

2. Manejo de Jinja2 y Contenido de HTML en Flask

Inicio



Manejando Jinja2 y Contenido HTML en Flask



1. Introducción a Jinja2



SECRETARÍA DE
INNOVACIÓN



ESIT
ESCUELA SUPERIOR
de Innovación
y Tecnología

Introducción a Jinja2



Contenido

Jinja2 es un motor de plantillas potente y flexible que se utiliza con frecuencia en el marco de trabajo Flask en Python para generar páginas HTML dinámicas. Su principal objetivo es facilitar la combinación de lógica y presentación, permitiendo a los desarrolladores injectar datos dinámicos en las plantillas HTML.

Características de Jinja2

Lógica de Plantillas:

Soporta estructuras de control como bucles (for) y condiciones (if).

Permite definir y reutilizar bloques de contenido mediante bloques y extensiones.

Filtros:

Incluye filtros predefinidos para manipular datos directamente en las plantillas, como upper, lower, length, etc.

Se pueden crear filtros personalizados.

Seguridad:

Escapa automáticamente caracteres peligrosos en datos para prevenir ataques como XSS (Cross-Site Scripting).

Modularidad:

Soporta herencia de plantillas para dividir las vistas en secciones reutilizables.

Usos Comunes de Jinja2 con Flask

Flask utiliza Jinja2 por defecto para renderizar plantillas. A través de la función render_template, puedes pasar datos dinámicos al HTML.

Ejemplo:

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/")
def home():
    user = {"name": "Juan", "age": 30}
    return render_template("index.html", user=user)

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

Plantilla index.html:

```
<!DOCTYPE html>
<html>
<head>
    <title>Inicio</title>
</head>
<body>
    <h1>Bienvenido, {{ user.name }}!</h1>
    <p>Tienes {{ user.age }} años.</p>
</body>
</html>
```

2. Herencia de Plantillas

Esto es útil para reutilizar elementos comunes como encabezados y pies de página.

Plantilla base (base.html):

```

<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}Mi Sitio{% endblock %}</title>
</head>
<body>
    <header>
        <h1>Cabecera Común</h1>
    </header>
    <main>
        {% block content %}{% endblock %}
    </main>
    <footer>
        <p>Pie de página común</p>
    </footer>
</body>
</html>

```

Plantilla hija (index.html):

```

{% extends "base.html" %}

{% block title %}Inicio{% endblock %}

{% block content %}
    <p>Contenido único para esta página.</p>
{% endblock %}

```

3. Uso de Filtros y Control de Flujo

Puedes modificar la presentación de datos directamente en la plantilla.

Ejemplo:

```
<p>Nombre en mayúsculas: {{ user.name | upper }}</p>
```

Control de Flujo: Se pueden emplear estructuras condicionales y bucles.

Ejemplo:

```
<ul>
    {% for item in items %}
        <li>{{ item }}</li>
    {% endfor %}
</ul>

{% if user.is_authenticated %}
    <p>Bienvenido, {{ user.name }}!</p>
{% else %}
    <p>Por favor, inicia sesión.</p>
{% endif %}
```

2. Manejo de Rutas en Flask



SECRETARÍA DE
INNOVACIÓN



ESIIT
ESCUELA SUPERIOR
de Innovación
y Tecnología

Manejo de las Rutas



Contenido

El manejo de rutas en Flask es uno de los pilares del framework, ya que permite definir cómo se asignan las URLs de una aplicación a funciones específicas (también conocidas como "vistas"). Esto facilita el desarrollo de aplicaciones web de manera estructurada y eficiente.

Conceptos Básicos de Rutas en Flask

- Definición de Rutas:** Flask utiliza decoradores como `@app.route()` para asignar una URL a una función.
- Métodos HTTP:** Las rutas pueden aceptar diferentes métodos HTTP como GET, POST, PUT, DELETE, etc.
- Variables en Rutas:** Es posible capturar parámetros de las URLs y pasarlos como argumentos a las funciones.
- Estructuración de Rutas:** Las rutas pueden organizarse en módulos o blueprints para mantener el código limpio y modular.

Ejemplos de Manejo de Rutas

1. Ruta Básica

Se asocia una URL específica con una función que devuelve una respuesta.

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def home():
    return "¡Bienvenido a la página principal!"

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

2. Rutas con Variables

Puedes capturar partes dinámicas de la URL y utilizarlas dentro de tu función.

```
@app.route("/usuario/<nombre>")
def saludar_usuario(nombre):
    return f"Hola, {nombre}!"

@app.route("/producto/<int:id>")
def mostrar_producto(id):
    return f"Mostrando producto con ID: {id}"
```

Ejemplos de URLs:

- /usuario/Juan devuelve: Hola, Juan!
- /producto/42 devuelve: Mostrando producto con ID: 42

3. Métodos HTTP

Por defecto, Flask acepta solo GET. Puedes habilitar otros métodos como POST, PUT o DELETE.

```
from flask import request

@app.route("/formulario", methods=["GET", "POST"])
def manejar_formulario():
    if request.method == "POST":
```

```
    datos = request.form["nombre"]
    return f"Recibido: {datos}"
return ''
<form method="post">
    Nombre: <input type="text" name="nombre">
    <input type="submit">
</form>
'''
```

4. Rutas con Múltiples Métodos y Parámetros Opcionales

Una sola ruta puede manejar varios métodos HTTP.

```
@app.route("/recurso", methods=["GET", "POST"])
def recurso():
    if request.method == "POST":
        return "Método POST recibido"
    return "Método GET recibido"
```

Parámetros Opcionales: Puedes definir valores por defecto para parámetros dinámicos.

```
@app.route("/saludo/<nombre>/<int:edad>", defaults={"edad": 18})
@app.route("/saludo/<nombre>/<int:edad>")
def saludo(nombre, edad):
    return f"Hola {nombre}, tienes {edad} años."
```

5. Estructuración de Rutas con Blueprints

Los Blueprints son una forma de dividir tu aplicación en módulos más pequeños y organizados.

