Sesión 1: Introducción a Python, Servidores Web y Flask

Introducción a Python

Python es un lenguaje de programación de alto nivel, interpretado y de propósito general, creado por Guido van Rossum y lanzado por primera vez en 1991. Python es conocido por su sintaxis simple y legible, lo que facilita a los desarrolladores escribir código claro y lógico para proyectos a pequeña y gran escala.

Algunas características clave de Python incluyen:

- Sintaxis simple y legible: Python está diseñado para ser fácil de leer y escribir, lo que permite a los programadores expresar conceptos en menos líneas de código.
- Multiparadigma: Soporta programación orientada a objetos, programación estructurada y funcional.
- Gran comunidad y ecosistema: Hay una gran cantidad de bibliotecas y frameworks disponibles para casi cualquier tarea.

Instalación de Python

Puedes descargar e instalar Python desde su sitio web oficial: python.org. Asegúrate de añadir Python a tu PATH durante la instalación para poder ejecutarlo desde la línea de comandos.

Variables y estructuras de datos

Las variables en Python no necesitan declaración explícita de tipo. El tipo de una variable es inferido automáticamente por el valor asignado a ella.

Ejemplo de declaración de variables:

```
1. # Declaración de variables
2. x = 5
3. y = "Hola, Mundo!"
4. z = 3.14
5. print(x)  # Output: 5
6. print(y)  # Output: Hola, Mundo!
7. print(z)  # Output: 3.14
```

Estructuras de datos:

Listas

Las listas son colecciones ordenadas y mutables de elementos.

```python
# Lista en Python
mi_lista = [1, 2, 3, 4, 5]
print(mi_lista)  # Output: [1, 2, 3, 4, 5]

# Acceso a elementos
print(mi_lista[0])  # Output: 1

# Añadir elementos
mi_lista.append(6)
print(mi_lista)  # Output: [1, 2, 3, 4, 5, 6]

# Eliminar elementos
mi_lista.remove(3)
print(mi_lista)  # Output: [1, 2, 4, 5, 6]

```

Tuplas

Las tuplas son colecciones ordenadas e inmutables de elementos.

```python
# Tupla en Python
mi_tupla = (1, 2, 3, 4, 5)
print(mi_tupla)  # Output: (1, 2, 3, 4, 5)

# Acceso a elementos
print(mi_tupla[1])  # Output: 2

# Las tuplas no se pueden modificar (inmutables)
# mi_tupla[1] = 10  # Esto producirá un error
```

Conjuntos

Los conjuntos son colecciones desordenadas de elementos únicos.

```python
# Conjunto en Python
mi_conjunto = {1, 2, 3, 4, 5}
print(mi_conjunto)  # Output: {1, 2, 3, 4, 5}

# Añadir elementos
mi_conjunto.add(6)
print(mi_conjunto)  # Output: {1, 2, 3, 4, 5, 6}

# Eliminar elementos
mi_conjunto.remove(3)
print(mi_conjunto)  # Output: {1, 2, 4, 5, 6}

```

Diccionarios

Los diccionarios son colecciones desordenadas de pares clave-valor.

```
# Diccionario en Python
mi_diccionario = {"nombre": "Juan", "edad": 30}
print(mi_diccionario)  # Output: {'nombre': 'Juan', 'edad': 30}

# Acceso a valores
print(mi_diccionario["nombre"])  # Output: Juan

# Añadir elementos
mi_diccionario["ciudad"] = "Madrid"
print(mi_diccionario)  # Output: {'nombre': 'Juan', 'edad': 30, 'ciudad': 'Madrid'}

# Eliminar elementos
del mi_diccionario["edad"]
print(mi_diccionario)  # Output: {'nombre': 'Juan', 'ciudad': 'Madrid'}

```

Operadores lógicos

Los operadores lógicos se utilizan para realizar operaciones lógicas sobre valores y variables. Los operadores lógicos más comunes en Python son and, or y not.

Ejemplo de operadores lógicos:

```
# Operadores lógicos
a = True
b = False

print(a and b)  # Output: False
print(a or b)   # Output: True
print(not a)    # Output: False

```

Ciclos y condicionales en Python

Condicionales:

Las estructuras condicionales permiten ejecutar bloques de código basados en condiciones específicas.

```
# Condicionales
x = 10
if x > 5:
    print("x es mayor que 5")
elif x == 5:
    print("x es igual a 5")
else:
    print("x es menor que 5")

```

Ciclos for:

Los ciclos for se utilizan para iterar sobre una secuencia (como una lista, tupla, diccionario, conjunto o una cadena de caracteres).

```
# Ciclo for
for i in range(5):
    print(i)

```

Ciclos while:

Los ciclos while se utilizan para ejecutar un bloque de código repetidamente mientras una condición sea verdadera.

```
# Ciclo while
i = 0
while i < 5:
    print(i)
    i += 1

```

Lectura de Archivos CSV en Python usando CSV Reader

Python proporciona una biblioteca estándar llamada csv que facilita la lectura y escritura de archivos CSV (Comma-Separated Values). Utilizando el módulo csv, podemos leer archivos CSV de manera sencilla y eficiente. Este módulo ofrece un objeto reader que convierte cada línea del archivo CSV en una lista de valores, permitiéndonos procesar datos de forma estructurada y accesible.

A continuación, se presenta un ejemplo de cómo leer un archivo CSV usando el módulo csv en Python:

```
import csv

# Supongamos que tenemos un archivo 'data.csv' con el siguiente contenido:
# nombre,edad,ciudad
# Ana,28,Madrid
# Luis,34,Barcelona
# Maria,22,Valencia

# Abre el archivo CSV en modo lectura
with open('data.csv', mode='r', newline='') as file:
    # Crea un objeto reader
    csv_reader = csv.reader(file)

    # Lee la cabecera del archivo CSV
    header = next(csv_reader)
    print(f"Encabezados: {header}")

    # Itera sobre cada fila en el archivo CSV
    for row in csv_reader:
        print(f"Nombre: {row[0]}, Edad: {row[1]}, Ciudad: {row[2]}")

```

En este ejemplo, primero abrimos el archivo CSV en modo lectura utilizando la función open. Luego, creamos un objeto csv.reader para leer el contenido del archivo. Usamos next(csv_reader) para leer la primera fila, que generalmente contiene los encabezados de las columnas. Posteriormente, iteramos sobre cada fila restante en el archivo CSV, extrayendo y mostrando los datos correspondientes. Esta técnica es útil para procesar datos tabulares en una variedad de aplicaciones.

Funciones en Python

Las funciones en Python permiten organizar el código en bloques reutilizables y modulares. Las funciones se definen usando la palabra clave def.

Ejemplo de funciones:

```
# Definición de una función
def saludar(nombre):
    return f"Hola, {nombre}!"

# Llamada a la función
print(saludar("Juan"))  # Output: Hola, Juan!

```

Las funciones pueden tener parámetros y devolver valores. También pueden tener valores predeterminados para los parámetros.

```
# Función con parámetros y valores predeterminados
def saludar(nombre, mensaje="Hola"):
    return f"{mensaje}, {nombre}!"

# Llamada a la función con y sin el parámetro mensaje
print(saludar("Juan"))             # Output: Hola, Juan!
print(saludar("Juan", "Buenos días"))  # Output: Buenos días, Juan!

```

Concepto de Servidores Web

Un servidor web es una aplicación de software que se encarga de atender las solicitudes de los clientes (navegadores web) y devolverles los recursos solicitados (páginas web, imágenes, archivos, etc.). Los servidores web pueden ser software específico como Apache o Nginx, o pueden ser aplicaciones personalizadas construidas con frameworks y bibliotecas de lenguajes de programación.

Los servidores web funcionan utilizando el protocolo HTTP (HyperText Transfer Protocol), que define cómo se estructuran y transmiten los mensajes entre el cliente y el servidor.

Introducción a Flask

Flask es un microframework para el desarrollo de aplicaciones web en Python. Es conocido por ser ligero y fácil de usar, lo que lo hace ideal para pequeños proyectos y aplicaciones que requieren flexibilidad.

Flask se basa en Werkzeug, un kit de herramientas de WSGI (Web Server Gateway Interface), y Jinja2, un motor de plantillas para Python. Flask no incluye funcionalidades adicionales como la autenticación o la gestión de base de datos, pero puedes agregar estas funcionalidades utilizando extensiones.

Instalación de Flask

Para instalar Flask, puedes usar pip, el gestor de paquetes de Python:

```
pip install Flask
```

Estructura de archivos y carpetas en Flask

La estructura básica de una aplicación Flask es bastante simple. Aquí hay un ejemplo de cómo podría organizarse:

```
/mi_app
    /static
    /templates
    app.py
```

- /static: Esta carpeta contiene archivos estáticos como CSS, JavaScript e imágenes.
- /templates: Esta carpeta contiene plantillas HTML.
- app.py: Este es el archivo principal de la aplicación Flask.

