado; que están registrados en la tabla *personarol* de la base de datos *seguridad_db*.

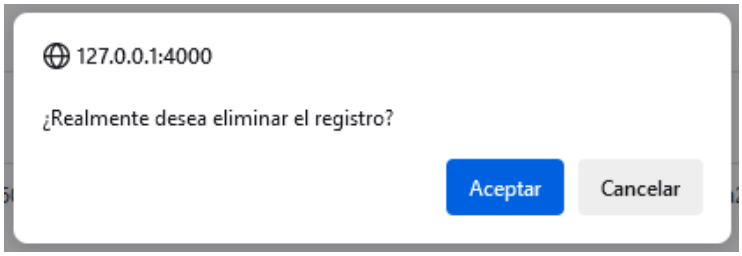
La etiqueta *<tbody>* contiene el artificio *<tr>*, que permite dibujar una fila, este argumento es dinámico ya que va a ir dibujando una fila cada vez que el ciclo *for* dispare un registro. Cada fila de registro presentado, se encontrará el botón de acción "Eliminar".

La columna de acciones para el botón *"Eliminar"*, se requiere establecer un script mediante la función *onclik()* para confirmar si se requiere eliminar o no dicho registro. El camino del *"si"*, contempla la ejecución de la función *del_entidad()* en el modelo, pero antes mediante las funciones de blueprint envía desde la vista al controlador la ruta "*AsignarRol_blueprint.delete*" que no es más que "*/seg/asignarrol/delete/*", y con los argumentos *idPersona = d.idPersona y idRol = d.idRol*, la ruta se completaría de la siguiente manera; "*/seg/asignarrol/delete/{{d.idPersona}}/{{d.idRol}}*".

```
<td><a onclick="return confirm('¿Realmente desea eliminar
el registro?')" href="{{
url_for(url_for('AsignarRol_blueprint.delete',idPersona =
d.idPersona,idRol = d.idRol)
) }}" class="btn btn-danger btn-sm">Eliminar</a></td>
```

Figura 35

Ventana de confirmación.



Nota. Elaborado por los autores.

Contenedor de la paginación

Por último, se requiere visualizar la paginación que de la misma manera fue enviada mediante un objeto llamado "*paginacion*", en este objeto viaja desde el modelo al controlador el número de páginas, el número de registros por páginas, la leyenda y las url de navegación.

```
<div class="row justify-content-md-center">
          <div class="col-md-auto">
            <span>
              {{ paginacion.info }}
              <hr />
            </span>
            {{ paginacion.links }}
          </div>
        </div>
```

En ese contexto, el objeto *paginacion.info* se utilizará para obtener la información de paginación como número de páginas, el número de registros por páginas, la leyenda, mientras que *paginacion.links* obtendrá todos los enlaces de paginación es decir, las *url* de navegación.

Para el componente de asignación de permisos a roles, se podría utilizar la misma estructura que se acaba de presentar ya que su tratamiento es similar. Con esto se ha logrado culminar el módulo que se propuso en el caso de estudio, cuyo objetivo era mostrar al lector una forma de implementación del patrón arquitectónico *MVC* con la herramienta de programación *Python*.

Conclusiones

- Esta implementación no pretende ser un estándar para que los programadores deban seguir obligatoriamente, esta estructura programática no es más que una guía para lograr entender la implementación de aplicaciones web con *Python* y el patrón arquitectónico *MVC*.
- El patrón metodológico *MVC* es una estructura de programación que permite flexibilizar los cambios y ayuda a llevar una línea de trabajo común para el equipo de desarrollo, garantizando de esta manera el éxito del proyecto.

- La herramienta de desarrollo *Python* es versátil, fácil de usar y robusta para la fabricación de proyectos de software de todo tipo, ya en la práctica hemos comprobado su gran flexibilidad para la construcción de proyectos web a medida.

Recomendaciones

- El libro contempla una serie de ejercicios propuestos para facilitar el aprendizaje de las herramientas y su aplicación en el desarrollo web con *Python*, Sin embargo, es recomendable que el lector fabrique y pruebe su propio código, de manera que los conceptos y ejemplos aprendidos, sean dominados y aplicados en proyectos reales.

Referencias

Bala, C. (2021). *Comprensión de Diccionario en Python.* https://www.freecodecamp.org/espanol/news/compresion-de-diccionario-en-python-explicado-con-ejemplos/

Blancarte, O. (2021). *Arquitectura en capas*. https://reactiveprogramming.io/blog/es/

Bustamante, S. (2021). *Entornos virtuales de Python explicados con ejemplos*. https://www.freecodecamp.org/espanol/news/entornos-virtuales-de-python-explicados-con-ejemplos/

Challenger, I., Díaz, Y., Becerra, R. (2014). *El lenguaje de programación Python.* https://www.redalyc.org/pdf/1815/181531232001.pdf

Downey, A., Elkner, J., Meyers, C. (2002). *Aprenda a pensar como un programador con Python.* https://argentinaenpython.com/quiero-aprender-python/aprenda-a-pensar-como-un-programador-con-python.pdf

Estevez, E. (2019). *Estilos y patrones de arquitectura – Parte 1.* http://cs.uns.edu.ar/~ece/ads/downloads/Clases/2019%2008%20AyDS%20-%20Estilos%20y%20Patrones%20de%20Arquitectura%20-%20Parte%201.pdf