Archivo principal (app.py):

```
from flask import Flask
from rutas.usuarios import usuarios_blueprint
```

```
app = Flask(__name__)
app.register_blueprint(usuarios_blueprint, url_prefix="/usuarios")

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

Archivo de rutas (rutas/usuarios.py):

```
from flask import Blueprint

usuarios_blueprint = Blueprint("usuarios", __name__)

@usuarios_blueprint.route("/")
def listar_usuarios():
    return "Lista de usuarios"

@usuarios_blueprint.route("/<int:id>")
def mostrar_usuario(id):
    return f"Usuario con ID: {id}"
```

6. Manejo de Errores con Rutas

Puedes manejar rutas no encontradas o errores de forma personalizada.

```
@app.errorhandler(404)
def pagina_no_encontrada(error):
    return "Página no encontrada, vuelve a intentarlo.", 404

@app.errorhandler(500)
def error_interno(error):
    return "Ocurrió un error interno.", 500
```

Buenas Prácticas para el Manejo de Rutas

Nombres Descriptivos: Utiliza nombres significativos para tus funciones y URLs.

Organización: Divide las rutas en módulos o blueprints en aplicaciones grandes.

Uso de Métodos HTTP: Aprovecha métodos HTTP para seguir el estándar RESTful en tus APIs.

Parámetros Tipados: Especifica tipos para los parámetros de la URL (<int:id>, <string:nombre>) para evitar errores inesperados.

Evitar Sobrecarga en una Ruta: No combines demasiada lógica en una sola función. Mantén las vistas simples y reutilizables.

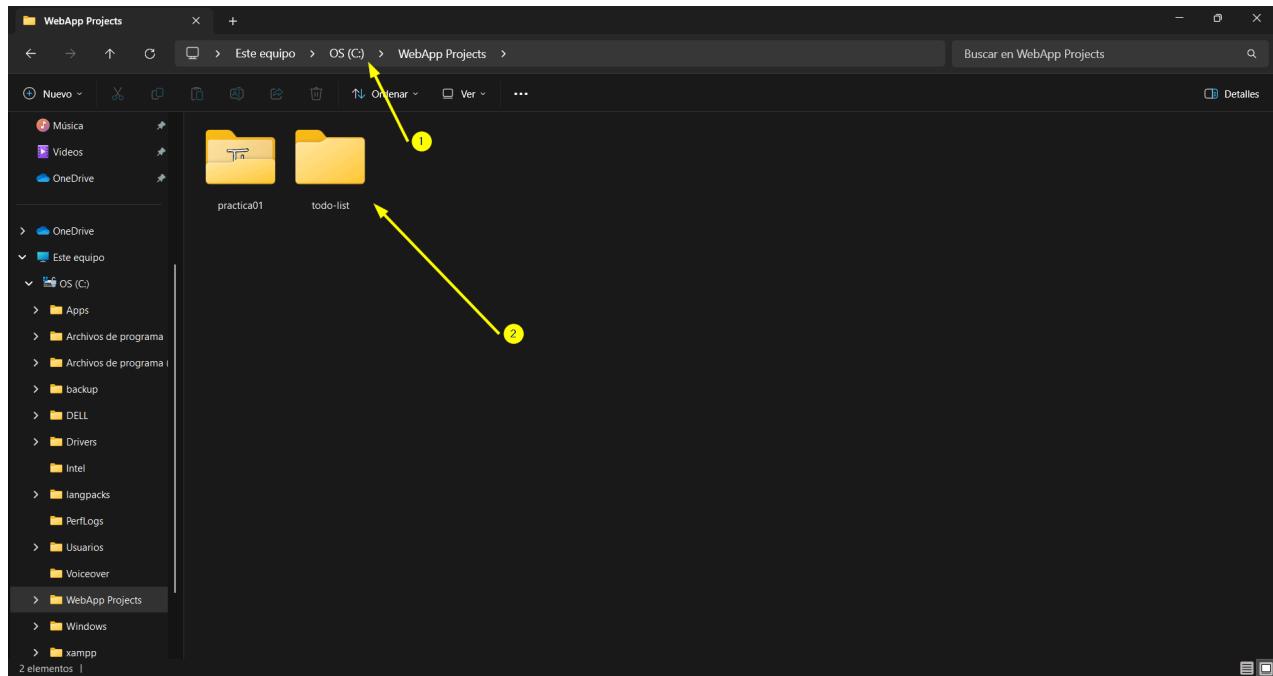
3. Creando la Estructura de un Proyecto en Flask



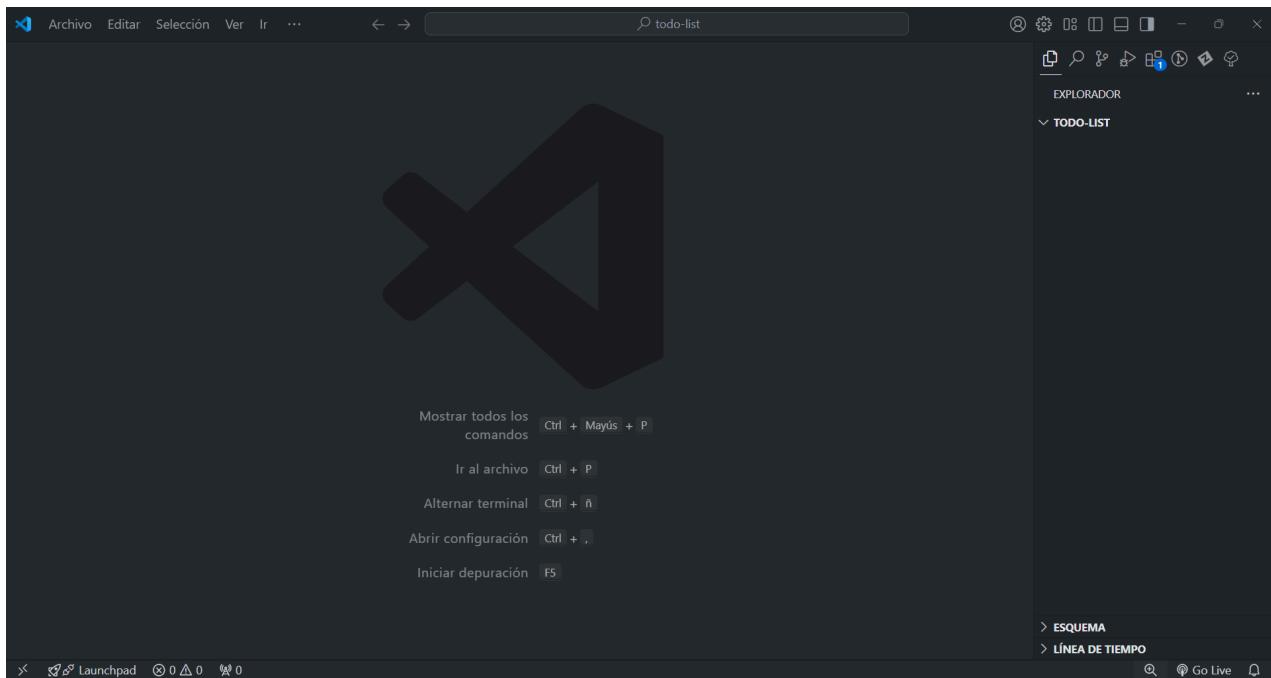
Contenido

1. Creando carpeta del proyecto

Vamos a crear la carpeta de nuestro proyecto, tendrá por nombre todo-list, para nuestro caso, reestructuraremos el contenido de WebApp Projects, creado en el material pasado, creamos un directorio llamado practica01 para guardar todo los archivos usados en el material anterior. En tu caso, puedes realizar el mismo proceso, o puedes crear el directorio por aparte, al final debe verse muy similar a lo siguiente:



Abrimos el directorio todo-list en Visual Studio Code:

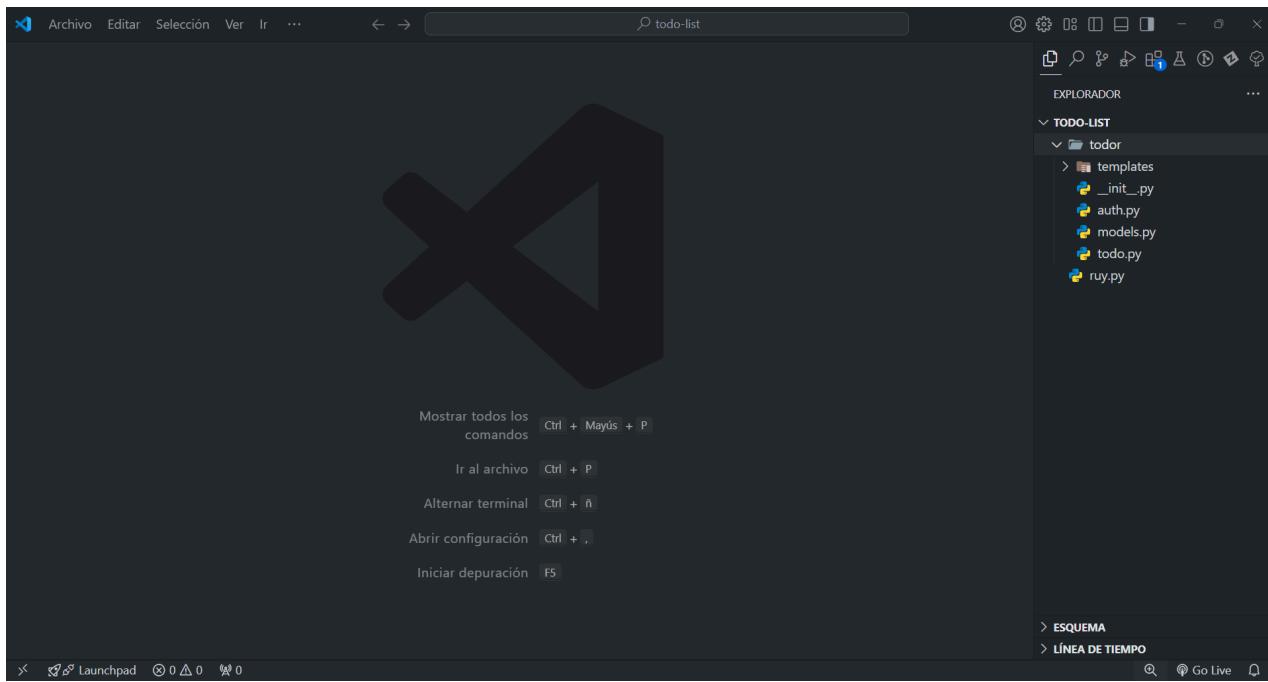


3. Estructura del Proyecto

La estructura del proyecto será de la siguiente manera:

```
todo-list/
|
└── todor/
    |
    ├── templates/      # Directorio que contendrá plantillas en HTML
    ├── __init__.py     # Archivo de inicialización del módulo
    ├── auth.py         # Archivo para autenticación de usuarios
    ├── models.py       # Archivo para definir modelos de datos
    ├── todo.py         # Archivo relacionado con las tareas o funcionali
    └── ruy.py          # Archivo de arranque
```

Se deberá ver en VSCode de la siguiente manera:



4. Configuración Inicial

Procedemos a configurar el archivo `__init__.py`:

```
# Importando la clase Flask
from flask import Flask

# Creando la variable de iniciación
app = Flask(__name__)

# Definiendo rutas
@app.route('/')
def index():
    return 'Hola Mundo'
```

Ahora, para rectificar el funcionamiento, usaremos el siguiente comando en la terminal de VSCode:

```
flask --app todor --debug run
```

The screenshot shows the Visual Studio Code interface with the following details:

- Editor Area:** Displays the file `__init__.py` containing the code for a simple Flask application.
- Terminal:** Shows the output of running the application with `flask --app todor --debug run`. It includes a warning about using it in production and logs for two requests to the root URL.
- Explorador (File Explorer):** Shows the project structure with files like `__init__.py`, `auth.py`, `models.py`, `todo.py`, and `ruy.py`.
- Bottom Status Bar:** Includes icons for Launchpad, status indicators (0△0, 0), and file navigation.

5. Primeros Ajustes

Crearemos una función global para control la ejecución de la app, así como la configuración del proyecto:

```
# Importando la clase Flask
from flask import Flask

# Creando función de control
def create_app():

    # Creando la variable de iniciación
    app = Flask(__name__)

    # Configuración del proyecto
    app.config.from_mapping(
        DEBUG = True,
        SECRET_KEY = 'dev_esit'
    )

    # Definiendo rutas
    @app.route('/')
    def index():
        return 'Hola Mundo'
```

```
return app
```

6. Configurando run.py

Procedemos a configurar la ejecución del proyecto sobre run.py:

```
# Importando __init__.py
from todor import create_app

# Validación de la función create_app
if __name__ == '__main__':
    app = create_app()
    app.run()
```

Ejecutamos en la terminal:

```
python run.py
```

The screenshot shows the Visual Studio Code interface. The left sidebar displays a file tree for a project named 'TODO-LIST' containing files like __init__.py, auth.py, models.py, todo.py, and run.py. The main editor area shows the code for run.py. Below the editor is the terminal pane, which shows the command 'python run.py' being run and the output of the Flask development server starting up on port 5000.