Ejemplo de estructura:

1. Crear el directorio del proyecto:

```
mkdir mi_app
cd mi_app
```

2. Crear subcarpetas y archivos necesarios:

```
mkdir static templates
touch app.py
```

Debug en Flask

El modo de depuración en Flask permite a los desarrolladores rastrear errores fácilmente. Cuando el modo de depuración está activado, Flask reiniciará automáticamente el servidor cuando detecte cambios en el código y proporcionará un traceback interactivo en el navegador cuando ocurra un error.

Activar el modo de depuración:

Para activar el modo de depuración, configura debug=True al ejecutar la aplicación.

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

```

Rutas en Flask

Las rutas en Flask determinan qué contenido se sirve para diferentes URL. Las rutas se definen usando decoradores.

Ejemplo de rutas en Flask:

```
# app.py
from flask import Flask

app = Flask(__name__)

@app.route('/')
def inicio():
    return "¡Hola, Mundo!"

@app.route('/saludo/<nombre>')
def saludar(nombre):
    return f"¡Hola, {nombre}!"

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

```

En este ejemplo, hay dos rutas definidas:

- La ruta '/' que devuelve "¡Hola, Mundo!".
- La ruta '/saludo/<nombre>' que devuelve un saludo personalizado con el nombre proporcionado en la URL.

Parte Práctica

Paso 1: Instalación de Flask

Para comenzar, necesitas instalar Flask. Puedes hacerlo usando pip:

```
pip install Flask
```

Paso 2: Crear la estructura de la aplicación

Crea una nueva carpeta para tu proyecto y dentro de ella, crea las subcarpetas y archivos necesarios:

```
mkdir mi_app
cd mi_app
mkdir static templates
touch app.py
```

Paso 3: Escribir el código de la aplicación

Abre app.py en tu editor de texto preferido y escribe el siguiente código:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def inicio():
    return "¡Hola, Mundo!"

@app.route('/saludo/<nombre>')
def saludar(nombre):
    return f"¡Hola, {nombre}!"

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

Paso 4: Ejecutar la aplicación

Para ejecutar tu aplicación Flask, abre una terminal y navega al directorio de tu proyecto. Luego ejecuta:

```
python app.py
```

Abre tu navegador web y ve a *http://127.0.0.1:5000/*. Deberías ver "¡Hola, Mundo!". Para probar la ruta de saludo, ve a *http://127.0.0.1:5000/saludo/TuNombre*.

Sesión 2: Templates y Formularios con Jinja2

Backend con Python y frontend con HTML, CSS y JS

En el desarrollo web, el backend se refiere al servidor y la lógica que maneja la comunicación entre la base de datos y el navegador del usuario. El frontend se refiere a la interfaz de usuario, incluyendo HTML para la estructura, CSS para el estilo y JavaScript para la interactividad.

Backend con Python: Python, junto con frameworks como Flask, se utiliza para crear la lógica del servidor, manejar rutas, procesar datos y conectarse a bases de datos.

Frontend con HTML, CSS y JS:

- HTML (HyperText Markup Language): Es el lenguaje de marcado utilizado para crear la estructura básica de una página web.
- CSS (Cascading Style Sheets): Es el lenguaje de estilo utilizado para diseñar y layout de las páginas web.
- JavaScript (JS): Es el lenguaje de programación utilizado para crear interactividad en las páginas web.

Introducción a templates con Flask y Jinja2

Flask utiliza Jinja2 como su motor de plantillas. Jinja2 permite a los desarrolladores de Flask crear HTML dinámico, que puede incorporar variables y lógica de control de flujo (como bucles y condicionales).

Ejemplo básico de plantilla Jinja2:

```
<!-- templates/inicio.html -->
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Inicio</title>
</head>
<body>
    <h1>{{ mensaje }}</h1>
</body>
</html>

```

Integración de Jinja2 con Flask:

```python
# app.py
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def inicio():
    mensaje = "¡Bienvenido a la aplicación Flask!"
    return render_template('inicio.html', mensaje=mensaje)

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

```

Static, HTML, CSS y JS

En Flask, los archivos estáticos (CSS, JavaScript e imágenes) se colocan en la carpeta static. Las plantillas HTML se colocan en la carpeta templates.

Estructura de directorios:

```
/mi_app
    /static
        /css
            estilos.css
        /js
            scripts.js
    /templates
        inicio.html
    app.py

```

Ejemplo de archivo CSS:

```css
/* static/css/estilos.css */
body {
    font-family: Arial, sans-serif;
    background-color: #f4f4f4;
    color: #333;
}
h1 {
    color: #0066cc;
}

```

Ejemplo de archivo JavaScript:

```
// static/js/scripts.js
document.addEventListener('DOMContentLoaded', function() {
    console.log('JavaScript está funcionando');
});

```

Integración de CSS y JS en HTML:

```
<!-- templates/inicio.html -->
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Inicio</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='css/estilos.css') }}">
</head>
<body>
    <h1>{{ mensaje }}</h1>
    <script src="{{ url_for('static', filename='js/scripts.js') }}"></script>
</body>
</html>

```

Introducción a Bootstrap CSS

Bootstrap es un framework de CSS popular que facilita el diseño de páginas web responsivas y modernas. Para usar Bootstrap, debes incluir sus archivos CSS y JS en tu proyecto.

Incluir Bootstrap en una plantilla HTML:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Bootstrap 5 Demo</title>
    <!-- Bootstrap 5 CSS -->
    <link href="https://stackpath.bootstrapcdn.com/bootstrap/5.0.0-
beta3/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
    <!-- Navbar -->
    <nav class="navbar navbar-expand-lg navbar-light bg-light">
        <div class="container-fluid">
            <a class="navbar-brand" href="#">Navbar</a>
            <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-
bs-target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle
navigation">
                <span class="navbar-toggler-icon"></span>
            </button>
            <div class="collapse navbar-collapse" id="navbarNav">
                <ul class="navbar-nav">
                    <li class="nav-item">
                        <a class="nav-link active" aria-current="page" href="#">Home</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="#">Features</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="#">Pricing</a>
                    </li>
                </ul>
            </div>
        </div>
    </nav>

    <div class="container mt-5">
        <!-- Grid System -->
        <h2>Grid System</h2>
        <div class="row">
            <div class="col-md-4">Column 1</div>
            <div class="col-md-4">Column 2</div>
            <div class="col-md-4">Column 3</div>
        </div>

        <!-- Buttons -->
        <h2 class="mt-5">Buttons</h2>
        <button type="button" class="btn btn-primary">Primary</button>
        <button type="button" class="btn btn-secondary">Secondary</button>
        <button type="button" class="btn btn-success">Success</button>
        <button type="button" class="btn btn-danger">Danger</button>
        <button type="button" class="btn btn-warning">Warning</button>
        <button type="button" class="btn btn-info">Info</button>
        <button type="button" class="btn btn-light">Light</button>
        <button type="button" class="btn btn-dark">Dark</button>

        <!-- Form -->
        <h2 class="mt-5">Form</h2>
        <form>
```

```
            <div class="mb-3">
                <label for="exampleInputEmail1" class="form-label">Email address</label>
                <input type="email" class="form-control" id="exampleInputEmail1" aria-
describedby="emailHelp">
                <div id="emailHelp" class="form-text">We'll never share your email with
anyone else.</div>
            </div>
            <div class="mb-3">
                <label for="exampleInputPassword1" class="form-label">Password</label>
                <input type="password" class="form-control" id="exampleInputPassword1">
            </div>
            <div class="mb-3 form-check">
                <input type="checkbox" class="form-check-input" id="exampleCheck1">
                <label class="form-check-label" for="exampleCheck1">Check me out</label>
            </div>
            <button type="submit" class="btn btn-primary">Submit</button>
        </form>

        <!-- Table -->
        <h2 class="mt-5">Table</h2>
        <table class="table">
            <thead>
                <tr>
                    <th scope="col">#</th>
                    <th scope="col">First</th>
                    <th scope="col">Last</th>
                    <th scope="col">Handle</th>
                </tr>
            </thead>
            <tbody>
                <tr>
                    <th scope="row">1</th>
                    <td>Mark</td>
                    <td>Otto</td>
                    <td>@mdo</td>
                </tr>
                <tr>
                    <th scope="row">2</th>
                    <td>Jacob</td>
                    <td>Thornton</td>
                    <td>@fat</td>
                </tr>
                <tr>
                    <th scope="row">3</th>
                    <td>Larry</td>
                    <td>the Bird</td>
                    <td>@twitter</td>
                </tr>
            </tbody>
        </table>

        <!-- Card -->
        <h2 class="mt-5">Card</h2>
        <div class="card" style="width: 18rem;">
            <img src="https://via.placeholder.com/150" class="card-img-top" alt="...">
            <div class="card-body">
                <h5 class="card-title">Card title</h5>
                <p class="card-text">Some quick example text to build on the card title
and make up the bulk of the card's content.</p>
                <a href="#" class="btn btn-primary">Go somewhere</a>
            </div>
        </div>
    </div>

    <!-- Bootstrap 5 JS and dependencies -->
```

```
    <script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.9.3/dist/umd/popper.min.js"></script>
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/5.0.0-beta3/js/bootstrap.min.js"></script>
</body>
</html>
```