Flores, F. (2021). *Qué es Visual Studio Code y qué ventajas ofrece*. https://openwebinars.net/blog/que-es-visual-studio-code-y-que-ventajas-ofrece/

GURU. (2023). *El catálogo de ejemplos en Python*. https://refactoring.guru/es/design-patterns/python

Informática Documentation. (2021). *Guía de seguridad.* https://docs.informatica.com/es_es/data-catalog/shared-content-for-data-catalog/10-5-1/guia-de-seguridad/permisos/resumen-de-permisos.html

IONOS. (2023). *Python tuples: cómo crear listas inalterables*. https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/python-tuples/

Llamas, J. (2023). *Python*. https://economipedia.com/definiciones/python.html

Lozano, J. (2023). *Introducción a Python*. https://j2logo.com/python/tutorial/introduccion-a-python/

Makai, M. (2022). *Full Stack Python-Bootstrap*. https://www.fullstackpython.com/bootstrap-css.html

Mehmood, S. (2022). *Flask CORS*. https://www.delftstack.com/es/howto/python-flask/flask-cors/

Prida, N. (2020). *Qué es un mockup o maqueta web*. https://openwebinars.net/blog/que-es-un-mockup-o-maqueta-web/

Salesforce. (2023). *User Permissions*. https://help.salesforce.com/s/articleView?id=sf.admin_userperms.htm&type=5

Sparkman, M. (2023). *Microsoft Ignite*. https://learn.microsoft.com/es-es/sql/reporting-services/security/role-definitions?view=sql-server-ver16

Valverde, V., Cajamarca, J., Moreano, G. (2023). *Fundamentos de Programación con DFD-PSeInt-Python.* https://cimogsys.espoch.edu.ec/direccion-publicaciones/public/docs/books/2024-01-23-142632-fundamentos_de_programación_con_DFD.pdf

Semblanzas de los autores

Vanessa Lorena Valverde González
https://orcid.org/0000-0002-3501-8353
v_valverde@espoch.edu.ec

Máster Universitario en Ingeniería de Software y Sistemas Informáticos. Magister en Informática Educativa. Ingeniera en Sistemas Informáticos. Analista en Sistemas Informáticos. Docente de la Escuela Superior Politécnica de Chimborazo. Forma parte del Grupo de Investigación Ciencia del Mantenimiento CIMANT. Autora de varios libros y de artículos científicos.

Byron Ernesto Vaca Barahona
https://orcid.org/0000-0002-3622-0668
byron.vaca@espoch.edu.ec

Doctor (Programa en Tecnología Educativa: E-Learning y Gestión del Conocimiento). Máster Universitario en Tecnología Educativa: E-Learning y Gestión de Conocimiento. Máster Universitario en Seguridad Informática y Sistemas Inteligentes. Diplomado en Manejo de Información a través de Internet. Magister en Informática Aplicada. Ingeniero en Sistemas. Rector de la Escuela Superior Politécnica de Chimborazo. Autor de varios libros y de artículos científicos.

Gustavo Xavier Hidalgo Solórzano
https://orcid.org/0000-0003-1811-2070
ghidalgo@espoch.edu.ec

Magister en Formulación, Evaluación y Gerencia de Proyectos para el desarrollo, Ingeniero en Sistemas Informáticos. Analista de automatización de procesos de la Escuela Superior Politécnica de Chimborazo. Autor de artículos científicos.

Desarrollo Web con Python y Flask es un libro diseñado especialmente para estudiantes que han aprobado las asignaturas de Bases de Datos y Programación Orientada a Objetos, así como para profesionales del área informática que deseen profundizar en esta tecnología.

La obra inicia con una introducción clara y concisa a los fundamentos de Python, Flask y Bootstrap, ofreciendo las bases necesarias para construir aplicaciones web modernas. A medida que avanza, guía al lector en la implementación del módulo de seguridad y manejo de usuarios, roles y permisos. Es una herramienta práctica y accesible que combina teoría y ejemplos reales, ideal para quienes buscan adquirir habilidades sólidas en el desarrollo web profesional.

Vanessa Lorena Valverde González. Máster Universitario en Ingeniería de Software y Sistemas Informáticos. Magister en Informática Educativa. Ingeniera en Sistemas Informáticos. Analista en Sistemas Informáticos. Docente de la Escuela Superior Politécnica de Chimborazo. Forma parte del Grupo de Investigación Ciencia del Mantenimiento CIMANT. Autora de varios libros y de artículos científicos.

Byron Ernesto Vaca Barahona. Doctor (Programa En Tecnología Educativa: E-Learning y Gestión Del Conocimiento). Máster Universitario en Tecnología Educativa: E-Learning Y Gestión De Conocimiento. Máster Universitario en Seguridad Informática y Sistemas Inteligentes. Diplomado en Manejo de Información a través de Internet. Magister en Informática Aplicada. Ingeniero en Sistemas. Rector de la Escuela Superior Politécnica de Chimborazo. Autor de varios libros y de artículos científicos.

Gustavo Xavier Hidalgo Solórzano. Magister en Formulación, Evaluación y Gerencia de Proyectos para el desarrollo, Ingeniero en sistemas Informáticos. Analista de automatización de procesos de la Escuela Superior Politécnica de Chimborazo. Autor de artículos científicos.

ISBN 978-9942-679-53-6
9789942679536