```
PS C:\WebApp Projects\todo-list> python run.py
 * Serving Flask app 'todor'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 122-941-379
127.0.0.1 - - [18/Dec/2024 11:36:28] "GET / HTTP/1.1" 200 -
```

7. Trabajando con Blueprint y vistas

Abrimos el archivo todo.py y procedemos a configurar Blueprint:

```
# Importando Blueprint
from flask import Blueprint

# Creando instancia
bp = Blueprint('todo', __name__, url_prefix='/todo')

#Creado ruta y función
@bp.route('/list')
def index():
    return "Lista de tareas"

@bp.route('/create')
def create():
    return "Crear una tarea"
```

Para que la configuración de Blueprint funcione, iremos al archivo `__init__.py` para agregar la configuración:

```
4 # Creando función de control
5 def create_app():
6
7     # Creando la variable de iniciación
8     app = Flask(__name__)
9
10    # Configuración del proyecto
11    app.config.from_mapping(
12        DEBUG = True,
13        SECRET_KEY = 'dev_esit'
14    )
15
16    # Registrando Blueprint
17    from . import todo
18    app.register_blueprint(todo.bp)
19
20    # Definiendo rutas
21    @app.route('/')
22    def index():
23        return 'Hola Mundo'
24
25    return app
```

8. Prueba de Blueprint

Si ejecutamos el servidor con `python run.py`, veremos los siguiente:



Si escribimos en el cuadro de la URL: `127.0.0.1:5000/todo/list`



Notarás que ya tenemos acceso a la vista mediante el manejo de rutas por Blueprint, prueba accediendo a `create`, la segunda vista creada.

9. Configurando Blueprint para autenticación

Abrimos el archivo `auth.py` y configuraremos Blueprint de la siguiente manera:

```
# Importando Blueprint
from flask import Blueprint
```

```

# Creando instancia
bp = Blueprint('auth', __name__, url_prefix='/auth')

#Creado ruta y función
@bp.route('/register')
def register():
    return "Registrar usuario"

@bp.route('/login')
def login():
    return "Iniciar sesión"

```

Registraremos el Blueprint de auth.py:

```

10     # Configuración del proyecto
11     app.config.from_mapping(
12         DEBUG = True,
13         SECRET_KEY = 'dev_esit'
14     )
15
16     # Registrando Blueprint
17     from . import todo
18     app.register_blueprint(todo.bp)
19
20     from . import auth
21     app.register_blueprint(auth.bp)
22
23     # Definiendo rutas
24     @app.route('/')
25     def index():
26         return 'Hola Mundo'
27
28     return app

```

Ahora, revisa el navegador, recuerda que si el servidor se detiene de forma automática, puedes reiniciarlo con el comando `python run.py`

4. Creado Plantillas con Jinja en Flask Parte 1



Creando Plantillas

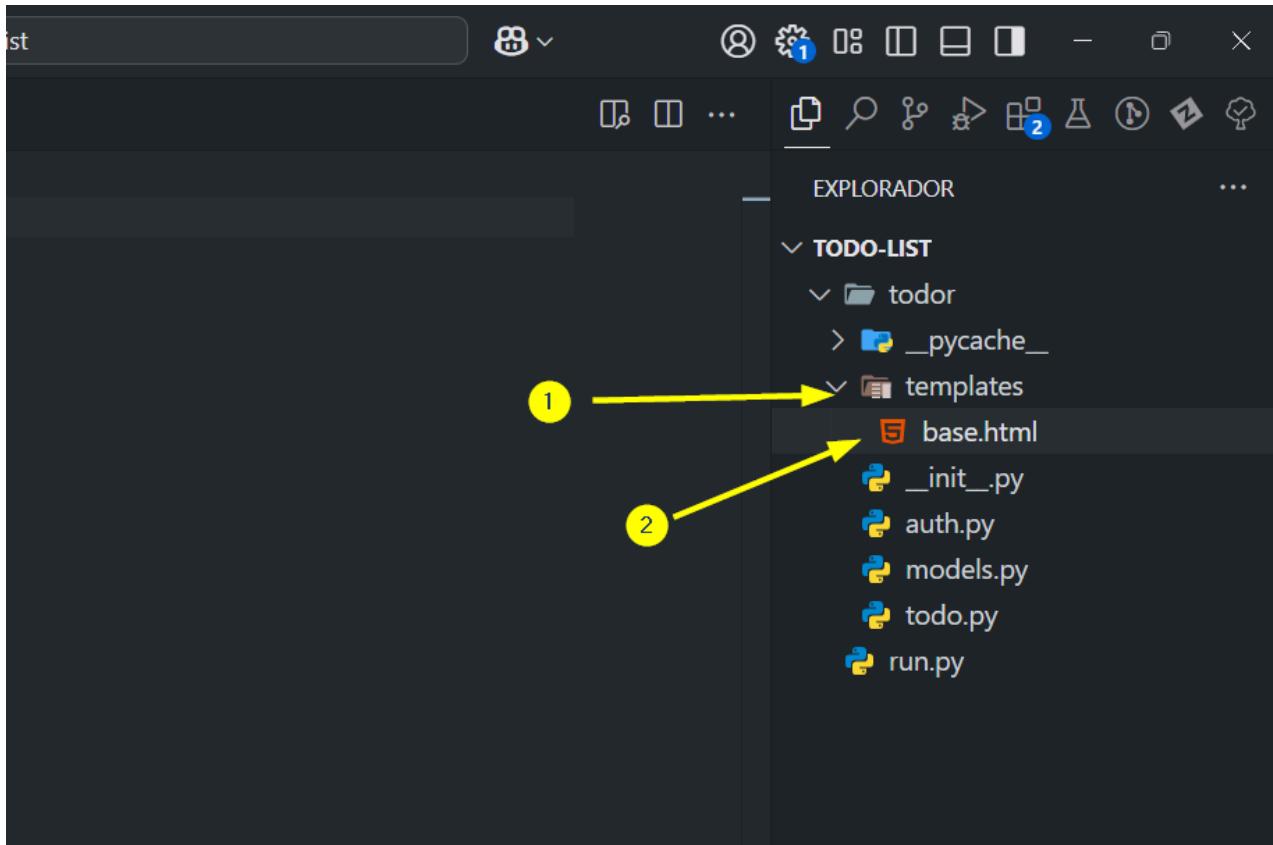


Contenido

1. Configuración inicial

Para configurar la vista principal, nos apoyaremos del framework de Bootstrap para CSS, nos facilitará el diseño visual de nuestras vistas, [puede hacer clic aquí <https://getbootstrap.com/docs/5.3/getting-started/introduction/>](https://getbootstrap.com/docs/5.3/getting-started/introduction/) para leer y explorar la documentación oficial del framework.

Iremos a la carpeta de templates, y dentro de ella creamos un archivo HTML llamado base.html:



2. Contenido de base.html

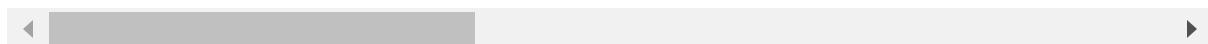
Agregamos el siguiente contenido al archivo base.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css">
    <title>TodoList - {% block title %}{% endblock%}</title>
</head>
<body>

    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js">
</body>
</html>
```