Decoradores

Los decoradores en Python son una herramienta poderosa que permite modificar el comportamiento de una función o método. Un decorador es esencialmente una función que toma otra función y le añade alguna funcionalidad adicional antes de devolverla. Los decoradores se utilizan comúnmente para la autenticación, el registro, la administración de transacciones, entre otros. El uso de decoradores mejora la reutilización de código y la separación de preocupaciones, ya que permite mantener el código limpio y modular. Aquí hay un ejemplo básico que muestra cómo se puede utilizar un decorador para registrar la ejecución de una función:

```python
def mi_decorador(func):
    def envoltura(*args, **kwargs):
        print(f"Llamando a la función {func.__name__}")
        resultado = func(*args, **kwargs)
        print(f"Función {func.__name__} ejecutada")
        return resultado
    return envoltura

@mi_decorador
def di_hola():
    print("¡Hola, mundo!")

di_hola()
```

Blueprints

Los Blueprints en Flask permiten organizar tu aplicación en componentes modulares. Son útiles para aplicaciones grandes que tienen múltiples funcionalidades.

Ejemplo de uso de Blueprints:

1. Crear una estructura de directorios

```
/mi_app
    /static
    /templates
    /modulos
        /admin
            __init__.py
            rutas.py
    app.py

```

2. Definir un Blueprint en rutas.py:

```python
# modulos/admin/rutas.py
from flask import Blueprint

admin_bp = Blueprint('admin', __name__)

@admin_bp.route('/admin')
def admin():
    return "Bienvenido al panel de administración"

```

3. Registrar el Blueprint en app.py:

```python
# app.py
from flask import Flask
from modulos.admin.rutas import admin_bp

app = Flask(__name__)
app.register_blueprint(admin_bp)

@app.route('/')
def inicio():
    return "¡Bienvenido a la aplicación Flask!"

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

```

Formularios

Estructura del Proyecto: Crea una estructura de proyecto sencilla:

```
1. /flask_form_example
2. ├── app.py
3. └── templates
4.     └── form.html
5.
```

Código en app.py:

```python
from flask import Flask, request, render_template

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('form.html')

@app.route('/submit', methods=['POST'])
def submit():
    name = request.form.get('name')
    email = request.form.get('email')
    return f'Received: Name - {name}, Email - {email}'

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

```

Código en templates/form.html:

```html
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Flask Form Example</title>
    <!-- Bootstrap 5 CSS -->
    <link href="https://stackpath.bootstrapcdn.com/bootstrap/5.0.0-beta3/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
    <div class="container mt-5">
        <h1 class="mb-4">Submit your information</h1>
        <form action="/submit" method="post">
            <div class="mb-3">
                <label for="name" class="form-label">Name:</label>
                <input type="text" class="form-control" id="name" name="name" required>
            </div>
            <div class="mb-3">
                <label for="email" class="form-label">Email:</label>
                <input type="email" class="form-control" id="email" name="email" required>
            </div>
            <button type="submit" class="btn btn-primary">Submit</button>
        </form>
    </div>
    <!-- Bootstrap 5 JS and dependencies -->
    <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.9.3/dist/umd/popper.min.js"></script>
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/5.0.0-beta3/js/bootstrap.min.js"></script>
</body>
</html>
```

Manejar datos de formulario con Python

Código en app.py

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

app = Flask(__name__)
app.secret_key = 'your_secret_key'  # Necesario para usar flash messages

def validate_email(email):
    return '@' in email and '.' in email

@app.route('/', methods=['GET', 'POST'])
def contact():
    if request.method == 'POST':
        email = request.form.get('email')
        comment = request.form.get('comment')

        # Validación del formulario
        if not email or not comment:
            flash('Both email and comment are required!', 'danger')
        elif not validate_email(email):
            flash('Invalid email address!', 'danger')
        else:
            flash('Your comment has been submitted successfully!', 'success')
            return redirect(url_for('contact'))

    return render_template('contact.html')

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

Código en templates/contact.html

```html
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Contact Form</title>
    <!-- Bootstrap 5 CSS -->
    <link href="https://stackpath.bootstrapcdn.com/bootstrap/5.0.0-
beta3/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
    <!-- Navbar -->
    <nav class="navbar navbar-expand-lg navbar-light bg-light">
        <div class="container-fluid">
            <a class="navbar-brand" href="#">MySite</a>
            <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-
bs-target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle
navigation">
                <span class="navbar-toggler-icon"></span>
```

```
            </button>
            <div class="collapse navbar-collapse" id="navbarNav">
                <ul class="navbar-nav">
                    <li class="nav-item">
                        <a class="nav-link active" aria-current="page" href="#">Home</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="#">About</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="#">Services</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="#">Contact</a>
                    </li>
                </ul>
            </div>
        </div>
    </nav>

    <div class="container mt-5">
        <h1>Contact Us</h1>
        <!-- Flash messages -->
        {% with messages = get_flashed_messages(with_categories=true) %}
            {% if messages %}
                {% for category, message in messages %}
                    <div class="alert alert-{{ category }} mt-4">
                        {{ message }}
                    </div>
                {% endfor %}
            {% endif %}
        {% endwith %}
        <form action="/" method="post">
            <div class="mb-3">
                <label for="email" class="form-label">Email address</label>
                <input type="email" class="form-control" id="email" name="email" required>
            </div>
            <div class="mb-3">
                <label for="comment" class="form-label">Comment</label>
                <textarea class="form-control" id="comment" name="comment" rows="3"
required></textarea>
            </div>
            <button type="submit" class="btn btn-primary">Submit</button>
        </form>
    </div>

    <!-- Bootstrap 5 JS and dependencies -->
    <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.9.3/dist/umd/popper.min.js"></script>
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/5.0.0-
beta3/js/bootstrap.min.js"></script>
</body>
</html>

```

Para evitar la inyección de código (tanto SQL Injection como Cross-Site Scripting - XSS), es fundamental seguir prácticas seguras y utilizar librerías especializadas cuando sea necesario.

Evitar la Inyección de Código en Formularios

Validación y Sanitización de Entrada

1. Validación del Lado del Servidor: Valida y sanitiza todas las entradas del usuario en el servidor. Nunca confíes solo en la validación del lado del cliente.
2. Escapado de Salida: Asegúrate de escapar cualquier dato de usuario antes de mostrarlo en una página web.

Uso de Librerías Adicionales

1. WTForms para la Validación de Formularios: WTForms es una librería que facilita la validación de formularios en Flask.

```python
from flask import Flask, render_template, request, flash, redirect, url_for
from flask_wtf import FlaskForm
from wtforms import StringField, TextAreaField, SubmitField
from wtforms.validators import DataRequired, Email

app = Flask(__name__)
app.secret_key = 'your_secret_key'

class ContactForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Email()])
    comment = TextAreaField('Comment', validators=[DataRequired()])
    submit = SubmitField('Submit')

@app.route('/', methods=['GET', 'POST'])
def contact():
    form = ContactForm()
    if form.validate_on_submit():
        email = form.email.data
        comment = form.comment.data
        flash('Your comment has been submitted successfully!', 'success')
        return redirect(url_for('contact'))
    return render_template('contact.html', form=form)

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

```

contact.html:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Contact Form</title>
    <link href="https://stackpath.bootstrapcdn.com/bootstrap/5.0.0-beta3/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
    <nav class="navbar navbar-expand-lg navbar-light bg-light">
        <div class="container-fluid">
            <a class="navbar-brand" href="#">MySite</a>

            <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">
                <span class="navbar-toggler-icon"></span>
            </button>
            <div class="collapse navbar-collapse" id="navbarNav">
                <ul class="navbar-nav">
                    <li class="nav-item">
                        <a class="nav-link active" aria-current="page" href="#">Home</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="#">About</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="#">Services</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="#">Contact</a>
                    </li>
                </ul>
            </div>
        </div>
    </nav>
    <div class="container mt-5">
        <h1>Contact Us</h1>
        {% with messages = get_flashed_messages(with_categories=true) %}
            {% if messages %}
                {% for category, message in messages %}
                    <div class="alert alert-{{ category }} mt-4">
                        {{ message }}
                    </div>
                {% endfor %}
            {% endif %}
        {% endwith %}
        <form method="post" novalidate>
            {{ form.hidden_tag() }}
            <div class="mb-3">
                {{ form.email.label(class="form-label") }}
                {{ form.email(class="form-control") }}
            </div>
            <div class="mb-3">
                {{ form.comment.label(class="form-label") }}
                {{ form.comment(class="form-control") }}
```

```
            </div>
            <div class="mb-3">
                {{ form.submit(class="btn btn-primary") }}
            </div>
        </form>
    </div>
    <script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.9.3/dist/umd/popper.min.js"></script>
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/5.0.0-
beta3/js/bootstrap.min.js"></script>
</body>
</html>

```

Evitar la Inyección de SQL

Uso de ORM

1. SQLAlchemy: Utiliza un Object-Relational Mapping (ORM) como SQLAlchemy para interactuar con la base de datos de forma segura.