Ahora vamos a agregar el menú:

```
<nav class="navbar navbar-expand-lg bg-body-tertiary">
  <div class="container-fluid">
    <a class="navbar-brand" href="#">Navbar</a>
    <button class="navbar-toggler" type="button" data-bs-toggle=""
      <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="#">
        </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>
        <li class="nav-item">
          <a class="nav-link disabled" aria-disabled="true">Disab
        </li>
      </ul>
    </div>
  </div>
</nav>
```

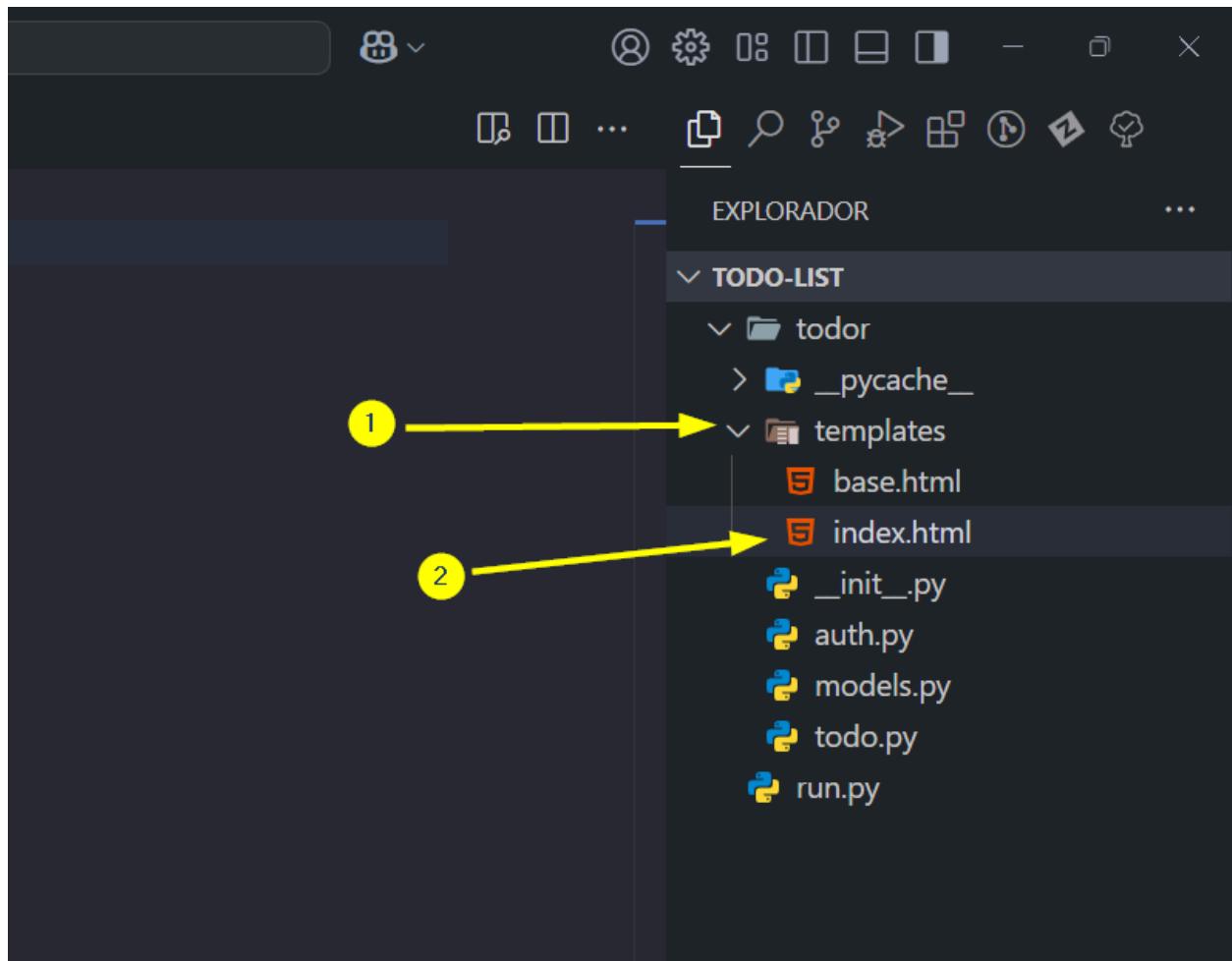


Finalizamos agregando la lectura de contenido:

```
todor > templates > base.html > html > body > nav.navbar.navbar-expand-lg.bg-body-tertiary
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
4       <meta charset="UTF-8">
5       <meta name="viewport" content="width=device-width, initial-scale=1.0">
6       <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css">
7       <title>TodoList - {% block title %}{% endblock %}</title>
8   </head>
9   <body>
10  >     <nav class="navbar navbar-expand-lg bg-body-tertiary"> ...
11  </nav>
12
13      [% block content %]
14      [% endblock %]
15
16      <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js">
17  </script>
18  </body>
19
20  </html>
```

3. Creando archivo index.html

Siempre en la carpeta templates, creamos un archivo llamado index.html:



4. Configurando index.html

Procedemos a crear la configuración de herencia y contenido en index.html:

```
{% extends 'base.html' %}

{% block title %}Inicio{% endblock %}

{% block content %}

<h1>Página de Inicio</h1>

{% endblock %}
```

1. {% extends 'base.html' %}

- Esta línea indica que la plantilla actual hereda de otra plantilla llamada base.html.
- Permite reutilizar una estructura común (como encabezados, pie de página, navegación, etc.) definida en base.html, evitando repetir código.

2. {% block title %}Inicio{% endblock %}

- Define un bloque llamado title, que es una sección específica de la plantilla que puede ser personalizada.
- Aquí, el contenido del bloque title se reemplaza con Inicio.
- Suele usarse para personalizar el título del navegador o metadatos.

3. {% block content %} ... {% endblock %}

- Define un bloque llamado content, que es otra sección personalizable.
- El contenido del bloque incluye un encabezado <h1> con el texto "Página de Inicio".
- En la plantilla base.html, el bloque content actúa como un área principal donde cada página define su propio contenido.

5. Configuración ruta hacia index.html

Abrimos __init__.py para configurar la renderización hacia el archivo index.html:

Primero importamos render_template

```
todor > _init_.py > ...
1 # Importando la clase Flask
2 from flask import Flask, render_template
3
4 # Creando función de control
5 def create_app():
6
7     # Creando la variable de iniciación
8     app = Flask(__name__)
9
```

Luego cambiamos por render_template hacia index.html:

```
16     # Registrando Blueprint
17     from . import todo
18     app.register_blueprint(todo.bp)
19
20     from . import auth
21     app.register_blueprint(auth.bp)
22
23     # Definiendo rutas
24     @app.route('/')
25     def index():
26         return render_template('index.html')
27
28     return app
```

6. Prueba en el servidor

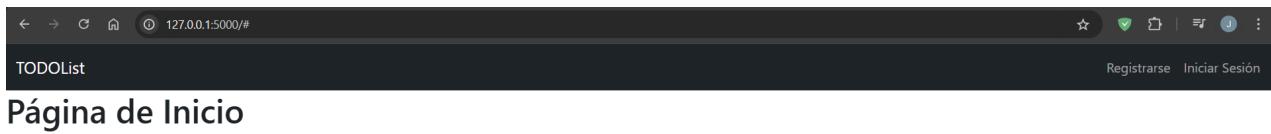
Usamos nuevamente el comando `python run.py` para levantar el servidor local y revisamos el navegador:



The screenshot shows a browser window with a dark-themed navbar at the top. The navbar includes links for 'Home', 'Features', 'Pricing', and 'Disabled'. Below the navbar, the main content area displays the heading 'Página de Inicio'.

```
<!-- Menú -->
<nav class="navbar navbar-expand-sm navbar-dark bg-dark data-bs-the
  <div class="container-fluid">
    <a class="navbar-brand" href="{{url_for('index')}}">TODOList<
    <button class="navbar-toggler" type="button" data-bs-toggle="
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse justify-content-end" id=
      <ul class="navbar-nav">
        <li class="nav-item">
          <a class="nav-link" href="#">Registrarse</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="#">Iniciar Sesión</a>
        </li>
      </ul>
    </div>
  </div>
</nav>
```

Al verlo en el navegador, tendremos lo siguiente:



8. Modificando index.html

Procedemos a modificar el contenido de index.html:

```
{% extends 'base.html' %}

{% block title %}Inicio{% endblock %}

{% block content %}

<div class="container-sm text-center">
    <h1 class="display-3">Bienvenido a nuestra lista de tareas</h1>

    <p class="lead">
        Aquí puedes llevar un registro de todas tus tareas y tener una
    </p>

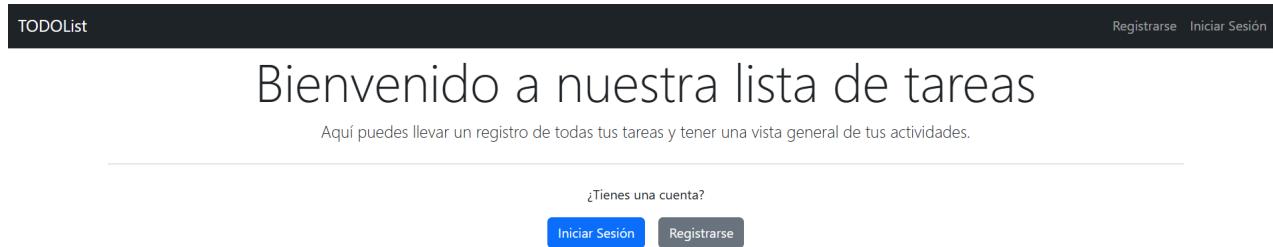
    <hr class="my-4">

    <p>¿Tienes una cuenta?</p>

    <div class="d-flex justify-content-center">
        <a href="" class="btn btn-primary mx-2">Iniciar Sesión</a>
        <a href="" class="btn btn-secondary mx-2">Registrarse</a>
    </div>
```

```
</div>  
  
{% endblock %}
```

En el navegador veremos lo siguiente:



5. Creado Plantillas con Jinja en Flask Parte 2



Creando Plantillas

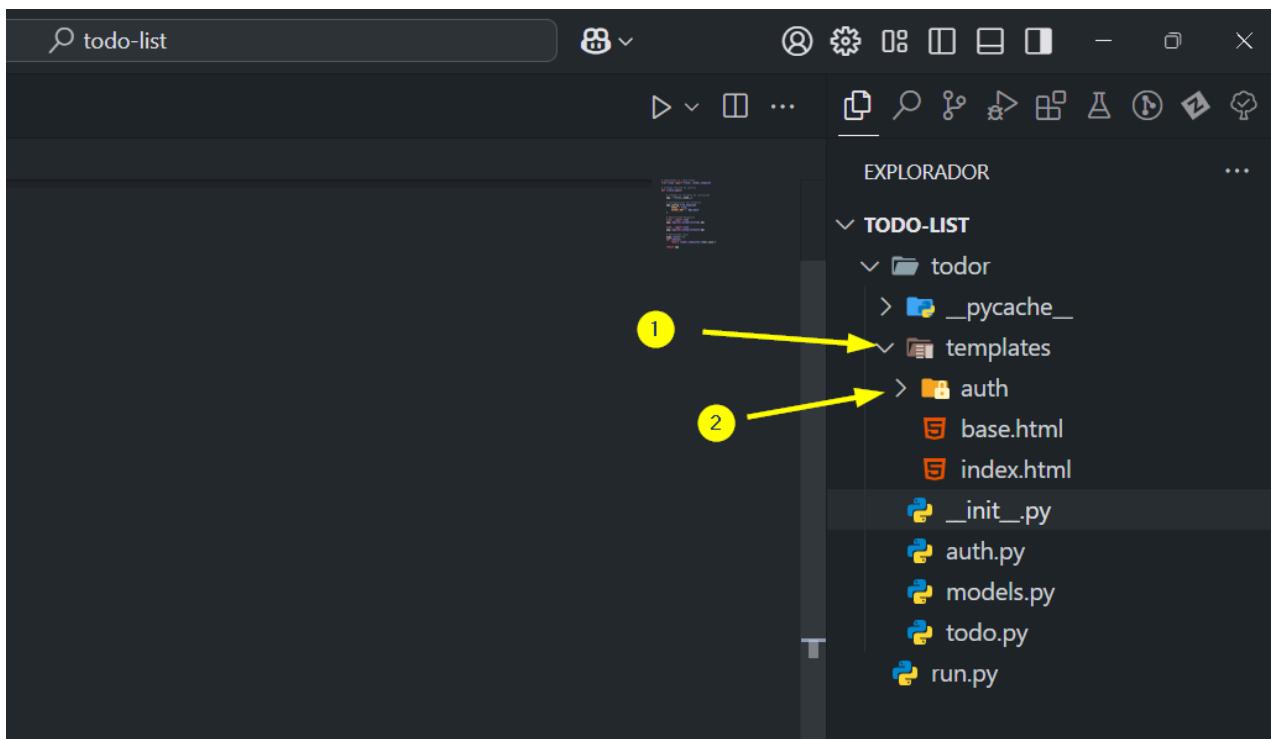


Contenid

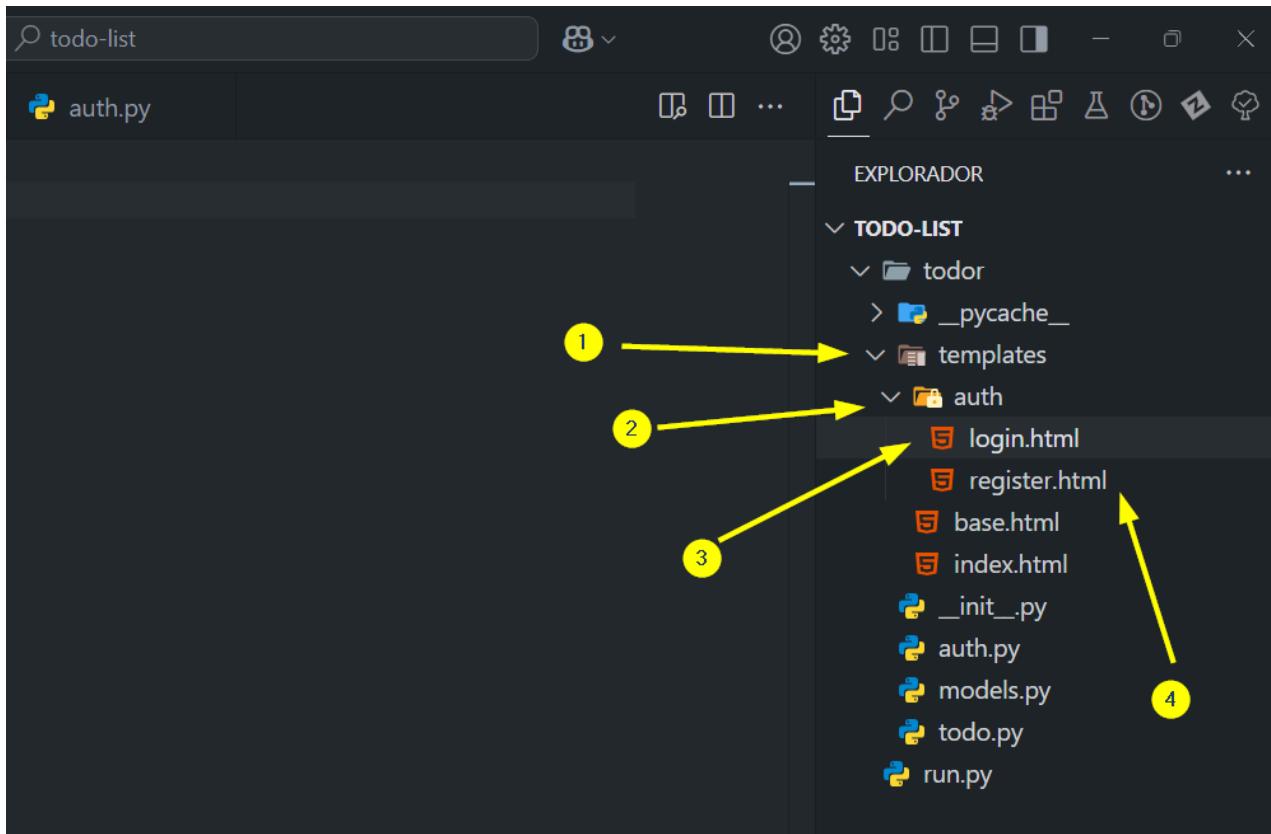
1. Configuración inicial

Vamos a crear la plantilla base de para registrar usuarios y la vista de login, en la sección anterior, tuvimos nuestro primer acercamiento con Jinja2, que nos facilita la creación de contenido base.