```python
from flask import Flask, request, render_template, redirect, url_for, flash
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///site.db'
app.secret_key = 'your_secret_key'
db = SQLAlchemy(app)

class Comment(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(120), unique=False, nullable=False)
    comment = db.Column(db.Text, nullable=False)

@app.route('/', methods=['GET', 'POST'])
def contact():
    if request.method == 'POST':
        email = request.form.get('email')
        comment = request.form.get('comment')
        if not email or not comment:
            flash('Both email and comment are required!', 'danger')
        elif not '@' in email or not '.' in email:
            flash('Invalid email address!', 'danger')
        else:
            new_comment = Comment(email=email, comment=comment)
            db.session.add(new_comment)
            db.session.commit()
            flash('Your comment has been submitted successfully!', 'success')
            return redirect(url_for('contact'))
    return render_template('contact.html')

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

```

Sesión 3: Programación Orientada a Objetos en Python y Desarrollo de APIs con Flask

Programación Orientada a Objetos: Conceptos

La Programación Orientada a Objetos (POO) es un paradigma de programación que utiliza "objetos" y sus interacciones para diseñar aplicaciones y programas informáticos. Algunos conceptos clave de la POO incluyen:

- Clases y Objetos: Una clase es una plantilla para crear objetos (instancias). Un objeto es una instancia de una clase.
- Encapsulamiento: Agrupa datos y métodos que operan sobre esos datos en una sola unidad, llamada clase.
- Herencia: Permite a una clase derivar propiedades y comportamientos de otra clase.
- Polimorfismo: Permite tratar objetos de diferentes clases de la misma manera, siempre que compartan una interfaz común.

Programación Orientada a Objetos con Python

Python es un lenguaje orientado a objetos, lo que significa que puedes definir tus propias clases y crear instancias de estas clases. Aquí te mostramos cómo.

Definir una clase

```python
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def saludar(self):
        return f"Hola, mi nombre es {self.nombre} y tengo {self.edad} años."

```

Crear objetos

```
# Crear instancias de la clase Persona
persona1 = Persona("Juan", 30)
persona2 = Persona("María", 25)

# Llamar a métodos del objeto
print(persona1.saludar())  # Output: Hola, mi nombre es Juan y tengo 30 años.
print(persona2.saludar())  # Output: Hola, mi nombre es María y tengo 25 años.

```

Herencia

```
# Definir una clase que hereda de Persona
class Estudiante(Persona):
    def __init__(self, nombre, edad, matricula):
        super().__init__(nombre, edad)
        self.matricula = matricula

    def saludar(self):
        return f"Hola, soy {self.nombre}, un estudiante de {self.edad} años con matrícula {self.matricula}."

# Crear una instancia de Estudiante
estudiante1 = Estudiante("Carlos", 22, "A1234")
print(estudiante1.saludar())  # Output: Hola, soy Carlos, un estudiante de 22 años con matrícula A1234.

```

Polimorfismo

```
def presentar(persona):
    print(persona.saludar())

# Usar la función presentar con diferentes objetos
presentar(persona1)  # Output: Hola, mi nombre es Juan y tengo 30 años.
presentar(estudiante1)  # Output: Hola, soy Carlos, un estudiante de 22 años con matrícula A1234.

```

Ejercicios de PPO con Python

1. Crear una clase Animal con subclases Perro y Gato

```python
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre

    def sonido(self):
        raise NotImplementedError("Este método debe ser implementado por las subclases")

class Perro(Animal):
    def sonido(self):
        return f"{self.nombre} dice: Guau!"

class Gato(Animal):
    def sonido(self):
        return f"{self.nombre} dice: Miau!"

perro = Perro("Rex")
gato = Gato("Michi")

print(perro.sonido())  # Output: Rex dice: Guau!



print(gato.sonido())  # Output: Michi dice: Miau!

```

2. Implementar una clase Vehiculo y subclases Coche y Bicicleta

```python
class Vehiculo:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

    def descripcion(self):
        return f"Marca: {self.marca}, Modelo: {self.modelo}"

class Coche(Vehiculo):
    def __init__(self, marca, modelo, puertas):
        super().__init__(marca, modelo)
        self.puertas = puertas

    def descripcion(self):
        return f"{super().descripcion()}, Puertas: {self.puertas}"

class Bicicleta(Vehiculo):
    def __init__(self, marca, modelo, tipo):
        super().__init__(marca, modelo)
        self.tipo = tipo

    def descripcion(self):
        return f"{super().descripcion()}, Tipo: {self.tipo}"

coche = Coche("Toyota", "Corolla", 4)
bicicleta = Bicicleta("Giant", "Talon", "Montaña")

print(coche.descripcion())  # Output: Marca: Toyota, Modelo: Corolla, Puertas: 4
print(bicicleta.descripcion())  # Output: Marca: Giant, Modelo: Talon, Tipo: Montaña

```

Concepto de Web API's o REST API

Una API (Interfaz de Programación de Aplicaciones) es un conjunto de reglas que permite a los programas comunicarse entre sí. Las APIs web, como las REST APIs (Representational State Transfer), permiten la comunicación entre un servidor y un cliente utilizando el protocolo HTTP.

Características de REST APIs

- Stateless: Cada solicitud del cliente al servidor debe contener toda la información necesaria para entender y procesar la solicitud.
- Métodos HTTP: Utiliza métodos HTTP estándar como GET, POST, PUT, DELETE.
- URIs: Utiliza URIs para identificar los recursos.
- Formato de Datos: Normalmente usa JSON para el intercambio de datos.

Ejemplo de API con Flask

Configurar Flask para crear una API

```python
from flask import Flask, jsonify, request

app = Flask(__name__)

# Simulamos una base de datos en memoria
usuarios = [
    {"id": 1, "nombre": "Juan"},
    {"id": 2, "nombre": "María"}
]

@app.route('/usuarios', methods=['GET'])
def obtener_usuarios():
    return jsonify(usuarios)

@app.route('/usuarios/<int:id>', methods=['GET'])
def obtener_usuario(id):
    usuario = next((u for u in usuarios if u["id"] == id), None)
    return jsonify(usuario) if usuario else (jsonify({"error": "Usuario no encontrado"}),
404)

@app.route('/usuarios', methods=['POST'])
def crear_usuario():
    nuevo_usuario = request.json
    usuarios.append(nuevo_usuario)
    return jsonify(nuevo_usuario), 201

@app.route('/usuarios/<int:id>', methods=['PUT'])
def actualizar_usuario(id):
    usuario = next((u for u in usuarios if u["id"] == id), None)
    if usuario:
        datos = request.json
        usuario.update(datos)
        return jsonify(usuario)
    return jsonify({"error": "Usuario no encontrado"}), 404

@app.route('/usuarios/<int:id>', methods=['DELETE'])
def eliminar_usuario(id):
    usuario = next((u for u in usuarios if u["id"] == id), None)
    if usuario:
        usuarios.remove(usuario)
        return jsonify({"mensaje": "Usuario eliminado"})
    return jsonify({"error": "Usuario no encontrado"}), 404

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

```

Interacción del Frontend y Backend por medio de APIs

El frontend interactúa con el backend a través de las APIs para enviar y recibir datos. Aquí hay un ejemplo simple usando JavaScript para interactuar con la API de Flask.

Ejemplo de interacción con API usando JavaScript

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Interacción con API</title>
</head>
<body>
    <h1>Usuarios</h1>
    <button onclick="obtenerUsuarios()">Obtener Usuarios</button>
    <ul id="lista-usuarios"></ul>

    <script>
        function obtenerUsuarios() {
            fetch('/usuarios')
                .then(response => response.json())
                .then(data => {
                    const lista = document.getElementById('lista-usuarios');
                    lista.innerHTML = '';
                    data.forEach(usuario => {
                        const item = document.createElement('li');
                        item.textContent = `${usuario.id}: ${usuario.nombre}`;
                        lista.appendChild(item);
                    });
                });
        }
    </script>
</body>
</html>

```

APIs Profesionales con Flask y Mejores Prácticas

Para crear APIs profesionales con Flask, sigue estas mejores prácticas:

1. Validación de Datos: Asegúrate de validar todos los datos de entrada para proteger tu aplicación de datos incorrectos o maliciosos.
2. Gestión de Errores: Proporciona mensajes de error claros y útiles. Utiliza códigos de estado HTTP apropiados.
3. Autenticación y Autorización: Protege tus endpoints utilizando mecanismos de autenticación y autorización.
4. Documentación: Documenta tu API para que otros desarrolladores puedan entender y utilizar tus endpoints correctamente.
5. Versionamiento: Utiliza versionamiento en tus APIs para manejar cambios y actualizaciones sin romper la compatibilidad con clientes existentes.

Ejemplo de API Profesional

```python
from flask import Flask, jsonify, request, abort

app = Flask(__name__)

usuarios = [
    {"id": 1, "nombre": "Juan"},
    {"id": 2, "nombre": "María"}
]

@app.route('/api/v1/usuarios', methods=['GET'])
def obtener_usuarios():
    return jsonify({"usuarios": usuarios})

@app.route('/api/v1/usuarios/<int:id>', methods=['GET'])
def obtener_usuario(id):
    usuario = next((u for u in usuarios if u["id"] == id), None)
    if usuario is None:
        abort(404, description="Usuario no encontrado")
    return jsonify(usuario)

@app.route('/api/v1/usuarios', methods=['POST'])
def crear_usuario():
    if not request.json or not 'nombre' in request.json:
        abort(400, description="Datos inválidos")
    nuevo_usuario = {
        "id": usuarios[-1]["id"] + 1 if usuarios else 1,
        "nombre": request.json['nombre']
    }
    usuarios.append(nuevo_usuario)
    return jsonify(nuevo_usuario), 201

@app.route('/api/v1/usuarios/<int:id>', methods=['PUT'])
def actualizar_usuario(id):
    usuario = next((u for u in usuarios if u["id"] == id), None)
    if usuario is None:
        abort(404, description="Usuario no encontrado")
    if not request.json:
        abort(400, description="Datos inválidos")
    usuario['nombre'] = request.json.get('nombre', usuario['nombre'])
    return jsonify(usuario)

@app.route('/api/v1/usuarios/<int:id>', methods=['DELETE'])
def eliminar_usuario(id):
    usuario = next((u for u in usuarios if u["id"] == id), None)
    if usuario is None:
        abort(404, description="Usuario no encontrado")
    usuarios.remove(usuario)
    return jsonify({"mensaje": "Usuario eliminado"})

@app.errorhandler(404)
def recurso_no_encontrado(e):
    return jsonify(error=str(e)), 404

@app.errorhandler(400)
```

```
55. def solicitud_invalida(e):
56.     return jsonify(error=str(e)), 400
57. 
58. if __name__ == '__main__':
59.     app.run(debug=True)
60. 
```

Sesión 4: Autenticación y Gestión de Usuarios

Contraseñas Hard Coded en el Código

Almacenar contraseñas directamente en el código es una mala práctica de seguridad. Las contraseñas hard coded son vulnerables porque cualquiera que tenga acceso al código puede verlas y explotarlas. En su lugar, las contraseñas deben encriptarse y almacenarse de manera segura.

Ejemplo de una mala práctica:

```
# Evitar hard code de contraseñas en el código
contraseña_admin = "supersecreta123"

```

Implementación de Autenticación por Contraseña con Flask

Implementaremos la autenticación de usuarios manualmente utilizando Flask. En este proceso, incluiremos la encriptación de contraseñas con Bcrypt y el manejo de sesiones con Flask.

Encriptación de Contraseñas con Bcrypt

Bcrypt es una biblioteca de encriptación de contraseñas que facilita el almacenamiento seguro de contraseñas.

Instalación de Bcrypt:

```
pip install bcrypt
```

Uso de Bcrypt:

```
import bcrypt

# Encriptar una contraseña
password = "mi_contraseña"
hashed = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

# Verificar una contraseña
bcrypt.checkpw(password.encode('utf-8'), hashed)

```

Almacenamiento de Usuarios y Contraseñas en Base de Datos SQLite

SQLite es una base de datos liviana y fácil de usar, ideal para pequeños proyectos.

Configuración básica con SQLite:

```python
from flask import Flask, request, render_template, redirect, url_for, session
import sqlite3
import bcrypt

app = Flask(__name__)
app.config['SECRET_KEY'] = 'mi_secreto'

# Conexión a la base de datos SQLite
def get_db_connection():
    conn = sqlite3.connect('database.db')
    conn.row_factory = sqlite3.Row
    return conn

# Crear la tabla de usuarios
def create_users_table():
    conn = get_db_connection()
    conn.execute('''
        CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            username TEXT NOT NULL UNIQUE,
            password TEXT NOT NULL,
            admin BOOLEAN NOT NULL DEFAULT 0
        )
    ''')
    conn.commit()
    conn.close()

create_users_table()

```

Registro de Usuarios

Vista para registrar usuarios:

```python
@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

        conn = get_db_connection()
        try:
            conn.execute('INSERT INTO users (username, password) VALUES (?, ?)',
(username, hashed_password))
            conn.commit()
```

```
        except sqlite3.IntegrityError:
            return "El usuario ya existe."
        finally:
            conn.close()
        return redirect(url_for('login'))
    return render_template('register.html')
```

Plantilla de registro (register.html):

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Registro</title>
</head>
<body>
    <h2>Registro</h2>
    <form method="post">
        <label for="username">Usuario:</label>
        <input type="text" id="username" name="username" required>
        <br>
        <label for="password">Contraseña:</label>
        <input type="password" id="password" name="password" required>
        <br>
        <button type="submit">Registrarse</button>
    </form>
</body>
</html>

```

Inicio de Sesión

Vista para iniciar sesión:

```python
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password'].encode('utf-8')

        conn = get_db_connection()
        user = conn.execute('SELECT * FROM users WHERE username = ?',
(username,)).fetchone()
        conn.close()

        if user and bcrypt.checkpw(password, user['password']):
            session['user_id'] = user['id']
            session['username'] = user['username']
            session['admin'] = user['admin']
            return redirect(url_for('dashboard'))
        else:
            return 'Credenciales incorrectas'
    return render_template('login.html')

```

Plantilla de inicio de sesión (login.html):

```html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Login</title>
</head>
<body>
    <h2>Login</h2>
    <form method="post">
        <label for="username">Usuario:</label>
        <input type="text" id="username" name="username" required>
        <br>
        <label for="password">Contraseña:</label>
        <input type="password" id="password" name="password" required>
        <br>
        <button type="submit">Login</button>
    </form>
</body>
</html>

```

Protección de Rutas

Para proteger las rutas, verificaremos si el usuario ha iniciado sesión.

Ejemplo de ruta protegida:

```
@app.route('/dashboard')
def dashboard():
    if 'user_id' not in session:
        return redirect(url_for('login'))
    return f'Bienvenido {session["username"]}!'

@app.route('/logout')
def logout():
    session.clear()
    return redirect(url_for('login'))

```

Niveles de Usuarios: Usuario Admin

Los niveles de usuarios permiten definir diferentes permisos y accesos en la aplicación.

Ejemplo de usuario admin:

```
@app.route('/admin')
def admin():
    if 'user_id' not in session or not session.get('admin'):
        return 'Acceso denegado', 403
    return 'Panel de administrador'

```

Ejemplo de Usuario Admin y Pantalla para Creación de Usuarios

Implementaremos una pantalla donde los administradores pueden crear nuevos usuarios.

Vista para crear usuarios:

```
@app.route('/admin/create_user', methods=['GET', 'POST'])
def create_user():
    if 'user_id' not in session or not session.get('admin'):
        return 'Acceso denegado', 403
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

        conn = get_db_connection()
        try:
```

```
            conn.execute('INSERT INTO users (username, password) VALUES (?, ?)',
(username, hashed_password))
            conn.commit()
        except sqlite3.IntegrityError:
            return "El usuario ya existe."
        finally:
            conn.close()
        return redirect(url_for('admin'))
    return render_template('create_user.html')

```

Plantilla para crear usuarios (create_user.html):

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Crear Usuario</title>
</head>
<body>
    <h2>Crear Usuario</h2>
    <form method="post">
        <label for="username">Usuario:</label>
        <input type="text" id="username" name="username" required>
        <br>
        <label for="password">Contraseña:</label>
        <input type="password" id="password" name="password" required>
        <br>
        <button type="submit">Crear Usuario</button>
    </form>
</body>
</html>

```

Parte Práctica Completa

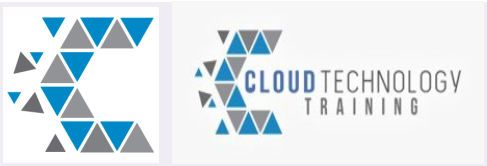
app.py con todas las funcionalidades:

```
from flask import Flask, render_template, redirect, url_for, request, session
import sqlite3
import bcrypt

app = Flask(__name__)
app.config['SECRET_KEY'] = 'mi_secreto'

# Conexión a la base de datos SQLite
def get_db_connection():
    conn = sqlite3.connect('database.db')
    conn.row_factory = sqlite3.Row
    return conn

# Crear la tabla de usuarios
def create_users_table():
    conn = get_db_connection()
```



```
    conn.execute('''
        CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            username TEXT NOT NULL UNIQUE,
            password TEXT NOT NULL,
            admin BOOLEAN NOT NULL DEFAULT 0
        )
    ''')
    conn.commit()
    conn.close()

create_users_table()

@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

        conn = get_db_connection()
        try:
            conn.execute('INSERT INTO users (username, password) VALUES (?, ?)',
(username, hashed_password))
            conn.commit()
        except sqlite3.IntegrityError:
            return "El usuario ya existe."
        finally:
            conn.close()
        return redirect(url_for('login'))
    return render_template('register.html')

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password'].encode('utf-8')

        conn = get_db_connection()
        user = conn.execute('SELECT * FROM users WHERE username = ?',
(username,)).fetchone()
        conn.close()

        if user and bcrypt.checkpw(password, user['password']):
            session['user_id'] = user['id']
            session['username'] = user['username']
            session['admin'] = user['admin']
            return redirect(url_for('dashboard'))
        else:
            return 'Credenciales incorrectas'
    return render_template('login.html')

@app.route('/dashboard')
def dashboard():
    if 'user_id' not in session:
        return redirect(url_for('login'))
    return f'Bienvenido {session["username"]}!'

```

```python
@app.route('/logout')
def logout():
    session.clear()
    return redirect(url_for('login'))

@app.route('/admin')
def admin():
    if 'user_id' not in session or not session.get('admin'):
        return 'Acceso denegado', 403
    return 'Panel de administrador'

@app.route('/admin/create_user', methods=['GET', 'POST'])
def create_user():
    if 'user_id' not in session or not session.get('admin'):
        return 'Acceso denegado', 403
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

        conn = get_db_connection()
        try:
            conn.execute('INSERT INTO users (username, password) VALUES (?, ?)',
(username, hashed_password))
            conn.commit()
        except sqlite3.IntegrityError:
            return "El usuario ya existe."
        finally:
            conn.close()
        return redirect(url_for('admin'))
    return render_template('create_user.html')

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

```

Sesión 5: SQLAlchemy con Flask y SQLite

Introducción a SQLAlchemy

SQLAlchemy es una biblioteca SQL de Python que proporciona un conjunto completo de patrones de persistencia diseñados para aplicaciones de alta calidad. Es un ORM (Object Relational Mapper) que permite mapear clases Python a tablas de bases de datos y viceversa, facilitando las operaciones de CRUD (Crear, Leer, Actualizar, Eliminar) en bases de datos de manera más intuitiva y orientada a objetos.

Programación Orientada a Objetos (PPO) con SQLAlchemy

SQLAlchemy permite integrar la Programación Orientada a Objetos (PPO) con bases de datos relacionales. Puedes definir clases Python que se mapean a tablas de bases de datos, y las instancias de estas clases representan filas en las tablas.

Objetos Clave en SQLAlchemy

En este tutorial, cubriremos tres objetos clave en SQLAlchemy: engine, declarative_base y session.

1. Engine: Gestiona la conexión a la base de datos y la comunicación con la misma.
2. Declarative Base: Permite definir modelos de datos mediante clases Python.
3. Session: Maneja las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en la base de datos.

Creación de una Aplicación Simple con SQLAlchemy

Crea una estructura de directorios para tu proyecto:

```
/mi_app
    /static
    /templates
        base.html
        index.html
    app.py

```

Uso del Engine

El objeto engine es responsable de gestionar la conexión a la base de datos. Se configura utilizando una URL de base de datos que especifica el tipo de base de datos, el nombre de usuario, la contraseña, la dirección del servidor y el nombre de la base de datos.

```
1. from sqlalchemy import create_engine
2.
3. # Crear el engine
4. engine = create_engine('sqlite:///mi_base_de_datos.db')
```

Uso de Declarative Base

El objeto declarative_base se utiliza para definir modelos de datos mediante clases Python. Proporciona una base a partir de la cual se pueden crear todas las clases de modelo.

Ejemplo:

```
 1. from sqlalchemy.ext.declarative import declarative_base
 2. from sqlalchemy import Column, Integer, String
 3.
 4. # Crear el objeto base
 5. Base = declarative_base()
 6.
 7. # Definir un modelo de datos
 8. class Usuario(Base):
 9.     __tablename__ = 'usuarios'
10.     id = Column(Integer, primary_key=True)
11.     nombre = Column(String, nullable=False)
12.     email = Column(String, unique=True, nullable=False)
13.
14.     def __repr__(self):
15.         return f'<Usuario(nombre={self.nombre}, email={self.email})>'
```

Uso de Session

El objeto session se utiliza para manejar las operaciones CRUD en la base de datos. Se crea utilizando una clase sessionmaker que está vinculada al engine.

Ejemplo:

```python
from sqlalchemy.orm import sessionmaker

# Crear una sesión
Session = sessionmaker(bind=engine)
session = Session()

# Crear un nuevo usuario
nuevo_usuario = Usuario(nombre='Juan', email='juan@example.com')

# Agregar el usuario a la sesión
session.add(nuevo_usuario)

# Confirmar los cambios en la base de datos
session.commit()
```

Ejemplo Completo de Aplicación

Vamos a crear una aplicación Flask completa que incluya un tutorial de SQLAlchemy. En este ejemplo, crearemos un CRUD de usuarios.

app.py Completo

```python
from flask import Flask, render_template, request, redirect, url_for
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

app = Flask(__name__)

# Configuración de la base de datos
DATABASE_URL = 'sqlite:///mi_base_de_datos.db'
engine = create_engine(DATABASE_URL)
Base = declarative_base()
Session = sessionmaker(bind=engine)
session = Session()

# Definición del modelo de usuario
class Usuario(Base):
```

```
    __tablename__ = 'usuarios'
    id = Column(Integer, primary_key=True)
    nombre = Column(String, nullable=False)
    email = Column(String, unique=True, nullable=False)

    def __repr__(self):
        return f'<Usuario(nombre={self.nombre}, email={self.email})>'

# Crear las tablas
Base.metadata.create_all(engine).
@app.route('/')
def index():
    usuarios = session.query(Usuario).all()
    return render_template('index.html', usuarios=usuarios)

@app.route('/crear_usuario', methods=['GET', 'POST'])
def crear_usuario():
    if request.method == 'POST':
        nombre = request.form['nombre']
        email = request.form['email']
        nuevo_usuario = Usuario(nombre=nombre, email=email)
        session.add(nuevo_usuario)
        session.commit()
        return redirect(url_for('index'))
    return render_template('crear_usuario.html')

@app.route('/actualizar_usuario/<int:id>', methods=['GET', 'POST'])
def actualizar_usuario(id):
    usuario = session.query(Usuario).filter_by(id=id).first()
    if request.method == 'POST':
        usuario.nombre = request.form['nombre']
        usuario.email = request.form['email']
        session.commit()
        return redirect(url_for('index'))
    return render_template('actualizar_usuario.html', usuario=usuario)


@app.route('/eliminar_usuario/<int:id>', methods=['POST'])
def eliminar_usuario(id):
    usuario = session.query(Usuario).filter_by(id=id).first()
    session.delete(usuario)
    session.commit()
    return redirect(url_for('index'))

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

```

Plantilla base (base.html)

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>{% block title %}{% endblock %}</title>
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css">
</head>
<body>
    <nav class="navbar navbar-expand-lg navbar-light bg-light">
        <a class="navbar-brand" href="{{ url_for('index') }}">MiApp</a>
        <div class="collapse navbar-collapse">
            <ul class="navbar-nav ml-auto">
```

```
                <li class="nav-item"><a class="nav-link" href="{{ url_for('index')
}}">Inicio</a></li>
                <li class="nav-item"><a class="nav-link" href="{{ url_for('crear_usuario')
}}">Crear Usuario</a></li>
            </ul>
        </div>
    </nav>
    <div class="container">
        {% block content %}{% endblock %}
    </div>
</body>
</html>
```

Plantilla de inicio (index.html)

```
{% extends "base.html" %}
{% block title %}Inicio{% endblock %}
{% block content %}
<h1>Lista de Usuarios</h1>
<ul class="list-group">
    {% for usuario in usuarios %}
        <li class="list-group-item">
            <strong>{{ usuario.nombre }}</strong> - {{ usuario.email }}
            <a href="{{ url_for('actualizar_usuario', id=usuario.id) }}" class="btn btn-
primary btn-sm">Editar</a>
            <form action="{{ url_for('eliminar_usuario', id=usuario.id) }}" method="post"
style="display:inline;">
                <button type="submit" class="btn btn-danger btn-sm">Eliminar</button>
            </form>
        </li>
    {% endfor %}
</ul>
{% endblock %}

```

Plantilla de creación de usuario (crear_usuario.html)

```
{% extends "base.html" %}
{% block title %}Crear Usuario{% endblock %}
{% block content %}
<h2>Crear Usuario</h2>
<form method="post">
    <div class="form-group">
        <label for="nombre">Nombre:</label>
        <input type="text" id="nombre" name="nombre" class="form-control" required>
    </div>
    <div class="form-group">
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" class="form-control" required>
    </div>
    <button type="submit" class="btn btn-primary">Crear Usuario</button>
</form>
{% endblock %}
```

Plantilla de actualización de usuario (actualizar_usuario.html)

```
{% extends "base.html" %}
{% block title %}Actualizar Usuario{% endblock %}
{% block content %}
<h2>Actualizar Usuario</h2>
<form method="post">
    <div class="form-group">
        <label for="nombre">Nombre:</label>
        <input type="text" id="nombre" name="nombre" class="form-control" value="{{ usuario.nombre }}" required>
    </div>
    <div class="form-group">
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" class="form-control" value="{{ usuario.email }}" required>
    </div>
    <button type="submit" class="btn btn-primary">Actualizar Usuario</button>
</form>
{% endblock %}
```

Ejemplo de CRUD de Usuarios con SQLAlchemy en SQLite

En este ejemplo, crearemos una aplicación Flask que maneje un CRUD de usuarios y permita a los usuarios guardar notas.

Paso 1: Configurar el proyecto

Crea una estructura de directorios para tu proyecto:

```
/mi_app
    /static

    /templates
        base.html
        index.html
        login.html
        register.html
        dashboard.html
    app.py

```

Paso 2: Configurar la aplicación Flask

```python
from flask import Flask, request, render_template, redirect, url_for, session, jsonify
import sqlite3
import bcrypt
from sqlalchemy import create_engine, Column, Integer, String, Boolean, ForeignKey
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker, relationship

app = Flask(__name__)
app.config['SECRET_KEY'] = 'mi_secreto'

# Configuración de la base de datos
DATABASE_URL = 'sqlite:///mi_base_de_datos.db'
engine = create_engine(DATABASE_URL)
Base = declarative_base()
Session = sessionmaker(bind=engine)
db_session = Session()

# Definición del modelo de usuario
class Usuario(Base):
    __tablename__ = 'usuarios'
    id = Column(Integer, primary_key=True)
    username = Column(String, unique=True, nullable=False)
    password = Column(String, nullable=False)
    admin = Column(Boolean, default=False)
    notas = relationship('Nota', backref='usuario', lazy=True)

class Nota(Base):
    __tablename__ = 'notas'
    id = Column(Integer, primary_key=True)
```

```
    titulo = Column(String, nullable=False)
    contenido = Column(String, nullable=False)
    usuario_id = Column(Integer, ForeignKey('usuarios.id'), nullable=False)

# Crear las tablas
Base.metadata.create_all(engine)

#### Paso 3: Crear la funcionalidad de registro de usuarios

@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

        nuevo_usuario = Usuario(username=username, password=hashed_password)
        db_session.add(nuevo_usuario)
        db_session.commit()
        return redirect(url_for('login'))


    return render_template('register.html')

#### Paso 4: Crear la funcionalidad de inicio de sesión

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password'].encode('utf-8')

        usuario = db_session.query(Usuario).filter_by(username=username).first()
        if usuario and bcrypt.checkpw(password, usuario.password):
            session['user_id'] = usuario.id
            session['username'] = usuario.username
            session['admin'] = usuario.admin
            return redirect(url_for('dashboard'))
        else:
            return 'Credenciales incorrectas'
    return render_template('login.html')

#### Paso 5: Proteger rutas y crear el dashboard

@app.route('/dashboard', methods=['GET', 'POST'])
def dashboard():
    if 'user_id' not in session:
        return redirect(url_for('login'))
    usuario = db_session.query(Usuario).filter_by(id=session['user_id']).first()
    if request.method == 'POST':
        titulo = request.form['titulo']
        contenido = request.form['contenido']
        nueva_nota = Nota(titulo=titulo, contenido=contenido, usuario_id=usuario.id)
        db_session.add(nueva_nota)
        db_session.commit()
    notas = db_session.query(Nota).filter_by(usuario_id=usuario.id).all()
    return render_template('dashboard.html', usuario=usuario, notas=notas)

@app.route('/logout')
def logout():
    session.clear()
    return redirect(url_for('login'))

```

Crear las plantillas HTML

Plantilla base (base.html)

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>{% block title %}{% endblock %}</title>
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css">
</head>
<body>
    <nav class="navbar navbar-expand-lg navbar-light bg-light">
        <a class="navbar-brand" href="{{ url_for('index') }}">MiApp</a>
        <div class="collapse navbar-collapse">
            <ul class="navbar-nav ml-auto">


                {% if session['username'] %}
                    <li class="nav-item"><a class="nav-link" href="{{ url_for('dashboard')
}}">Dashboard</a></li>
                    <li class="nav-item"><a class="nav-link" href="{{ url_for('logout')
}}">Logout</a></li>
                {% else %}
                    <li class="nav-item"><a class="nav-link" href="{{ url_for('login')
}}">Login</a></li>
                    <li class="nav-item"><a class="nav-link" href="{{ url_for('register')
}}">Register</a></li>
                {% endif %}
            </ul>
        </div>
    </nav>
    <div class="container">
        {% block content %}{% endblock %}
    </div>
</body>
</html>

```

Plantilla de inicio (index.html)

```
{% extends "base.html" %}
{% block title %}Inicio{% endblock %}
{% block content %}
<h1>Bienvenido a MiApp</h1>
<p>Esta es una aplicación de ejemplo que utiliza Flask, SQLAlchemy y SQLite.</p>
{% endblock %}
```

Plantilla de registro (register.html)

```
{% extends "base.html" %}
{% block title %}Registro{% endblock %}
{% block content %}
<h2>Registro</h2>
<form method="post">
    <div class="form-group">
        <label for="username">Usuario:</label>
        <input type="text" id="username" name="username" class="form-control" required>
    </div>
    <div class="form-group">
        <label for="password">Contraseña:</label>
        <input type="password" id="password" name="password" class="form-control"
required>
    </div>
    <button type="submit" class="btn btn-primary">Registrarse</button>
</form>
{% endblock %}

```

Plantilla de inicio de sesión (login.html)

```
{% extends "base.html" %}
{% block title %}Login{% endblock %}
{% block content %}
<h2>Login</h2>

<form method="post">
    <div class="form-group">
        <label for="username">Usuario:</label>
        <input type="text" id="username" name="username" class="form-control" required>
    </div>
    <div class="form-group">
        <label for="password">Contraseña:</label>
        <input type="password" id="password" name="password" class="form-control"
required>
    </div>
    <button type="submit" class="btn btn-primary">Login</button>
</form>
{% endblock %}

```

Plantilla de dashboard (dashboard.html)

```
{% extends "base.html" %}
{% block title %}Dashboard{% endblock %}
{% block content %}
<h2>Dashboard</h2>
<h3>Bienvenido, {{ usuario.username }}</h3>
<form method="post">
    <div class="form-group">
        <label for="titulo">Título:</label>
        <input type="text" id="titulo" name="titulo" class="form-control" required>
    </div>
    <div class="form-group">
        <label for="contenido">Contenido:</label>
        <textarea id="contenido" name="contenido" class="form-control"
required></textarea>
    </div>
    <button type="submit" class="btn btn-primary">Guardar Nota</button>
</form>
<h3>Tus Notas</h3>
<ul class="list-group">
    {% for nota in notas %}
        <li class="list-group-item">
            <h5>{{ nota.titulo }}</h5>
            <p>{{ nota.contenido }}</p>
        </li>
    {% endfor %}
</ul>
{% endblock %}

```

Sesión 6: Hosting con Flask

Mejores Prácticas en el Hosting de Apps con Flask

Cuando despliegas una aplicación Flask en producción, debes seguir algunas mejores prácticas para asegurarte de que tu aplicación sea segura, eficiente y fácil de mantener. Aquí hay algunas recomendaciones clave:

1. Configurar Variables de Entorno: Utiliza variables de entorno para almacenar configuraciones sensibles, como claves secretas y credenciales de base de datos.
2. Usar un Servidor de Producción: No uses el servidor de desarrollo de Flask para producción. Utiliza servidores de producción como Gunicorn o uWSGI.
3. Habilitar HTTPS: Asegura que tu aplicación esté accesible a través de HTTPS para proteger la transmisión de datos entre el cliente y el servidor.
4. Gestión de Dependencias: Utiliza un archivo requirements.txt o Pipfile para gestionar las dependencias de tu proyecto y asegurar que todas las dependencias necesarias estén instaladas.
5. Registro de Logs: Configura un sistema de registro (logging) para rastrear errores y eventos importantes en tu aplicación.
6. Monitorización y Escalabilidad: Configura herramientas de monitorización y considera la escalabilidad de tu aplicación para manejar aumentos en el tráfico.

Opciones para el Hosting de Apps en Flask

Existen varias opciones para alojar aplicaciones Flask, cada una con sus propias ventajas y desventajas. Algunas opciones populares incluyen:

1. Heroku: Una plataforma como servicio (PaaS) que simplifica el despliegue y la gestión de aplicaciones web. Es ideal para desarrolladores que desean un proceso de despliegue sencillo.
2. AWS (Amazon Web Services): Una plataforma en la nube que ofrece servicios como EC2 para el hosting de servidores. Es altamente escalable y flexible, pero puede ser más complejo de configurar.
3. Google Cloud Platform (GCP): Ofrece servicios similares a AWS, con opciones como App Engine y Compute Engine para el despliegue de aplicaciones.
4. DigitalOcean: Una plataforma de infraestructura en la nube que ofrece servidores virtuales (droplets) fáciles de configurar y administrar.
5. PythonAnywhere: Un servicio de hosting específico para aplicaciones Python, que incluye herramientas y entornos preconfigurados para un despliegue rápido.

Tutorial para Hosting en Heroku

Heroku es una opción popular para desplegar aplicaciones Flask debido a su facilidad de uso y configuración rápida. A continuación, se muestra un tutorial paso a paso para desplegar una aplicación Flask en Heroku.

Paso 1: Preparar tu aplicación Flask

Asegúrate de que tu aplicación Flask esté lista para ser desplegada. Crea un archivo requirements.txt y un archivo Procfile en el directorio raíz de tu proyecto.

Archivo requirements.txt:

```
Flask==2.0.1
gunicorn==20.1.0

```

Archivo Procfile:

```
web: gunicorn app:app
```

Paso 2: Instalar la CLI de Heroku

Si aún no tienes la CLI de Heroku instalada, descárgala e instálala desde devcenter.heroku.com/articles/heroku-cli.

Paso 3: Crear un Nuevo Proyecto en Heroku

Inicia sesión en Heroku desde la línea de comandos y crea una nueva aplicación.

```
heroku login
heroku create nombre-de-tu-aplicacion
```

Paso 4: Configurar Variables de Entorno

Configura las variables de entorno necesarias para tu aplicación.

```
heroku config:set SECRET_KEY=tu_clave_secreta
```

Paso 5: Desplegar la Aplicación

Inicializa un repositorio git (si aún no lo tienes) y realiza un commit de tus cambios.

```
git init
git add .
git commit -m "Preparar aplicación Flask para despliegue en Heroku"
```

Añade el remote de Heroku y despliega tu aplicación.

```
git remote add heroku https://git.heroku.com/nombre-de-tu-aplicacion.git
git push heroku master
```

Paso 6: Escalar la Aplicación

Escala tu aplicación para asegurarte de que al menos un dyno web esté en ejecución.

```
heroku ps:scale web=1
```

Paso 7: Abrir la Aplicación

Abre tu aplicación en un navegador web.

```
heroku open
```

Ejemplo de App Completa, con Mejores Prácticas y Profesional en Flask

Vamos a crear una aplicación Flask completa siguiendo las mejores prácticas y desplegarla en Heroku.

Estructura del Proyecto

```
/mi_app
    /static
    /templates
        base.html
        index.html
        login.html
        registro.html
    app.py
    requirements.txt
    Procfile
```

app.py Completo

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

```
from flask_sqlalchemy import SQLAlchemy
from flask_login import LoginManager, UserMixin, login_user, login_required, logout_user, current_user
from flask_bcrypt import Bcrypt
import os

app = Flask(__name__)
app.config['SECRET_KEY'] = os.environ.get('SECRET_KEY') or 'mi_secreto'
app.config['SQLALCHEMY_DATABASE_URI'] = os.environ.get('DATABASE_URL') or 'sqlite:///mi_base_de_datos.db'
db = SQLAlchemy(app)
bcrypt = Bcrypt(app)
login_manager = LoginManager()
login_manager.init_app(app)
login_manager.login_view = 'login'

class Usuario(UserMixin, db.Model):
    id = db.Column(db.Integer, primary_key=True)
    nombre = db.Column(db.String(100), nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
    password = db.Column(db.String(150), nullable=False)

@login_manager.user_loader
def cargar_usuario(usuario_id):
    return Usuario.query.get(int(usuario_id))

@app.before_first_request
def crear_tablas():
    db.create_all()

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/registro', methods=['GET', 'POST'])
def registro():
    if request.method == 'POST':
        nombre = request.form['nombre']
        email = request.form['email']
        password = request.form['password']
        hashed_password = bcrypt.generate_password_hash(password).decode('utf-8')
        nuevo_usuario = Usuario(nombre=nombre, email=email, password=hashed_password)
        db.session.add(nuevo_usuario)
        db.session.commit()
        return redirect(url_for('login'))
    return render_template('registro.html')

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']
        usuario = Usuario.query.filter_by(email=email).first()
        if usuario and bcrypt.check_password_hash(usuario.password, password):
            login_user(usuario)
            return redirect(url_for('index'))
    return render_template('login.html')

@app.route('/logout')
@login_required
```

```
60. def logout():
61.     logout_user()
62.     return redirect(url_for('login'))
63.
64. if __name__ == '__main__':
65.     app.run(debug=True)
```

Plantillas HTML

base.html

```
 1. <!DOCTYPE html>
 2. <html lang="es">
 3. <head>
 4.     <meta charset="UTF-8">
 5.     <title>{% block title %}{% endblock %}</title>
 6.     <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css">
 7. </head>
 8. <body>
 9.     <nav class="navbar navbar-expand-lg navbar-light bg-light">
10.         <a class="navbar-brand" href="{{ url_for('index') }}">MiApp</a>
11.         <div class="collapse navbar-collapse">
12.             <ul class="navbar-nav ml-auto">
13.                 {% if current_user.is_authenticated %}
14.                     <li class="nav-item"><a class="nav-link" href="{{ url_for('logout')
}}">Logout</a></li>
15.                 {% else %}
16.                     <li class="nav-item"><a class="nav-link" href="{{ url_for('login')
}}">Login</a></li>
17.                     <li class="nav-item"><a class="nav-link" href="{{ url_for('registro')
}}">Registro</a></li>
18.                 {% endif %}
19.             </ul>
20.         </div>
21.     </nav>
22.     <div class="container">
23.         {% block content %}{% endblock %}
24.     </div>
25. </body>
26. </html>
```

index.html

```
1. {% extends "base.html" %}
2. {% block title %}Inicio{% endblock %}
3. {% block content %}
4. <h1>Bienvenido a MiApp</h1>
5. <p>Esta es una aplicación de ejemplo desplegada en Heroku.</p>
6. {% endblock %}
```

login.html

```
{% extends "base.html" %}
{% block title %}Login{% endblock %}
{% block content %}
<h2>Login</h2>
<form method="post">
    <div class="form-group">
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" class="form-control" required>
    </div>
    <div class="form-group">
        <label for="password">Contraseña:</label>
        <input type="password" id="password" name="password" class="form-control"
required>
    </div>
    <button type="submit" class="btn btn-primary">Login</button>
</form>
{% endblock %}
```

registro.html

```
{% extends "base.html" %}
{% block title %}Registro{% endblock %}
{% block content %}
<h2>Registro</h2>
<form method="post">
    <div class="form-group">
        <label for="nombre">Nombre:</label>
        <input type="text" id="nombre" name="nombre" class="form-control" required>
    </div>
    <div class="form-group">
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" class="form-control" required>
    </div>
    <div class="form-group">
        <label for="password">Contraseña:</label>
        <input type="password" id="password" name="password" class="form-control"
required>
    </div>
    <button type="submit" class="btn btn-primary">Registrarse</button>
</form>
{% endblock %}
```

FIN DEL TEMARIO……..