Vamos a la carpeta templates, y creamos una carpeta llamada auth:



Dentro de la carpeta auth, creamos 2 archivos, uno llamado register.html y uno llamado login.html



3. Renderización a register y login

Abrimos el archivo auth.py y configuramos la renderización:

```
# Importando Blueprint
from flask import Blueprint, render_template

# Creando instancia
bp = Blueprint('auth', __name__, url_prefix='/auth')

#Creado ruta y función
@bp.route('/register')
def register():
    return render_template('auth/register.html')

@bp.route('/login')
def login():
    return render_template('auth/login.html')
```

Vamos a base.html y agregamos la ruta de renderización:

```
<!-- Menu -->
<nav class="navbar navbar-expand-sm navbar-dark bg-dark data-bs-theme="dark">
  <div class="container-fluid">
    <a class="navbar-brand" href="{{url_for('index')}}">TODOList</a>
    <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target=
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse justify-content-end" id="navbarNav">
      <ul class="navbar-nav">
        <li class="nav-item">
          <a class="nav-link" href="{{url_for('auth.register')}}">Registrarse</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="{{url_for('auth.login')}}">Iniciar Sesión</a>
        </li>
      </ul>
    </div>
  </div>
</nav>
```

Vamos index.html y agregamos la configuración para la renderización:

```
10   <p class="lead">
11     |   Aquí puedes llevar un registro de todas tus tareas y tener una vista general de tus activ
12   </p>
13
14   <hr class="my-4">
15
16   <p>¿Tienes una cuenta?</p>
17
18   <div class="d-flex justify-content-center">
19     <a href="{{url_for('auth.login')}}" class="btn btn-primary mx-2">Iniciar Sesión</a>
20     <a href="{{url_for('auth.register')}}" class="btn btn-secondary mx-2">Registrarse</a>
21   </div>
22
23
24  {% endblock %}
```

4. Creando contenido de register.html

Abrimos register.html y creamos el contenido:

```
{% extends 'base.html' %}

{% block content %}

<div class="container mt-5">
  <div class="row justify-content-center">
    <div class="col-sm-8">
```

```

<div class="card">
    <div class="card-header">
        <h1 class="text-center">{% block title %}Registrar U
    </div>
    <div class="card-body">
        <form action="" method="post">
            <div class="form-group">
                <label for="username">Nombre de Usuario</label>
                <input type="text" class="form-control" id=
            </div>

            <div class="form-group">
                <label for="password">Contraseña</label>
                <input type="text" class="form-control" id=
            </div>

            <br>
            <div class="text-center">
                <input type="submit" class="btn btn-primary"
            </div>
            </form>
        </div>
    </div>

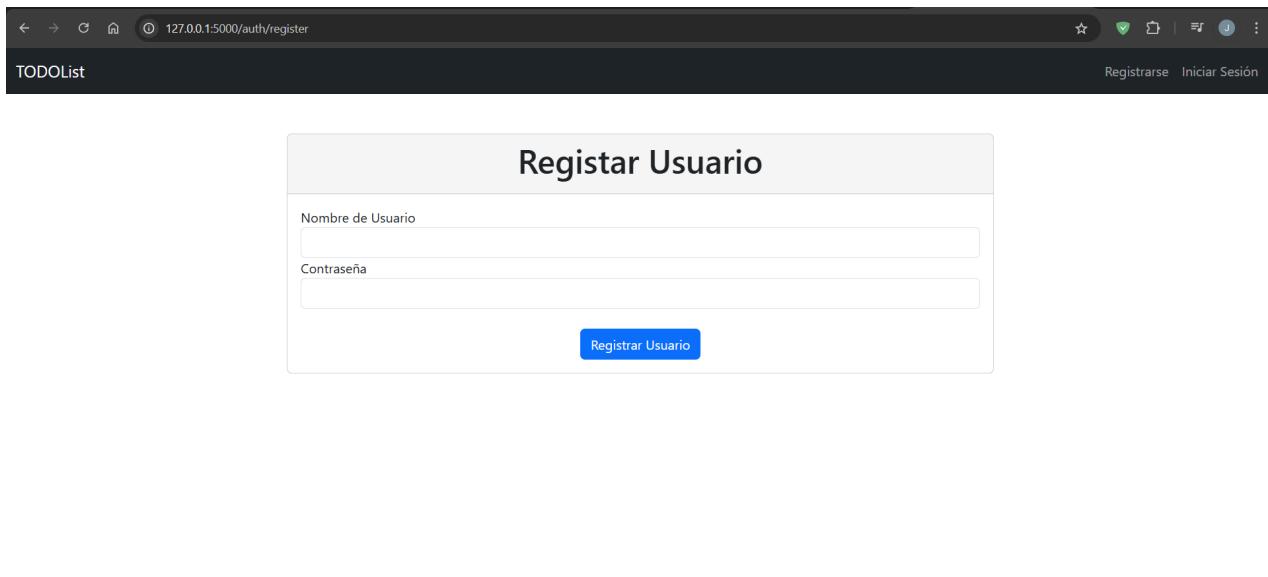
    </div>
</div>
</div>

{% endblock %}

```

5. Revisando la renderización

Si tenemos levantado el servidor, actualizamos el navegador, sino, usamos el comando `python ruy.py` para iniciar el servidor y revisamos en el navegador:



6. Creando contenido de login.html

Abrimos login.html y creamos el siguiente contenido:

```
{% extends 'base.html' %}

{% block content %}

<div class="container mt-5">
    <div class="row justify-content-center">
        <div class="col-sm-8">

            <div class="card">
                <div class="card-header">
                    <h1 class="text-center">{% block title %}Iniciar Us
                </div>
                <div class="card-body">
                    <form action="" method="post">
                        <div class="form-group">
                            <label for="username">Nombre de Usuario</la
                            <input type="text" class="form-control" id=
                        </div>

                        <div class="form-group">
                            <label for="password">Contraseña</label>
                            <input type="text" class="form-control" id=
```

```

        </div>

        <br>
        <div class="text-center">
            <input type="submit" class="btn btn-primary"
        </div>
        </form>
    </div>
</div>

</div>
</div>
</div>

{% endblock %}

```

7. Revisando la renderización

Si tenemos levantado el servidor, actualizamos el navegador, sino, usamos el comando `python ruy.py` para iniciar el servidor y revisamos en el navegador:

